Deep learning project

Department of Engineering - Aarhus University

June 3, 2017

Stud. no.: 201270782	Torben Werenberg Vogt
Stud. no.: 201270097	Simon Østergaard Kristensen
Stud. no.: 201270278	Ivan Bjerring Hansen

Contents

1 Introduction

2 Applied theory

3 Implementation

4 Results

5 Discussion

6 Conclusion

Bibliography

7 Exercise 1

Implementing a Neural Network

In this exercise we will develop a neural network with fully-connected layers to perform classification, and test it out on the CIFAR-10 dataset.

In [2]:
```
# A bit of setup

import numpy as np
import matplotlib.pyplot as plt

%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))))
```

The neural network parameters will be stored in a dictionary (`model` below), where the keys are the parameter names and the values are numpy arrays. Below, we initialize toy data and a toy model that we will use to verify your implementations.

In [3]:
```
# Create some toy data to check your implementations
input_size = 4
hidden_size = 10
num_classes = 3
num_inputs = 5

def init_toy_model():
  model = {}
  model['W1'] = np.linspace(-0.2, 0.6, num=input_size*hidden_size).reshape(inp
ut_size, hidden_size)
  model['b1'] = np.linspace(-0.3, 0.7, num=hidden_size)
  model['W2'] = np.linspace(-0.4, 0.1, num=hidden_size*num_classes).reshape(hi
dden_size, num_classes)
  model['b2'] = np.linspace(-0.5, 0.9, num=num_classes)
  return model

def init_toy_data():
  X = np.linspace(-0.2, 0.5, num=num_inputs*input_size).reshape(num_inputs, in
put_size)
  y = np.array([0, 1, 2, 2, 1])
  return X, y

model = init_toy_model()
X, y = init_toy_data()
```

Forward pass: compute scores

Open the file `cs231n/classifiers/neural_net.py` and look at the function `two_layer_net`. This function is very similar to the loss functions you have written for the SVM and Softmax exercises: It takes the data and weights and computes the class scores, the loss, and the gradients on the parameters.

Implement the first part of the forward pass which uses the weights and biases to compute the scores for all inputs.

In [4]:
```
from cs231n.classifiers.neural_net import two_layer_net

scores = two_layer_net(X, model)
print scores
correct_scores = [[-0.5328368, 0.20031504, 0.93346689],
 [-0.59412164, 0.15498488, 0.9040914 ],
 [-0.67658362, 0.08978957, 0.85616275],
 [-0.77092643, 0.01339997, 0.79772637],
 [-0.89110401, -0.08754544, 0.71601312]]

# the difference should be very small. We get 3e-8
print 'Difference between your scores and correct scores:'
print np.sum(np.abs(scores - correct_scores))
```

```
[[-0.5328368   0.20031504  0.93346689]
 [-0.59412164  0.15498488  0.9040914 ]
 [-0.67658362  0.08978957  0.85616275]
 [-0.77092643  0.01339997  0.79772637]
 [-0.89110401 -0.08754544  0.71601312]]
Difference between your scores and correct scores:
3.84868227808e-08
```

Forward pass: compute loss

In the same function, implement the second part that computes the data and regularizaion loss.

In [5]:
```
reg = 0.1
loss, _ = two_layer_net(X, model, y, reg)
correct_loss = 1.38191946092

# should be very small, we get 5e-12
print 'Difference between your loss and correct loss:'
print np.sum(np.abs(loss - correct_loss))
```

```
Difference between your loss and correct loss:
4.67736960275e-12
```

Backward pass

Implement the rest of the function. This will compute the gradient of the loss with respect to the variables `W1`, `b1`, `W2`, and `b2`. Now that you (hopefully!) have a correctly implemented forward pass, you can debug your backward pass using a numeric gradient check:

In [6]:
```
from cs231n.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward p
ass.
# If your implementation is correct, the difference between the numeric and
# analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b2.

loss, grads = two_layer_net(X, model, y, reg)

# these should all be less than 1e-8 or so
for param_name in grads:
  param_grad_num = eval_numerical_gradient(lambda W: two_layer_net(X, model,
y, reg)[0], model[param_name], verbose=False)
  print '%s max relative error: %e' % (param_name, rel_error(param_grad_num, g
rads[param_name]))
```

```
W1 max relative error: 4.426512e-09
W2 max relative error: 1.401432e-09
b2 max relative error: 7.311059e-11
b1 max relative error: 2.746125e-08
```

Train the network

To train the network we will use SGD with Momentum. Last assignment you implemented vanilla SGD. You will now implement the momentum update and the RMSProp update. Open the file `classifier_trainer.py` and familiarze yourself with the `ClassifierTrainer` class. It performs optimization given an arbitrary cost function data, and model. By default it uses vanilla SGD, which we have already implemented for you. First, run the optimization below using Vanilla SGD:

In [7]:
```
from cs231n.classifier_trainer import ClassifierTrainer

model = init_toy_model()
trainer = ClassifierTrainer()
# call the trainer to optimize the loss
# Notice that we're using sample_batches=False, so we're performing Gradient D
escent (no sampled batches of data)
best_model, loss_history, _, _ = trainer.train(X, y, X, y,
                                             model, two_layer_net,
                                             reg=0.001,
                                             learning_rate=1e-1, momentum=0.0,
 learning_rate_decay=1,
                                             update='sgd', sample_batches=Fals
e,
                                             num_epochs=100,
                                             verbose=False)
print 'Final loss with vanilla SGD: %f' % (loss_history[-1], )
```

```
starting iteration  0
starting iteration  10
starting iteration  20
starting iteration  30
starting iteration  40
starting iteration  50
starting iteration  60
starting iteration  70
starting iteration  80
starting iteration  90
Final loss with vanilla SGD: 0.940686
```

Now fill in the **momentum update** in the first missing code block inside the `train` function, and run the same optimization as above but with the momentum update. You should see a much better result in the final obtained loss:

In [8]:
```
model = init_toy_model()
trainer = ClassifierTrainer()
# call the trainer to optimize the loss
# Notice that we're using sample_batches=False, so we're performing Gradient D
escent (no sampled batches of data)
best_model, loss_history, _, _ = trainer.train(X, y, X, y,
                                             model, two_layer_net,
                                             reg=0.001,
                                             learning_rate=1e-1, momentum=0.9,
 learning_rate_decay=1,
                                             update='momentum',
sample_batches=False,
                                             num_epochs=100,
                                             verbose=False)
correct_loss = 0.494394
print 'Final loss with momentum SGD: %f. We get: %f' % (loss_history[-1], corr
ect_loss)
```

```
starting iteration  0
starting iteration  10
starting iteration  20
starting iteration  30
starting iteration  40
starting iteration  50
starting iteration  60
starting iteration  70
starting iteration  80
starting iteration  90
Final loss with momentum SGD: 0.494394. We get: 0.494394
```

Now also implement the **RMSProp** update rule inside the `train` function and rerun the optimization:

In [9]:
```
model = init_toy_model()
trainer = ClassifierTrainer()
# call the trainer to optimize the loss
# Notice that we're using sample_batches=False, so we're performing Gradient D
escent (no sampled batches of data)
best_model, loss_history, _, _ = trainer.train(X, y, X, y,
                                             model, two_layer_net,
                                             reg=0.001,
                                             learning_rate=1e-1, momentum=0.9,
 learning_rate_decay=1,
                                             update='rmsprop',
sample_batches=False,
                                             num_epochs=100,
                                             verbose=False)
correct_loss = 0.439368
print 'Final loss with RMSProp: %f. We get: %f' % (loss_history[-1], correct_l
oss)
```

```
starting iteration  0
starting iteration  10
starting iteration  20
starting iteration  30
starting iteration  40
starting iteration  50
starting iteration  60
starting iteration  70
starting iteration  80
starting iteration  90
Final loss with RMSProp: 1.125201. We get: 0.439368
```

Load the data

Now that you have implemented a two-layer network that passes gradient checks, it's time to load up our favorite CIFAR-10 data so we can use it to train a classifier.

In [10]:
```
from cs231n.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the two-layer neural net classifier. These are the same steps as
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

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

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis=0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image

    # Reshape data to rows
    X_train = X_train.reshape(num_training, -1)
    X_val = X_val.reshape(num_validation, -1)
    X_test = X_test.reshape(num_test, -1)

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


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
print 'Train data shape: ', X_train.shape
print 'Train labels shape: ', y_train.shape
print 'Validation data shape: ', X_val.shape
print 'Validation labels shape: ', y_val.shape
print 'Test data shape: ', X_test.shape
print 'Test labels shape: ', y_test.shape
```

```
Train data shape:  (49000L, 3072L)
Train labels shape:  (49000L,)
Validation data shape:  (1000L, 3072L)
Validation labels shape:  (1000L,)
Test data shape:  (1000L, 3072L)
Test labels shape:  (1000L,)
```

Train a network

To train our network we will use SGD with momentum. In addition, we will adjust the learning rate with an exponential learning rate schedule as optimization proceeds; after each epoch, we will reduce the learning rate by multiplying it by a decay rate.

In [11]:
```
from cs231n.classifiers.neural_net import init_two_layer_model

model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, numbe
r of classes
trainer = ClassifierTrainer()
best_model, loss_history, train_acc, val_acc = trainer.train(X_train, y_train,
 X_val, y_val,
                                             model, two_layer_net,
                                             num_epochs=5, reg=1.0,
                                             momentum=0.9, learning_rate_decay
 = 0.95,
                                             learning_rate=1e-5, verbose=True)
```

```
starting iteration  0
Finished epoch 0 / 5: cost 2.302593, train: 0.087000, val 0.082000, lr 1.0000
00e-05
starting iteration  10
starting iteration  20
starting iteration  30
starting iteration  40
starting iteration  50
starting iteration  60
starting iteration  70
starting iteration  80
starting iteration  90
starting iteration  100
starting iteration  110
starting iteration  120
starting iteration  130
starting iteration  140
starting iteration  150
starting iteration  160
starting iteration  170
starting iteration  180
starting iteration  190
starting iteration  200
starting iteration  210
starting iteration  220
starting iteration  230
starting iteration  240
starting iteration  250
starting iteration  260
starting iteration  270
starting iteration  280
starting iteration  290
starting iteration  300
starting iteration  310
starting iteration  320
starting iteration  330
starting iteration  340
starting iteration  350
starting iteration  360
starting iteration  370
starting iteration  380
starting iteration  390
starting iteration  400
starting iteration  410
starting iteration  420
starting iteration  430
starting iteration  440
starting iteration  450
starting iteration  460
starting iteration  470
starting iteration  480
Finished epoch 1 / 5: cost 2.268128, train: 0.158000, val 0.163000, lr 9.5000
00e-06
starting iteration  490
starting iteration  500
starting iteration  510
starting iteration  520
```

```
starting iteration  530
starting iteration  540
starting iteration  550
starting iteration  560
starting iteration  570
starting iteration  580
starting iteration  590
starting iteration  600
starting iteration  610
starting iteration  620
starting iteration  630
starting iteration  640
starting iteration  650
starting iteration  660
starting iteration  670
starting iteration  680
starting iteration  690
starting iteration  700
starting iteration  710
starting iteration  720
starting iteration  730
starting iteration  740
starting iteration  750
starting iteration  760
starting iteration  770
starting iteration  780
starting iteration  790
starting iteration  800
starting iteration  810
starting iteration  820
starting iteration  830
starting iteration  840
starting iteration  850
starting iteration  860
starting iteration  870
starting iteration  880
starting iteration  890
starting iteration  900
starting iteration  910
starting iteration  920
starting iteration  930
starting iteration  940
starting iteration  950
starting iteration  960
starting iteration  970
Finished epoch 2 / 5: cost 1.971051, train: 0.232000, val 0.243000, lr 9.0250
00e-06
starting iteration  980
starting iteration  990
starting iteration  1000
starting iteration  1010
starting iteration  1020
starting iteration  1030
starting iteration  1040
starting iteration  1050
starting iteration  1060
starting iteration  1070
```

```
starting iteration  1080
starting iteration  1090
starting iteration  1100
starting iteration  1110
starting iteration  1120
starting iteration  1130
starting iteration  1140
starting iteration  1150
starting iteration  1160
starting iteration  1170
starting iteration  1180
starting iteration  1190
starting iteration  1200
starting iteration  1210
starting iteration  1220
starting iteration  1230
starting iteration  1240
starting iteration  1250
starting iteration  1260
starting iteration  1270
starting iteration  1280
starting iteration  1290
starting iteration  1300
starting iteration  1310
starting iteration  1320
starting iteration  1330
starting iteration  1340
starting iteration  1350
starting iteration  1360
starting iteration  1370
starting iteration  1380
starting iteration  1390
starting iteration  1400
starting iteration  1410
starting iteration  1420
starting iteration  1430
starting iteration  1440
starting iteration  1450
starting iteration  1460
Finished epoch 3 / 5: cost 1.950968, train: 0.283000, val 0.298000, lr 8.5737
50e-06
starting iteration  1470
starting iteration  1480
starting iteration  1490
starting iteration  1500
starting iteration  1510
starting iteration  1520
starting iteration  1530
starting iteration  1540
starting iteration  1550
starting iteration  1560
starting iteration  1570
starting iteration  1580
starting iteration  1590
starting iteration  1600
starting iteration  1610
starting iteration  1620
```

```
starting iteration  1630
starting iteration  1640
starting iteration  1650
starting iteration  1660
starting iteration  1670
starting iteration  1680
starting iteration  1690
starting iteration  1700
starting iteration  1710
starting iteration  1720
starting iteration  1730
starting iteration  1740
starting iteration  1750
starting iteration  1760
starting iteration  1770
starting iteration  1780
starting iteration  1790
starting iteration  1800
starting iteration  1810
starting iteration  1820
starting iteration  1830
starting iteration  1840
starting iteration  1850
starting iteration  1860
starting iteration  1870
starting iteration  1880
starting iteration  1890
starting iteration  1900
starting iteration  1910
starting iteration  1920
starting iteration  1930
starting iteration  1940
starting iteration  1950
Finished epoch 4 / 5: cost 1.877080, train: 0.337000, val 0.344000, lr 8.1450
63e-06
starting iteration  1960
starting iteration  1970
starting iteration  1980
starting iteration  1990
starting iteration  2000
starting iteration  2010
starting iteration  2020
starting iteration  2030
starting iteration  2040
starting iteration  2050
starting iteration  2060
starting iteration  2070
starting iteration  2080
starting iteration  2090
starting iteration  2100
starting iteration  2110
starting iteration  2120
starting iteration  2130
starting iteration  2140
starting iteration  2150
starting iteration  2160
starting iteration  2170
```

```
starting iteration  2180
starting iteration  2190
starting iteration  2200
starting iteration  2210
starting iteration  2220
starting iteration  2230
starting iteration  2240
starting iteration  2250
starting iteration  2260
starting iteration  2270
starting iteration  2280
starting iteration  2290
starting iteration  2300
starting iteration  2310
starting iteration  2320
starting iteration  2330
starting iteration  2340
starting iteration  2350
starting iteration  2360
starting iteration  2370
starting iteration  2380
starting iteration  2390
starting iteration  2400
starting iteration  2410
starting iteration  2420
starting iteration  2430
starting iteration  2440
Finished epoch 5 / 5: cost 1.957579, train: 0.331000, val 0.376000, lr 7.7378
09e-06
finished optimization. best validation accuracy: 0.376000
```

Debug the training

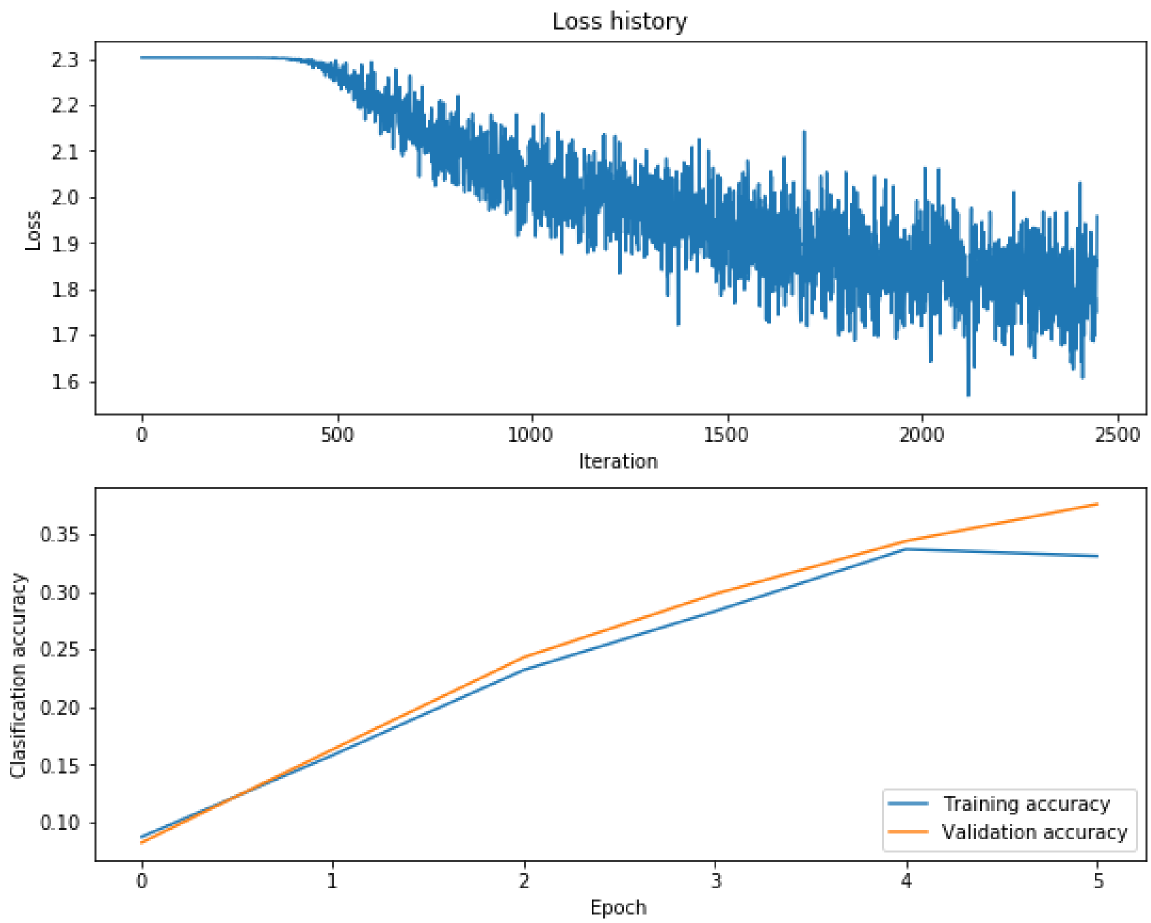
With the default parameters we provided above, you should get a validation accuracy of about 0.37 on the validation set. This isn't very good.

One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

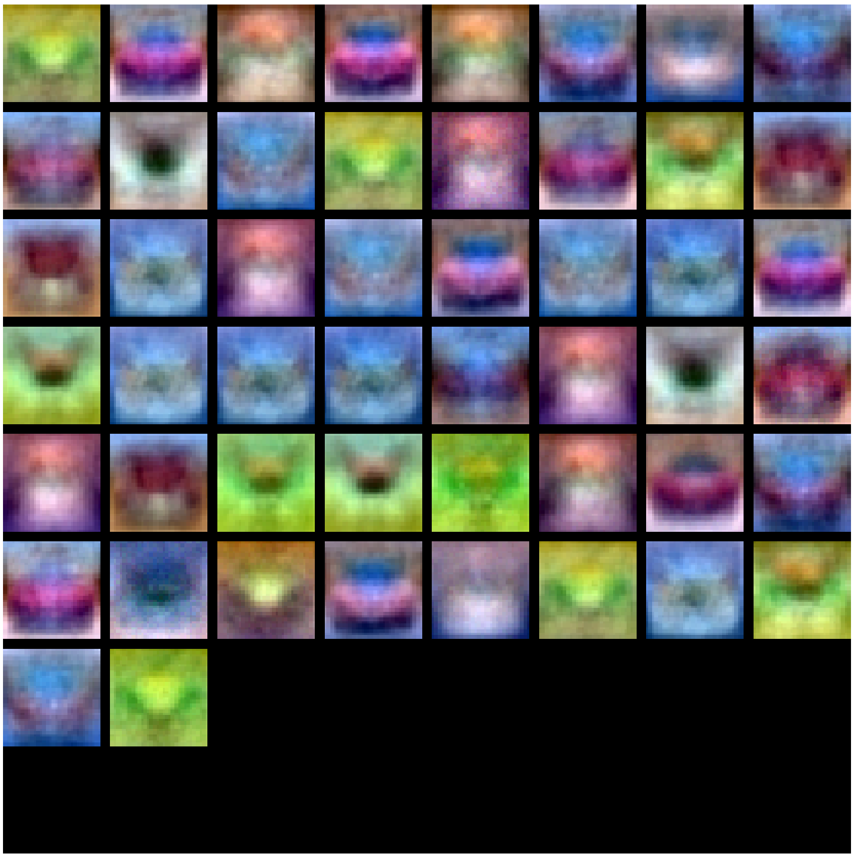
Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

In [12]:
```
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
plt.plot(loss_history)
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(2, 1, 2)
plt.plot(train_acc)
plt.plot(val_acc)
plt.legend(['Training accuracy', 'Validation accuracy'], loc='lower right')
plt.xlabel('Epoch')
plt.ylabel('Clasification accuracy')
```

Out[12]: <matplotlib.text.Text at 0x8053ef0>



In [13]:
```
from cs231n.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(model):
    plt.imshow(visualize_grid(model['W1'].T.reshape(-1, 32, 32, 3),
padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()

show_net_weights(model)
```



Tune your hyperparameters

What's wrong?. Looking at the visualizations above, we see that the loss is decreasing more or less linearly, which seems to suggest that the learning rate may be too low. Moreover, there is no gap between the training and validation accuracy, suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

Tuning. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, numer of training epochs, and regularization strength. You might also consider tuning the momentum and learning rate decay parameters, but you should be able to get good performance using the default values.

Approximate results. You should be aim to achieve a classification accuracy of greater than 50% on the validation set. Our best network gets over 56% on the validation set.

Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can, with a fully-connected Neural Network. For every 1% above 56% on the Test set we will award you with one extra bonus point. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

In [15]:
```
best_model = None # store the best model into this

################################################################################
###
# TODO: Tune hyperparameters using the validation set. Store your best trained
  #
# model in best_model.
  #
#
  #
# To help debug your network, it may help to use visualizations similar to the
  #
# ones we used above; these visualizations will have significant qualitative
  #
# differences from the ones we saw above for the poorly tuned network.
  #
#
  #
# Tweaking hyperparameters by hand can be fun, but you might find it useful to
  #
# write code to sweep through possible combinations of hyperparameters
  #
# automatically like we did on the previous assignment.
  #
################################################################################
###
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, numbe
r of classes
trainer = ClassifierTrainer()
best_model, loss_history, train_acc, val_acc = trainer.train(X_train, y_train,
 X_val, y_val,
                                             model, two_layer_net,
                                             num_epochs=28, reg=1.1,
                                             momentum=0.9, learning_rate_decay
 = 0.9,
                                             learning_rate=1.5e-4, verbose=Tru
e)
################################################################################
###
#                                END OF YOUR CODE
  #
################################################################################
###
```

```
starting iteration  0
Finished epoch 0 / 28: cost 2.302594, train: 0.132000, val 0.109000, lr 1.500
000e-04
starting iteration  10
starting iteration  20
starting iteration  30
starting iteration  40
starting iteration  50
starting iteration  60
starting iteration  70
starting iteration  80
starting iteration  90
starting iteration  100
starting iteration  110
starting iteration  120
starting iteration  130
starting iteration  140
starting iteration  150
starting iteration  160
starting iteration  170
starting iteration  180
starting iteration  190
starting iteration  200
starting iteration  210
starting iteration  220
starting iteration  230
starting iteration  240
starting iteration  250
starting iteration  260
starting iteration  270
starting iteration  280
starting iteration  290
starting iteration  300
starting iteration  310
starting iteration  320
starting iteration  330
starting iteration  340
starting iteration  350
starting iteration  360
starting iteration  370
starting iteration  380
starting iteration  390
starting iteration  400
starting iteration  410
starting iteration  420
starting iteration  430
starting iteration  440
starting iteration  450
starting iteration  460
starting iteration  470
starting iteration  480
Finished epoch 1 / 28: cost 1.677143, train: 0.432000, val 0.416000, lr 1.350
000e-04
starting iteration  490
starting iteration  500
starting iteration  510
starting iteration  520
```

```
starting iteration  530
starting iteration  540
starting iteration  550
starting iteration  560
starting iteration  570
starting iteration  580
starting iteration  590
starting iteration  600
starting iteration  610
starting iteration  620
starting iteration  630
starting iteration  640
starting iteration  650
starting iteration  660
starting iteration  670
starting iteration  680
starting iteration  690
starting iteration  700
starting iteration  710
starting iteration  720
starting iteration  730
starting iteration  740
starting iteration  750
starting iteration  760
starting iteration  770
starting iteration  780
starting iteration  790
starting iteration  800
starting iteration  810
starting iteration  820
starting iteration  830
starting iteration  840
starting iteration  850
starting iteration  860
starting iteration  870
starting iteration  880
starting iteration  890
starting iteration  900
starting iteration  910
starting iteration  920
starting iteration  930
starting iteration  940
starting iteration  950
starting iteration  960
starting iteration  970
Finished epoch 2 / 28: cost 1.483524, train: 0.496000, val 0.454000, lr 1.215
000e-04
starting iteration  980
starting iteration  990
starting iteration  1000
starting iteration  1010
starting iteration  1020
starting iteration  1030
starting iteration  1040
starting iteration  1050
starting iteration  1060
starting iteration  1070
```

```
starting iteration  1080
starting iteration  1090
starting iteration  1100
starting iteration  1110
starting iteration  1120
starting iteration  1130
starting iteration  1140
starting iteration  1150
starting iteration  1160
starting iteration  1170
starting iteration  1180
starting iteration  1190
starting iteration  1200
starting iteration  1210
starting iteration  1220
starting iteration  1230
starting iteration  1240
starting iteration  1250
starting iteration  1260
starting iteration  1270
starting iteration  1280
starting iteration  1290
starting iteration  1300
starting iteration  1310
starting iteration  1320
starting iteration  1330
starting iteration  1340
starting iteration  1350
starting iteration  1360
starting iteration  1370
starting iteration  1380
starting iteration  1390
starting iteration  1400
starting iteration  1410
starting iteration  1420
starting iteration  1430
starting iteration  1440
starting iteration  1450
starting iteration  1460
Finished epoch 3 / 28: cost 1.701114, train: 0.433000, val 0.462000, lr 1.093
500e-04
starting iteration  1470
starting iteration  1480
starting iteration  1490
starting iteration  1500
starting iteration  1510
starting iteration  1520
starting iteration  1530
starting iteration  1540
starting iteration  1550
starting iteration  1560
starting iteration  1570
starting iteration  1580
starting iteration  1590
starting iteration  1600
starting iteration  1610
starting iteration  1620
```

```
starting iteration  1630
starting iteration  1640
starting iteration  1650
starting iteration  1660
starting iteration  1670
starting iteration  1680
starting iteration  1690
starting iteration  1700
starting iteration  1710
starting iteration  1720
starting iteration  1730
starting iteration  1740
starting iteration  1750
starting iteration  1760
starting iteration  1770
starting iteration  1780
starting iteration  1790
starting iteration  1800
starting iteration  1810
starting iteration  1820
starting iteration  1830
starting iteration  1840
starting iteration  1850
starting iteration  1860
starting iteration  1870
starting iteration  1880
starting iteration  1890
starting iteration  1900
starting iteration  1910
starting iteration  1920
starting iteration  1930
starting iteration  1940
starting iteration  1950
Finished epoch 4 / 28: cost 1.704135, train: 0.492000, val 0.454000, lr 9.841
500e-05
starting iteration  1960
starting iteration  1970
starting iteration  1980
starting iteration  1990
starting iteration  2000
starting iteration  2010
starting iteration  2020
starting iteration  2030
starting iteration  2040
starting iteration  2050
starting iteration  2060
starting iteration  2070
starting iteration  2080
starting iteration  2090
starting iteration  2100
starting iteration  2110
starting iteration  2120
starting iteration  2130
starting iteration  2140
starting iteration  2150
starting iteration  2160
starting iteration  2170
```

```
starting iteration  2180
starting iteration  2190
starting iteration  2200
starting iteration  2210
starting iteration  2220
starting iteration  2230
starting iteration  2240
starting iteration  2250
starting iteration  2260
starting iteration  2270
starting iteration  2280
starting iteration  2290
starting iteration  2300
starting iteration  2310
starting iteration  2320
starting iteration  2330
starting iteration  2340
starting iteration  2350
starting iteration  2360
starting iteration  2370
starting iteration  2380
starting iteration  2390
starting iteration  2400
starting iteration  2410
starting iteration  2420
starting iteration  2430
starting iteration  2440
Finished epoch 5 / 28: cost 1.551213, train: 0.504000, val 0.486000, lr 8.857
350e-05
starting iteration  2450
starting iteration  2460
starting iteration  2470
starting iteration  2480
starting iteration  2490
starting iteration  2500
starting iteration  2510
starting iteration  2520
starting iteration  2530
starting iteration  2540
starting iteration  2550
starting iteration  2560
starting iteration  2570
starting iteration  2580
starting iteration  2590
starting iteration  2600
starting iteration  2610
starting iteration  2620
starting iteration  2630
starting iteration  2640
starting iteration  2650
starting iteration  2660
starting iteration  2670
starting iteration  2680
starting iteration  2690
starting iteration  2700
starting iteration  2710
starting iteration  2720
```

```
starting iteration  2730
starting iteration  2740
starting iteration  2750
starting iteration  2760
starting iteration  2770
starting iteration  2780
starting iteration  2790
starting iteration  2800
starting iteration  2810
starting iteration  2820
starting iteration  2830
starting iteration  2840
starting iteration  2850
starting iteration  2860
starting iteration  2870
starting iteration  2880
starting iteration  2890
starting iteration  2900
starting iteration  2910
starting iteration  2920
starting iteration  2930
Finished epoch 6 / 28: cost 1.593545, train: 0.485000, val 0.486000, lr 7.971
615e-05
starting iteration  2940
starting iteration  2950
starting iteration  2960
starting iteration  2970
starting iteration  2980
starting iteration  2990
starting iteration  3000
starting iteration  3010
starting iteration  3020
starting iteration  3030
starting iteration  3040
starting iteration  3050
starting iteration  3060
starting iteration  3070
starting iteration  3080
starting iteration  3090
starting iteration  3100
starting iteration  3110
starting iteration  3120
starting iteration  3130
starting iteration  3140
starting iteration  3150
starting iteration  3160
starting iteration  3170
starting iteration  3180
starting iteration  3190
starting iteration  3200
starting iteration  3210
starting iteration  3220
starting iteration  3230
starting iteration  3240
starting iteration  3250
starting iteration  3260
starting iteration  3270
```

```
starting iteration  3280
starting iteration  3290
starting iteration  3300
starting iteration  3310
starting iteration  3320
starting iteration  3330
starting iteration  3340
starting iteration  3350
starting iteration  3360
starting iteration  3370
starting iteration  3380
starting iteration  3390
starting iteration  3400
starting iteration  3410
starting iteration  3420
Finished epoch 7 / 28: cost 1.526852, train: 0.502000, val 0.496000, lr 7.174
453e-05
starting iteration  3430
starting iteration  3440
starting iteration  3450
starting iteration  3460
starting iteration  3470
starting iteration  3480
starting iteration  3490
starting iteration  3500
starting iteration  3510
starting iteration  3520
starting iteration  3530
starting iteration  3540
starting iteration  3550
starting iteration  3560
starting iteration  3570
starting iteration  3580
starting iteration  3590
starting iteration  3600
starting iteration  3610
starting iteration  3620
starting iteration  3630
starting iteration  3640
starting iteration  3650
starting iteration  3660
starting iteration  3670
starting iteration  3680
starting iteration  3690
starting iteration  3700
starting iteration  3710
starting iteration  3720
starting iteration  3730
starting iteration  3740
starting iteration  3750
starting iteration  3760
starting iteration  3770
starting iteration  3780
starting iteration  3790
starting iteration  3800
starting iteration  3810
starting iteration  3820
```

```
starting iteration  3830
starting iteration  3840
starting iteration  3850
starting iteration  3860
starting iteration  3870
starting iteration  3880
starting iteration  3890
starting iteration  3900
starting iteration  3910
Finished epoch 8 / 28: cost 1.754050, train: 0.488000, val 0.505000, lr 6.457
008e-05
starting iteration  3920
starting iteration  3930
starting iteration  3940
starting iteration  3950
starting iteration  3960
starting iteration  3970
starting iteration  3980
starting iteration  3990
starting iteration  4000
starting iteration  4010
starting iteration  4020
starting iteration  4030
starting iteration  4040
starting iteration  4050
starting iteration  4060
starting iteration  4070
starting iteration  4080
starting iteration  4090
starting iteration  4100
starting iteration  4110
starting iteration  4120
starting iteration  4130
starting iteration  4140
starting iteration  4150
starting iteration  4160
starting iteration  4170
starting iteration  4180
starting iteration  4190
starting iteration  4200
starting iteration  4210
starting iteration  4220
starting iteration  4230
starting iteration  4240
starting iteration  4250
starting iteration  4260
starting iteration  4270
starting iteration  4280
starting iteration  4290
starting iteration  4300
starting iteration  4310
starting iteration  4320
starting iteration  4330
starting iteration  4340
starting iteration  4350
starting iteration  4360
starting iteration  4370
```

```
starting iteration  4380
starting iteration  4390
starting iteration  4400
Finished epoch 9 / 28: cost 1.611458, train: 0.513000, val 0.486000, lr 5.811
307e-05
starting iteration  4410
starting iteration  4420
starting iteration  4430
starting iteration  4440
starting iteration  4450
starting iteration  4460
starting iteration  4470
starting iteration  4480
starting iteration  4490
starting iteration  4500
starting iteration  4510
starting iteration  4520
starting iteration  4530
starting iteration  4540
starting iteration  4550
starting iteration  4560
starting iteration  4570
starting iteration  4580
starting iteration  4590
starting iteration  4600
starting iteration  4610
starting iteration  4620
starting iteration  4630
starting iteration  4640
starting iteration  4650
starting iteration  4660
starting iteration  4670
starting iteration  4680
starting iteration  4690
starting iteration  4700
starting iteration  4710
starting iteration  4720
starting iteration  4730
starting iteration  4740
starting iteration  4750
starting iteration  4760
starting iteration  4770
starting iteration  4780
starting iteration  4790
starting iteration  4800
starting iteration  4810
starting iteration  4820
starting iteration  4830
starting iteration  4840
starting iteration  4850
starting iteration  4860
starting iteration  4870
starting iteration  4880
starting iteration  4890
Finished epoch 10 / 28: cost 1.579166, train: 0.502000, val 0.499000, lr 5.23
0177e-05
starting iteration  4900
```

```
starting iteration  4910
starting iteration  4920
starting iteration  4930
starting iteration  4940
starting iteration  4950
starting iteration  4960
starting iteration  4970
starting iteration  4980
starting iteration  4990
starting iteration  5000
starting iteration  5010
starting iteration  5020
starting iteration  5030
starting iteration  5040
starting iteration  5050
starting iteration  5060
starting iteration  5070
starting iteration  5080
starting iteration  5090
starting iteration  5100
starting iteration  5110
starting iteration  5120
starting iteration  5130
starting iteration  5140
starting iteration  5150
starting iteration  5160
starting iteration  5170
starting iteration  5180
starting iteration  5190
starting iteration  5200
starting iteration  5210
starting iteration  5220
starting iteration  5230
starting iteration  5240
starting iteration  5250
starting iteration  5260
starting iteration  5270
starting iteration  5280
starting iteration  5290
starting iteration  5300
starting iteration  5310
starting iteration  5320
starting iteration  5330
starting iteration  5340
starting iteration  5350
starting iteration  5360
starting iteration  5370
starting iteration  5380
Finished epoch 11 / 28: cost 1.721801, train: 0.530000, val 0.478000, lr 4.70
7159e-05
starting iteration  5390
starting iteration  5400
starting iteration  5410
starting iteration  5420
starting iteration  5430
starting iteration  5440
starting iteration  5450
```

```
starting iteration  5460
starting iteration  5470
starting iteration  5480
starting iteration  5490
starting iteration  5500
starting iteration  5510
starting iteration  5520
starting iteration  5530
starting iteration  5540
starting iteration  5550
starting iteration  5560
starting iteration  5570
starting iteration  5580
starting iteration  5590
starting iteration  5600
starting iteration  5610
starting iteration  5620
starting iteration  5630
starting iteration  5640
starting iteration  5650
starting iteration  5660
starting iteration  5670
starting iteration  5680
starting iteration  5690
starting iteration  5700
starting iteration  5710
starting iteration  5720
starting iteration  5730
starting iteration  5740
starting iteration  5750
starting iteration  5760
starting iteration  5770
starting iteration  5780
starting iteration  5790
starting iteration  5800
starting iteration  5810
starting iteration  5820
starting iteration  5830
starting iteration  5840
starting iteration  5850
starting iteration  5860
starting iteration  5870
Finished epoch 12 / 28: cost 1.628940, train: 0.536000, val 0.496000, lr 4.23
6443e-05
starting iteration  5880
starting iteration  5890
starting iteration  5900
starting iteration  5910
starting iteration  5920
starting iteration  5930
starting iteration  5940
starting iteration  5950
starting iteration  5960
starting iteration  5970
starting iteration  5980
starting iteration  5990
starting iteration  6000
```

```
starting iteration  6010
starting iteration  6020
starting iteration  6030
starting iteration  6040
starting iteration  6050
starting iteration  6060
starting iteration  6070
starting iteration  6080
starting iteration  6090
starting iteration  6100
starting iteration  6110
starting iteration  6120
starting iteration  6130
starting iteration  6140
starting iteration  6150
starting iteration  6160
starting iteration  6170
starting iteration  6180
starting iteration  6190
starting iteration  6200
starting iteration  6210
starting iteration  6220
starting iteration  6230
starting iteration  6240
starting iteration  6250
starting iteration  6260
starting iteration  6270
starting iteration  6280
starting iteration  6290
starting iteration  6300
starting iteration  6310
starting iteration  6320
starting iteration  6330
starting iteration  6340
starting iteration  6350
starting iteration  6360
Finished epoch 13 / 28: cost 1.558282, train: 0.513000, val 0.511000, lr 3.81
2799e-05
starting iteration  6370
starting iteration  6380
starting iteration  6390
starting iteration  6400
starting iteration  6410
starting iteration  6420
starting iteration  6430
starting iteration  6440
starting iteration  6450
starting iteration  6460
starting iteration  6470
starting iteration  6480
starting iteration  6490
starting iteration  6500
starting iteration  6510
starting iteration  6520
starting iteration  6530
starting iteration  6540
starting iteration  6550
```

```
starting iteration  6560
starting iteration  6570
starting iteration  6580
starting iteration  6590
starting iteration  6600
starting iteration  6610
starting iteration  6620
starting iteration  6630
starting iteration  6640
starting iteration  6650
starting iteration  6660
starting iteration  6670
starting iteration  6680
starting iteration  6690
starting iteration  6700
starting iteration  6710
starting iteration  6720
starting iteration  6730
starting iteration  6740
starting iteration  6750
starting iteration  6760
starting iteration  6770
starting iteration  6780
starting iteration  6790
starting iteration  6800
starting iteration  6810
starting iteration  6820
starting iteration  6830
starting iteration  6840
starting iteration  6850
Finished epoch 14 / 28: cost 1.523632, train: 0.546000, val 0.502000, lr 3.43
1519e-05
starting iteration  6860
starting iteration  6870
starting iteration  6880
starting iteration  6890
starting iteration  6900
starting iteration  6910
starting iteration  6920
starting iteration  6930
starting iteration  6940
starting iteration  6950
starting iteration  6960
starting iteration  6970
starting iteration  6980
starting iteration  6990
starting iteration  7000
starting iteration  7010
starting iteration  7020
starting iteration  7030
starting iteration  7040
starting iteration  7050
starting iteration  7060
starting iteration  7070
starting iteration  7080
starting iteration  7090
starting iteration  7100
```

```
starting iteration  7110
starting iteration  7120
starting iteration  7130
starting iteration  7140
starting iteration  7150
starting iteration  7160
starting iteration  7170
starting iteration  7180
starting iteration  7190
starting iteration  7200
starting iteration  7210
starting iteration  7220
starting iteration  7230
starting iteration  7240
starting iteration  7250
starting iteration  7260
starting iteration  7270
starting iteration  7280
starting iteration  7290
starting iteration  7300
starting iteration  7310
starting iteration  7320
starting iteration  7330
starting iteration  7340
Finished epoch 15 / 28: cost 1.396022, train: 0.518000, val 0.489000, lr 3.08
8367e-05
starting iteration  7350
starting iteration  7360
starting iteration  7370
starting iteration  7380
starting iteration  7390
starting iteration  7400
starting iteration  7410
starting iteration  7420
starting iteration  7430
starting iteration  7440
starting iteration  7450
starting iteration  7460
starting iteration  7470
starting iteration  7480
starting iteration  7490
starting iteration  7500
starting iteration  7510
starting iteration  7520
starting iteration  7530
starting iteration  7540
starting iteration  7550
starting iteration  7560
starting iteration  7570
starting iteration  7580
starting iteration  7590
starting iteration  7600
starting iteration  7610
starting iteration  7620
starting iteration  7630
starting iteration  7640
starting iteration  7650
```

```
starting iteration  7660
starting iteration  7670
starting iteration  7680
starting iteration  7690
starting iteration  7700
starting iteration  7710
starting iteration  7720
starting iteration  7730
starting iteration  7740
starting iteration  7750
starting iteration  7760
starting iteration  7770
starting iteration  7780
starting iteration  7790
starting iteration  7800
starting iteration  7810
starting iteration  7820
starting iteration  7830
Finished epoch 16 / 28: cost 1.521139, train: 0.555000, val 0.509000, lr 2.77
9530e-05
starting iteration  7840
starting iteration  7850
starting iteration  7860
starting iteration  7870
starting iteration  7880
starting iteration  7890
starting iteration  7900
starting iteration  7910
starting iteration  7920
starting iteration  7930
starting iteration  7940
starting iteration  7950
starting iteration  7960
starting iteration  7970
starting iteration  7980
starting iteration  7990
starting iteration  8000
starting iteration  8010
starting iteration  8020
starting iteration  8030
starting iteration  8040
starting iteration  8050
starting iteration  8060
starting iteration  8070
starting iteration  8080
starting iteration  8090
starting iteration  8100
starting iteration  8110
starting iteration  8120
starting iteration  8130
starting iteration  8140
starting iteration  8150
starting iteration  8160
starting iteration  8170
starting iteration  8180
starting iteration  8190
starting iteration  8200
```

```
starting iteration  8210
starting iteration  8220
starting iteration  8230
starting iteration  8240
starting iteration  8250
starting iteration  8260
starting iteration  8270
starting iteration  8280
starting iteration  8290
starting iteration  8300
starting iteration  8310
starting iteration  8320
Finished epoch 17 / 28: cost 1.508517, train: 0.552000, val 0.526000, lr 2.50
1577e-05
starting iteration  8330
starting iteration  8340
starting iteration  8350
starting iteration  8360
starting iteration  8370
starting iteration  8380
starting iteration  8390
starting iteration  8400
starting iteration  8410
starting iteration  8420
starting iteration  8430
starting iteration  8440
starting iteration  8450
starting iteration  8460
starting iteration  8470
starting iteration  8480
starting iteration  8490
starting iteration  8500
starting iteration  8510
starting iteration  8520
starting iteration  8530
starting iteration  8540
starting iteration  8550
starting iteration  8560
starting iteration  8570
starting iteration  8580
starting iteration  8590
starting iteration  8600
starting iteration  8610
starting iteration  8620
starting iteration  8630
starting iteration  8640
starting iteration  8650
starting iteration  8660
starting iteration  8670
starting iteration  8680
starting iteration  8690
starting iteration  8700
starting iteration  8710
starting iteration  8720
starting iteration  8730
starting iteration  8740
starting iteration  8750
```

```
starting iteration  8760
starting iteration  8770
starting iteration  8780
starting iteration  8790
starting iteration  8800
starting iteration  8810
Finished epoch 18 / 28: cost 1.448959, train: 0.536000, val 0.497000, lr 2.25
1420e-05
starting iteration  8820
starting iteration  8830
starting iteration  8840
starting iteration  8850
starting iteration  8860
starting iteration  8870
starting iteration  8880
starting iteration  8890
starting iteration  8900
starting iteration  8910
starting iteration  8920
starting iteration  8930
starting iteration  8940
starting iteration  8950
starting iteration  8960
starting iteration  8970
starting iteration  8980
starting iteration  8990
starting iteration  9000
starting iteration  9010
starting iteration  9020
starting iteration  9030
starting iteration  9040
starting iteration  9050
starting iteration  9060
starting iteration  9070
starting iteration  9080
starting iteration  9090
starting iteration  9100
starting iteration  9110
starting iteration  9120
starting iteration  9130
starting iteration  9140
starting iteration  9150
starting iteration  9160
starting iteration  9170
starting iteration  9180
starting iteration  9190
starting iteration  9200
starting iteration  9210
starting iteration  9220
starting iteration  9230
starting iteration  9240
starting iteration  9250
starting iteration  9260
starting iteration  9270
starting iteration  9280
starting iteration  9290
starting iteration  9300
```

```
Finished epoch 19 / 28: cost 1.346652, train: 0.557000, val 0.507000, lr 2.02
6278e-05
starting iteration  9310
starting iteration  9320
starting iteration  9330
starting iteration  9340
starting iteration  9350
starting iteration  9360
starting iteration  9370
starting iteration  9380
starting iteration  9390
starting iteration  9400
starting iteration  9410
starting iteration  9420
starting iteration  9430
starting iteration  9440
starting iteration  9450
starting iteration  9460
starting iteration  9470
starting iteration  9480
starting iteration  9490
starting iteration  9500
starting iteration  9510
starting iteration  9520
starting iteration  9530
starting iteration  9540
starting iteration  9550
starting iteration  9560
starting iteration  9570
starting iteration  9580
starting iteration  9590
starting iteration  9600
starting iteration  9610
starting iteration  9620
starting iteration  9630
starting iteration  9640
starting iteration  9650
starting iteration  9660
starting iteration  9670
starting iteration  9680
starting iteration  9690
starting iteration  9700
starting iteration  9710
starting iteration  9720
starting iteration  9730
starting iteration  9740
starting iteration  9750
starting iteration  9760
starting iteration  9770
starting iteration  9780
starting iteration  9790
Finished epoch 20 / 28: cost 1.515841, train: 0.569000, val 0.506000, lr 1.82
3650e-05
starting iteration  9800
starting iteration  9810
starting iteration  9820
starting iteration  9830
```

```
starting iteration  9840
starting iteration  9850
starting iteration  9860
starting iteration  9870
starting iteration  9880
starting iteration  9890
starting iteration  9900
starting iteration  9910
starting iteration  9920
starting iteration  9930
starting iteration  9940
starting iteration  9950
starting iteration  9960
starting iteration  9970
starting iteration  9980
starting iteration  9990
starting iteration  10000
starting iteration  10010
starting iteration  10020
starting iteration  10030
starting iteration  10040
starting iteration  10050
starting iteration  10060
starting iteration  10070
starting iteration  10080
starting iteration  10090
starting iteration  10100
starting iteration  10110
starting iteration  10120
starting iteration  10130
starting iteration  10140
starting iteration  10150
starting iteration  10160
starting iteration  10170
starting iteration  10180
starting iteration  10190
starting iteration  10200
starting iteration  10210
starting iteration  10220
starting iteration  10230
starting iteration  10240
starting iteration  10250
starting iteration  10260
starting iteration  10270
starting iteration  10280
Finished epoch 21 / 28: cost 1.469721, train: 0.549000, val 0.520000, lr 1.64
1285e-05
starting iteration  10290
starting iteration  10300
starting iteration  10310
starting iteration  10320
starting iteration  10330
starting iteration  10340
starting iteration  10350
starting iteration  10360
starting iteration  10370
starting iteration  10380
```

```
starting iteration  10390
starting iteration  10400
starting iteration  10410
starting iteration  10420
starting iteration  10430
starting iteration  10440
starting iteration  10450
starting iteration  10460
starting iteration  10470
starting iteration  10480
starting iteration  10490
starting iteration  10500
starting iteration  10510
starting iteration  10520
starting iteration  10530
starting iteration  10540
starting iteration  10550
starting iteration  10560
starting iteration  10570
starting iteration  10580
starting iteration  10590
starting iteration  10600
starting iteration  10610
starting iteration  10620
starting iteration  10630
starting iteration  10640
starting iteration  10650
starting iteration  10660
starting iteration  10670
starting iteration  10680
starting iteration  10690
starting iteration  10700
starting iteration  10710
starting iteration  10720
starting iteration  10730
starting iteration  10740
starting iteration  10750
starting iteration  10760
starting iteration  10770
Finished epoch 22 / 28: cost 1.465887, train: 0.552000, val 0.516000, lr 1.47
7156e-05
starting iteration  10780
starting iteration  10790
starting iteration  10800
starting iteration  10810
starting iteration  10820
starting iteration  10830
starting iteration  10840
starting iteration  10850
starting iteration  10860
starting iteration  10870
starting iteration  10880
starting iteration  10890
starting iteration  10900
starting iteration  10910
starting iteration  10920
starting iteration  10930
```

```
starting iteration  10940
starting iteration  10950
starting iteration  10960
starting iteration  10970
starting iteration  10980
starting iteration  10990
starting iteration  11000
starting iteration  11010
starting iteration  11020
starting iteration  11030
starting iteration  11040
starting iteration  11050
starting iteration  11060
starting iteration  11070
starting iteration  11080
starting iteration  11090
starting iteration  11100
starting iteration  11110
starting iteration  11120
starting iteration  11130
starting iteration  11140
starting iteration  11150
starting iteration  11160
starting iteration  11170
starting iteration  11180
starting iteration  11190
starting iteration  11200
starting iteration  11210
starting iteration  11220
starting iteration  11230
starting iteration  11240
starting iteration  11250
starting iteration  11260
Finished epoch 23 / 28: cost 1.454009, train: 0.569000, val 0.512000, lr 1.32
9441e-05
starting iteration  11270
starting iteration  11280
starting iteration  11290
starting iteration  11300
starting iteration  11310
starting iteration  11320
starting iteration  11330
starting iteration  11340
starting iteration  11350
starting iteration  11360
starting iteration  11370
starting iteration  11380
starting iteration  11390
starting iteration  11400
starting iteration  11410
starting iteration  11420
starting iteration  11430
starting iteration  11440
starting iteration  11450
starting iteration  11460
starting iteration  11470
starting iteration  11480
```

```
starting iteration  11490
starting iteration  11500
starting iteration  11510
starting iteration  11520
starting iteration  11530
starting iteration  11540
starting iteration  11550
starting iteration  11560
starting iteration  11570
starting iteration  11580
starting iteration  11590
starting iteration  11600
starting iteration  11610
starting iteration  11620
starting iteration  11630
starting iteration  11640
starting iteration  11650
starting iteration  11660
starting iteration  11670
starting iteration  11680
starting iteration  11690
starting iteration  11700
starting iteration  11710
starting iteration  11720
starting iteration  11730
starting iteration  11740
starting iteration  11750
Finished epoch 24 / 28: cost 1.509655, train: 0.538000, val 0.520000, lr 1.19
6497e-05
starting iteration  11760
starting iteration  11770
starting iteration  11780
starting iteration  11790
starting iteration  11800
starting iteration  11810
starting iteration  11820
starting iteration  11830
starting iteration  11840
starting iteration  11850
starting iteration  11860
starting iteration  11870
starting iteration  11880
starting iteration  11890
starting iteration  11900
starting iteration  11910
starting iteration  11920
starting iteration  11930
starting iteration  11940
starting iteration  11950
starting iteration  11960
starting iteration  11970
starting iteration  11980
starting iteration  11990
starting iteration  12000
starting iteration  12010
starting iteration  12020
starting iteration  12030
```

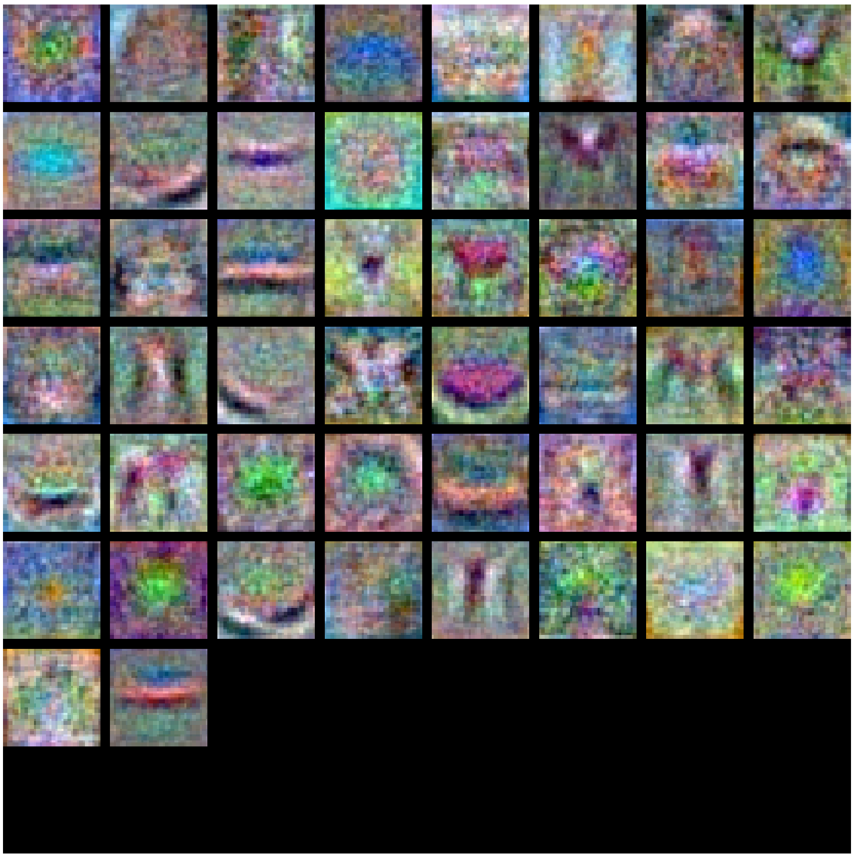
```
starting iteration  12040
starting iteration  12050
starting iteration  12060
starting iteration  12070
starting iteration  12080
starting iteration  12090
starting iteration  12100
starting iteration  12110
starting iteration  12120
starting iteration  12130
starting iteration  12140
starting iteration  12150
starting iteration  12160
starting iteration  12170
starting iteration  12180
starting iteration  12190
starting iteration  12200
starting iteration  12210
starting iteration  12220
starting iteration  12230
starting iteration  12240
Finished epoch 25 / 28: cost 1.430350, train: 0.575000, val 0.518000, lr 1.07
6847e-05
starting iteration  12250
starting iteration  12260
starting iteration  12270
starting iteration  12280
starting iteration  12290
starting iteration  12300
starting iteration  12310
starting iteration  12320
starting iteration  12330
starting iteration  12340
starting iteration  12350
starting iteration  12360
starting iteration  12370
starting iteration  12380
starting iteration  12390
starting iteration  12400
starting iteration  12410
starting iteration  12420
starting iteration  12430
starting iteration  12440
starting iteration  12450
starting iteration  12460
starting iteration  12470
starting iteration  12480
starting iteration  12490
starting iteration  12500
starting iteration  12510
starting iteration  12520
starting iteration  12530
starting iteration  12540
starting iteration  12550
starting iteration  12560
starting iteration  12570
starting iteration  12580
```

```
starting iteration  12590
starting iteration  12600
starting iteration  12610
starting iteration  12620
starting iteration  12630
starting iteration  12640
starting iteration  12650
starting iteration  12660
starting iteration  12670
starting iteration  12680
starting iteration  12690
starting iteration  12700
starting iteration  12710
starting iteration  12720
starting iteration  12730
Finished epoch 26 / 28: cost 1.519867, train: 0.580000, val 0.515000, lr 9.69
1623e-06
starting iteration  12740
starting iteration  12750
starting iteration  12760
starting iteration  12770
starting iteration  12780
starting iteration  12790
starting iteration  12800
starting iteration  12810
starting iteration  12820
starting iteration  12830
starting iteration  12840
starting iteration  12850
starting iteration  12860
starting iteration  12870
starting iteration  12880
starting iteration  12890
starting iteration  12900
starting iteration  12910
starting iteration  12920
starting iteration  12930
starting iteration  12940
starting iteration  12950
starting iteration  12960
starting iteration  12970
starting iteration  12980
starting iteration  12990
starting iteration  13000
starting iteration  13010
starting iteration  13020
starting iteration  13030
starting iteration  13040
starting iteration  13050
starting iteration  13060
starting iteration  13070
starting iteration  13080
starting iteration  13090
starting iteration  13100
starting iteration  13110
starting iteration  13120
starting iteration  13130
```

```
starting iteration  13140
starting iteration  13150
starting iteration  13160
starting iteration  13170
starting iteration  13180
starting iteration  13190
starting iteration  13200
starting iteration  13210
starting iteration  13220
Finished epoch 27 / 28: cost 1.428485, train: 0.555000, val 0.521000, lr 8.72
2461e-06
starting iteration  13230
starting iteration  13240
starting iteration  13250
starting iteration  13260
starting iteration  13270
starting iteration  13280
starting iteration  13290
starting iteration  13300
starting iteration  13310
starting iteration  13320
starting iteration  13330
starting iteration  13340
starting iteration  13350
starting iteration  13360
starting iteration  13370
starting iteration  13380
starting iteration  13390
starting iteration  13400
starting iteration  13410
starting iteration  13420
starting iteration  13430
starting iteration  13440
starting iteration  13450
starting iteration  13460
starting iteration  13470
starting iteration  13480
starting iteration  13490
starting iteration  13500
starting iteration  13510
starting iteration  13520
starting iteration  13530
starting iteration  13540
starting iteration  13550
starting iteration  13560
starting iteration  13570
starting iteration  13580
starting iteration  13590
starting iteration  13600
starting iteration  13610
starting iteration  13620
starting iteration  13630
starting iteration  13640
starting iteration  13650
starting iteration  13660
starting iteration  13670
starting iteration  13680
```

```
starting iteration  13690
starting iteration  13700
starting iteration  13710
Finished epoch 28 / 28: cost 1.380179, train: 0.551000, val 0.533000, lr 7.85
0214e-06
finished optimization. best validation accuracy: 0.533000
```

In [16]:
```
# visualize the weights
show_net_weights(best_model)
```



Run on the test set

When you are done experimenting, you should evaluate your final trained network on the test set.

We will give you extra bonus point for every 1% of accuracy above 56%.

In [17]:
```
scores_test = two_layer_net(X_test, best_model)
print 'Test accuracy: ', np.mean(np.argmax(scores_test, axis=1) == y_test)
```

```
Test accuracy:  0.52
```

In [ ]:
```
```

Modular neural nets

In the previous exercise, we computed the loss and gradient for a two-layer neural network in a single monolithic function. This isn't very difficult for a small two-layer network, but would be tedious and error-prone for larger networks. Ideally we want to build networks using a more modular design so that we can snap together different types of layers and loss functions in order to quickly experiment with different architectures.

In this exercise we will implement this approach, and develop a number of different layer types in isolation that can then be easily plugged together. For each layer we will implement `forward` and `backward` functions. The `forward` function will receive data, weights, and other parameters, and will return both an output and a `cache` object that stores data needed for the backward pass. The `backward` function will recieve upstream derivatives and the cache object, and will return gradients with respect to the data and all of the weights. This will allow us to write code that looks like this:

```python
def two_layer_net(X, W1, b1, W2, b2, reg):
    # Forward pass; compute scores
    s1, fc1_cache = affine_forward(X, W1, b1)
    a1, relu_cache = relu_forward(s1)
    scores, fc2_cache = affine_forward(a1, W2, b2)

    # Loss functions return data loss and gradients on scores
    data_loss, dscores = svm_loss(scores, y)

    # Compute backward pass
    da1, dW2, db2 = affine_backward(dscores, fc2_cache)
    ds1 = relu_backward(da1, relu_cache)
    dX, dW1, db1 = affine_backward(ds1, fc1_cache)

    # A real network would add regularization here

    # Return loss and gradients
    return loss, dW1, db1, dW2, db2
```

In [3]:
```
# As usual, a bit of setup

import numpy as np
import matplotlib.pyplot as plt
from cs231n.gradient_check import eval_numerical_gradient_array, eval_numerica
l_gradient
from cs231n.layers import *

%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))))
```

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

Affine layer: forward

Open the file `cs231n/layers.py` and implement the `affine_forward` function.

Once you are done we will test your can test your implementation by running the following:

In [5]:
```
# Test the affine_forward function

num_inputs = 2
input_shape = (4, 5, 6)
output_dim = 3

input_size = num_inputs * np.prod(input_shape)
weight_size = output_dim * np.prod(input_shape)

x = np.linspace(-0.1, 0.5, num=input_size).reshape(num_inputs, *input_shape)
w = np.linspace(-0.2, 0.3, num=weight_size).reshape(np.prod(input_shape), outp
ut_dim)
b = np.linspace(-0.3, 0.1, num=output_dim)

out, _ = affine_forward(x, w, b)
correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
                        [ 3.25553199,  3.5141327,   3.77273342]])

# Compare your output with ours. The error should be around 1e-9.
print 'Testing affine_forward function:'
print 'difference: ', rel_error(out, correct_out)
```

```
Testing affine_forward function:
difference:  9.76984772881e-10
```

Affine layer: backward

Now implement the `affine_backward` function. You can test your implementation using numeric gradient checking.

In [6]:
```
# Test the affine_backward function

x = np.random.randn(10, 2, 3)
w = np.random.randn(6, 5)
b = np.random.randn(5)
dout = np.random.randn(10, 5)

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

_, cache = affine_forward(x, w, b)
dx, dw, db = affine_backward(dout, cache)

# The error should be less than 1e-10
print 'Testing affine_backward function:'
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 affine_backward function:
dx error:  6.99811149861e-10
dw error:  9.14830163174e-11
db error:  2.16251754573e-11
```

ReLU layer: forward

Implement the `relu_forward` function and test your implementation by running the following:

In [7]:
```
# Test the relu_forward function

x = np.linspace(-0.5, 0.5, num=12).reshape(3, 4)

out, _ = relu_forward(x)
correct_out = np.array([[ 0.,          0.,          0.,          0.,        ],
                        [ 0.,          0.,          0.04545455,  0.13636364,],
                        [ 0.22727273,  0.31818182,  0.40909091,  0.5,
]])

# Compare your output with ours. The error should be around 1e-8
print 'Testing relu_forward function:'
print 'difference: ', rel_error(out, correct_out)
```

```
Testing relu_forward function:
difference:  4.99999979802e-08
```

ReLU layer: backward

Implement the `relu_backward` function and test your implementation using numeric gradient checking:

In [8]:
```
x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)

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

_, cache = relu_forward(x)
dx = relu_backward(dout, cache)

# The error should be around 1e-12
print 'Testing relu_backward function:'
print 'dx error: ', rel_error(dx_num, dx)
```

```
Testing relu_backward function:
dx error:  3.27564222471e-12
```

Loss layers: Softmax and SVM

You implemented these loss functions in the last assignment, so we'll give them to you for free here. It's still a good idea to test them to make sure they work correctly.

In [9]:
```
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)

dx_num = eval_numerical_gradient(lambda x: svm_loss(x, y)[0], x,
verbose=False)
loss, dx = svm_loss(x, y)

# Test svm_loss function. Loss should be around 9 and dx error should be 1e-9
print 'Testing svm_loss:'
print 'loss: ', loss
print 'dx error: ', rel_error(dx_num, dx)

dx_num = eval_numerical_gradient(lambda x: softmax_loss(x, y)[0], x, verbose=F
alse)
loss, dx = softmax_loss(x, y)

# Test softmax_loss function. Loss should be 2.3 and dx error should be 1e-8
print '\nTesting softmax_loss:'
print 'loss: ', loss
print 'dx error: ', rel_error(dx_num, dx)
```

```
Testing svm_loss:
loss:  8.99999185188
dx error:  8.18289447289e-10

Testing softmax_loss:
loss:  2.30258473372
dx error:  8.68969367873e-09
```

Listing 7.1: *neural_net.py*

```
import numpy as np
import matplotlib.pyplot as plt

def init_two_layer_model(input_size, hidden_size, output_size):
"""
Initialize the weights and biases for a two-layer fully connected neural
network. The net has an input dimension of D, a hidden layer dimension of H,
and performs classification over C classes. Weights are initialized to small
random values and biases are initialized to zero.

Inputs:
- input_size: The dimension D of the input data
- hidden_size: The number of neurons H in the hidden layer
- ouput_size: The number of classes C

Returns:
A dictionary mapping parameter names to arrays of parameter values. It has
the following keys:
- W1: First layer weights; has shape (D, H)
- b1: First layer biases; has shape (H,)
- W2: Second layer weights; has shape (H, C)
- b2: Second layer biases; has shape (C,)
"""
# initialize a model
model = {}
model['W1'] = 0.00001 * np.random.randn(input_size, hidden_size)
model['b1'] = np.zeros(hidden_size)
model['W2'] = 0.00001 * np.random.randn(hidden_size, output_size)
model['b2'] = np.zeros(output_size)
return model

def two_layer_net(X, model, y=None, reg=0.0):
"""
Compute the loss and gradients for a two layer fully connected neural network.
The net has an input dimension of D, a hidden layer dimension of H, and
performs classification over C classes. We use a softmax loss function and L2
regularization the the weight matrices. The two layer net should use a ReLU
nonlinearity after the first affine layer.

The two layer net has the following architecture:

input - fully connected layer - ReLU - fully connected layer - softmax

The outputs of the second fully-connected layer are the scores for each
class.

Inputs:
- X: Input data of shape (N, D). Each X[i] is a training sample.
- model: Dictionary mapping parameter names to arrays of parameter values.
It should contain the following:
- W1: First layer weights; has shape (D, H)
- b1: First layer biases; has shape (H,)
- W2: Second layer weights; has shape (H, C)
- b2: Second layer biases; has shape (C,)
- y: Vector of training labels. y[i] is the label for X[i], and each y[i] is
an integer in the range 0 <= y[i] < C. This parameter is optional; if it
is not passed then we only return scores, and if it is passed then we
instead return the loss and gradients.
- reg: Regularization strength.

Returns:
If y not is passed, return a matrix scores of shape (N, C) where scores[i, c]
is the score for class c on input X[i].

If y is not passed, instead return a tuple of:
- loss: Loss (data loss and regularization loss) for this batch of training
samples.
- grads: Dictionary mapping parameter names to gradients of those parameters
with respect to the loss function. This should have the same keys as model.
"""

```

```
# unpack variables from the model dictionary
W1,b1,W2,b2 = model['W1'], model['b1'], model['W2'], model['b2']
N, D = X.shape
# input - fully connected layer - ReLU - fully connected layer - softmax
#Activation function
reluF = lambda x: np.maximum(0, x)
#Compared to lecture notes we switch X and W1 because to multiple the inputs
#with the correct weights
h1 = reluF(np.dot(X, W1) + b1)
scores = np.dot(h1, W2) + b2
#############################################################################
#                              END OF YOUR CODE                             #
#############################################################################

# If the targets are not given then jump out, we're done
if y is None:
return scores

#############################################################################
# TODO: Finish the forward pass, and compute the loss. This should include  #
# both the data loss and L2 regularization for W1 and W2. Store the result  #
# in the variable loss, which should be a scalar. Use the Softmax           #
# classifier loss. So that your results match ours, multiply the           #
# regularization loss by 0.5                                                #
#############################################################################
# compute the loss
expScores = np.exp(scores)
rowSum = expScores.sum(axis=1, keepdims=True)
# Normalized scores
propScores = expScores / rowSum
logprob_correctLabel = -np.log(propScores[range(N),y])
softmax_loss = 1/float(N) * np.sum(logprob_correctLabel)
#regulization loss
reg_loss = 0.5 * reg * np.sum(W1*W1) + 0.5 * reg * np.sum(W2 * W2)
#Final loss
loss = softmax_loss + reg_loss
#############################################################################
#                              END OF YOUR CODE                             #
#############################################################################

# compute the gradients
grads = {}

#############################################################################
# TODO: Compute the backward pass, computing the derivatives of the weights #
# and biases. Store the results in the grads dictionary. For example,       #
# grads['W1'] should store the gradient on W1, and be a matrix of same size #
#############################################################################
# Firstly we calculate the gradient on the scores.
# The gradient from the loss function is simply -1.
# This is subtracted from the correct scores for each
# dscores are the probabilities for all classes as a row for each sample
dscores = propScores
#For each row(sample) in dscores 1 is subtracted from the correct element
#specified by y
dscores[range(N),y] -= 1
# We then divide all elements with N(number of samples)
dscores /= N

# The gradient for W2 is simply the output from the RELU activation function (h1)
# multiplyed with the dscores that contains the gradient on the scores.
# d/dw(w*x) = x which is our h1 then we get the input times dscores
grads['W2'] = np.dot(h1.T, dscores)
#bias is just the sum of the dscores
grads['b2'] = np.sum(dscores, axis=0)

# next backprop into hidden layer. This is the scores multiplied with the weights
# for second layer
dhidden = np.dot(dscores, W2.T)
# backprop the ReLU non-linearity.
#For elements < or equals 0  we set them equals to 0
# remember how Relu is just max, so it routes the gradients
dhidden[h1 <= 0] = 0
```

```
# same thing as second layer - d/dw(w*x) = x, so x times our gradient for dhidden
grads['W1'] = np.dot(X.T, dhidden)
grads['b1'] = np.sum(dhidden, axis=0)

# adding gradient for regulization
# d/dw(1/2*reg*W1*w1) = reg * W1
grads['W1'] += reg * W1
grads['W2'] += reg * W2
#############################################################################
#                              END OF YOUR CODE                             #
#############################################################################

return loss, grads
```

Listing 7.2: *classifier_trainer.py*

```
import numpy as np


class ClassifierTrainer(object):
""" The trainer class performs SGD with momentum on a cost function """
def __init__(self):
self.step_cache = {} # for storing velocities in momentum update

def train(self, X, y, X_val, y_val,
model, loss_function,
reg=0.0,
learning_rate=1e-2, momentum=0, learning_rate_decay=0.95,
update='momentum', sample_batches=True,
num_epochs=30, batch_size=100, acc_frequency=None,
verbose=False):
"""
Optimize the parameters of a model to minimize a loss function. We use
training data X and y to compute the loss and gradients, and periodically
check the accuracy on the validation set.

Inputs:
- X: Array of training data; each X[i] is a training sample.
- y: Vector of training labels; y[i] gives the label for X[i].
- X_val: Array of validation data
- y_val: Vector of validation labels
- model: Dictionary that maps parameter names to parameter values. Each
parameter value is a numpy array.
- loss_function: A function that can be called in the following ways:
scores = loss_function(X, model, reg=reg)
loss, grads = loss_function(X, model, y, reg=reg)
- reg: Regularization strength. This will be passed to the loss function.
- learning_rate: Initial learning rate to use.
- momentum: Parameter to use for momentum updates.
- learning_rate_decay: The learning rate is multiplied by this after each
epoch.
- update: The update rule to use. One of 'sgd', 'momentum', or 'rmsprop'.
- sample_batches: If True, use a minibatch of data for each parameter update
(stochastic gradient descent); if False, use the entire training set for
each parameter update (gradient descent).
- num_epochs: The number of epochs to take over the training data.
- batch_size: The number of training samples to use at each iteration.
- acc_frequency: If set to an integer, we compute the training and
validation set error after every acc_frequency iterations.
- verbose: If True, print status after each epoch.

Returns a tuple of:
- best_model: The model that got the highest validation accuracy during
training.
- loss_history: List containing the value of the loss function at each
iteration.
- train_acc_history: List storing the training set accuracy at each epoch.
- val_acc_history: List storing the validation set accuracy at each epoch.
"""

```

```
N = X.shape[0]

if sample_batches:
iterations_per_epoch = N / batch_size # using SGD
else:
iterations_per_epoch = 1 # using GD
num_iters = num_epochs * iterations_per_epoch
epoch = 0
best_val_acc = 0.0
best_model = {}
loss_history = []
train_acc_history = []
val_acc_history = []
for it in xrange(num_iters):
if it % 10 == 0:  print 'starting iteration ', it

# get batch of data
if sample_batches:
batch_mask = np.random.choice(N, batch_size)
X_batch = X[batch_mask]
y_batch = y[batch_mask]
else:
# no SGD used, full gradient descent
X_batch = X
y_batch = y

# evaluate cost and gradient
cost, grads = loss_function(X_batch, model, y_batch, reg)
loss_history.append(cost)

# perform a parameter update
for p in model:
# compute the parameter step
if update == 'sgd':
dx = -learning_rate * grads[p]
elif update == 'momentum':
if not p in self.step_cache:
self.step_cache[p] = np.zeros(grads[p].shape)
dx = np.zeros_like(grads[p]) # you can remove this after
#############################################################################
# TODO: implement the momentum update formula and store the step            #
# update into variable dx. You should use the variable                      #
# step_cache[p] and the momentum strength is stored in momentum.            #
# Don't forget to also update the step_cache[p].                            #
#############################################################################
# Momentum update
# Momentum strength is a coefficient explains to friction,
# moment strength damps velocity and reduces the kinetic energy
#it ensures that the particle stops at the bottom
# step_cache is our velocity at time t-1
# From that we subtract the learning rate multiplied the gradients,
# because we go in the opposite direction of the gradient
dx = momentum * self.step_cache[p] - learning_rate * grads[p] # integrate velocity
self.step_cache[p] = dx

#############################################################################
#                             END OF YOUR CODE                              #
#############################################################################
elif update == 'rmsprop':
decay_rate = 0.99 # you could also make this an option
if not p in self.step_cache:
self.step_cache[p] = np.zeros(grads[p].shape)
#############################################################################
# TODO: implement the RMSProp update and store the parameter update #
# dx. Don't forget to also update step_cache[p]. Use smoothing 1e-8 #
#############################################################################
#eps = 1e-8
eps = 1e5
self.step_cache[p] = decay_rate * self.step_cache[p] + (1 - decay_rate) * grads[p]**2
dx = - learning_rate * grads[p] / (np.sqrt(self.step_cache[p]) + eps)


#############################################################################
```

```
#                              END OF YOUR CODE                              #
##############################################################################
else:
raise ValueError('Unrecognized update type "%s"' % update)

# update the parameters
model[p] += dx

# every epoch perform an evaluation on the validation set
first_it = (it == 0)
epoch_end = (it + 1) % iterations_per_epoch == 0
acc_check = (acc_frequency is not None and it % acc_frequency == 0)
if first_it or epoch_end or acc_check:
if it > 0 and epoch_end:
# decay the learning rate
learning_rate *= learning_rate_decay
epoch += 1

# evaluate train accuracy
if N > 1000:
train_mask = np.random.choice(N, 1000)
X_train_subset = X[train_mask]
y_train_subset = y[train_mask]
else:
X_train_subset = X
y_train_subset = y
scores_train = loss_function(X_train_subset, model)
y_pred_train = np.argmax(scores_train, axis=1)
train_acc = np.mean(y_pred_train == y_train_subset)
train_acc_history.append(train_acc)

# evaluate val accuracy
scores_val = loss_function(X_val, model)
y_pred_val = np.argmax(scores_val, axis=1)
val_acc = np.mean(y_pred_val ==  y_val)
val_acc_history.append(val_acc)

# keep track of the best model based on validation accuracy
if val_acc > best_val_acc:
# make a copy of the model
best_val_acc = val_acc
best_model = {}
for p in model:
best_model[p] = model[p].copy()

# print progress if needed
if verbose:
print ('Finished epoch %d / %d: cost %f, train: %f, val %f, lr %e'
% (epoch, num_epochs, cost, train_acc, val_acc, learning_rate))

if verbose:
print 'finished optimization. best validation accuracy: %f' % (best_val_acc, )
# return the best model and the training history statistics
return best_model, loss_history, train_acc_history, val_acc_history
```

Listing 7.3: *layers.py*

```
import numpy as np

def affine_forward(x, w, b):
"""
Computes the forward pass for an affine (fully-connected) layer.

The input x has shape (N, d_1, ..., d_k) where x[i] is the ith input.
We multiply this against a weight matrix of shape (D, M) where
D = \prod_i d_i

Inputs:
x - Input data, of shape (N, d_1, ..., d_k)
w - Weights, of shape (D, M)
```

```
b - Biases, of shape (M,)

Returns a tuple of:
- out: output, of shape (N, M)
- cache: (x, w, b)
"""
out = None
###########################################################################
# TODO: Implement the affine forward pass. Store the result in out. You   #
# will need to reshape the input into rows.                               #
###########################################################################
# First we reshape X to mulitply it with the incoming weights
# We get column and row size and then reshape
row_size = x.shape[0]
col_size = np.prod(x.shape[1:])
x_reshape = x.reshape(row_size, col_size)
# To execute the forward pass we simply need to multiply the inputs with the weights
out = np.dot(x_reshape, w) + b
###########################################################################
#                             END OF YOUR CODE                            #
###########################################################################
cache = (x, w, b)
return out, cache


def affine_backward(dout, cache):
"""
Computes the backward pass for an affine layer.

Inputs:
- dout: Upstream derivative, of shape (N, M)
- cache: Tuple of:
- x: Input data, of shape (N, d_1, ... d_k)
- w: Weights, of shape (D, M)

Returns a tuple of:
- dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
- dw: Gradient with respect to w, of shape (D, M)
- db: Gradient with respect to b, of shape (M,)
"""
x, w, b = cache
dx, dw, db = None, None, None
###########################################################################
# TODO: Implement the affine backward pass.                               #
###########################################################################
# First we reshape X to mulitply it with the incoming weights
# We get column and row size and then reshape
row_size = x.shape[0]
col_size = np.prod(x.shape[1:])
x_reshape = x.reshape(row_size, col_size)

# After reshaping we calculate the backward pass

# Gradient with respect to x
# Gradient of x*w with respect to x is simply w,
# We multiply that with the upstream gradient
dx2 = np.dot(dout, w.T)
dx = np.reshape(dx2, x.shape)

# Gradient with respect to weights
# Gradient of x*w with respect to w is simply x,
# We multiply that with the upstream gradient
dw = np.dot(x_reshape.T, dout)

# Gradient with respect to bias
# Biases are added so the gradient is simply 1,
# We multiply that with the upstream gradient.
db = np.dot(dout.T, np.ones(row_size))

###########################################################################
#                             END OF YOUR CODE                            #
###########################################################################
return dx, dw, db
```

```python


def relu_forward(x):
"""
Computes the forward pass for a layer of rectified linear units (ReLUs).

Input:
- x: Inputs, of any shape

Returns a tuple of:
- out: Output, of the same shape as x
- cache: x
"""
out = None
###########################################################################
# TODO: Implement the ReLU forward pass.                                  #
###########################################################################
reluF = lambda x: np.maximum(0, x)
out = reluF(x)
###########################################################################
#                             END OF YOUR CODE                            #
###########################################################################
cache = x
return out, cache


def relu_backward(dout, cache):
"""
Computes the backward pass for a layer of rectified linear units (ReLUs).

Input:
- dout: Upstream derivatives, of any shape
- cache: Input x, of same shape as dout

Returns:
- dx: Gradient with respect to x
"""
dx, x = None, cache
###########################################################################
# TODO: Implement the ReLU backward pass.                                 #
###########################################################################
#reluf function
reluF = lambda x: np.maximum(0, x)

out = reluF(x)
# Reluf is a max gate and so we can think of it as a router of gradients
# the max value is the one that the gradient is routated to
# we simpy set the out value to 1 if the out value is bigger than 0
out[out > 0] = 1

# Multiply out  with upstream gradient, to "route" the gradient
dx = out * dout
###########################################################################
#                             END OF YOUR CODE                            #
###########################################################################
return dx

def dropout_forward(x, dropout_param):
"""
Performs the forward pass for (inverted) dropout.

Inputs:
- x: Input data, of any shape
- dropout_param: A dictionary with the following keys:
- p: Dropout parameter. We keep each neuron output with probability p.
- mode: 'test' or 'train'. If the mode is train, then perform dropout;
if the mode is test, then just return the input.
- seed: Seed for the random number generator. Passing seed makes this
function deterministic, which is needed for gradient checking but not in
real networks.

Outputs:
- out: Array of the same shape as x.
```

```
- cache: A tuple (dropout_param, mask). In training mode, mask is the dropout
mask that was used to multiply the input; in test mode, mask is None.
"""
p, mode = dropout_param['p'], dropout_param['mode']
if 'seed' in dropout_param:
np.random.seed(dropout_param['seed'])

mask = None
out = None

if mode == 'train':
###########################################################################
# TODO: Implement the training phase forward pass for inverted dropout.   #
# Store the dropout mask in the mask variable.                            #
###########################################################################
pass
###########################################################################
#                             END OF YOUR CODE                            #
###########################################################################
elif mode == 'test':
###########################################################################
# TODO: Implement the test phase forward pass for inverted dropout.       #
###########################################################################
pass
###########################################################################
#                             END OF YOUR CODE                            #
###########################################################################

cache = (dropout_param, mask)
out = out.astype(x.dtype, copy=False)

return out, cache


def dropout_backward(dout, cache):
"""
Perform the backward pass for (inverted) dropout.

Inputs:
- dout: Upstream derivatives, of any shape
- cache: (dropout_param, mask) from dropout_forward.
"""
dropout_param, mask = cache
mode = dropout_param['mode']
if mode == 'train':
###########################################################################
# TODO: Implement the training phase forward pass for inverted dropout.   #
# Store the dropout mask in the mask variable.                            #
###########################################################################
pass
###########################################################################
#                             END OF YOUR CODE                            #
###########################################################################
elif mode == 'test':
dx = dout
return dx


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

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

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

```
- 'pad': The number of pixels that will be used to zero-pad the input.

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


def conv_backward_naive(dout, cache):
"""
A naive implementation of the backward pass for a convolutional layer.

Inputs:
- dout: Upstream derivatives.
- cache: A tuple of (x, w, b, conv_param) as in conv_forward_naive

Returns a tuple of:
- dx: Gradient with respect to x
- dw: Gradient with respect to w
- db: Gradient with respect to b
"""
dx, dw, db = None, None, None
###########################################################################
# TODO: Implement the convolutional backward pass.                        #
###########################################################################
pass
###########################################################################
#                             END OF YOUR CODE                            #
###########################################################################
return dx, dw, db


def max_pool_forward_naive(x, pool_param):
"""
A naive implementation of the forward pass for a max pooling layer.

Inputs:
- x: Input data, of shape (N, C, H, W)
- pool_param: dictionary with the following keys:
- 'pool_height': The height of each pooling region
- 'pool_width': The width of each pooling region
- 'stride': The distance between adjacent pooling regions

Returns a tuple of:
- out: Output data
- cache: (x, pool_param)
"""
out = None
###########################################################################
# TODO: Implement the max pooling forward pass                            #
###########################################################################
pass
###########################################################################
#                             END OF YOUR CODE                            #
###########################################################################
cache = (x, pool_param)
return out, cache


def max_pool_backward_naive(dout, cache):
```

```
"""
A naive implementation of the backward pass for a max pooling layer.

Inputs:
- dout: Upstream derivatives
- cache: A tuple of (x, pool_param) as in the forward pass.

Returns:
- dx: Gradient with respect to x
"""
dx = None
###########################################################################
# TODO: Implement the max pooling backward pass                           #
###########################################################################
pass
###########################################################################
#                             END OF YOUR CODE                            #
###########################################################################
return dx


def svm_loss(x, y):
"""
Computes the loss and gradient using for multiclass SVM classification.

Inputs:
- x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
for the ith input.
- y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
0 <= y[i] < C

Returns a tuple of:
- loss: Scalar giving the loss
- dx: Gradient of the loss with respect to x
"""
N = x.shape[0]
correct_class_scores = x[np.arange(N), y]
margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
margins[np.arange(N), y] = 0
loss = np.sum(margins) / N
num_pos = np.sum(margins > 0, axis=1)
dx = np.zeros_like(x)
dx[margins > 0] = 1
dx[np.arange(N), y] -= num_pos
dx /= N
return loss, dx


def softmax_loss(x, y):
"""
Computes the loss and gradient for softmax classification.

Inputs:
- x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
for the ith input.
- y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
0 <= y[i] < C

Returns a tuple of:
- loss: Scalar giving the loss
- dx: Gradient of the loss with respect to x
"""
probs = np.exp(x - np.max(x, axis=1, keepdims=True))
probs /= np.sum(probs, axis=1, keepdims=True)
N = x.shape[0]
loss = -np.sum(np.log(probs[np.arange(N), y])) / N
dx = probs.copy()
dx[np.arange(N), y] -= 1
dx /= N
return loss, dx
```

8 Exercise 2

Modular neural nets

In the previous exercise, we started to build modules/general layers for implementing large neural networks. In this exercise, we will expand on this by implementing a convolutional layer, max pooling layer and a dropout layer. For each layer we will implement `forward` and `backward` functions. The `forward` function will receive data, weights, and other parameters, and will return both an output and a `cache` object that stores data needed for the backward pass. The `backward` function will recieve upstream derivatives and the cache object, and will return gradients with respect to the data and all of the weights. This will allow us to write code that looks like this:

```
def two_layer_net(X, W1, b1, W2, b2, reg):
    # Forward pass; compute scores
    s1, fc1_cache = affine_forward(X, W1, b1)
    a1, relu_cache = relu_forward(s1)
    scores, fc2_cache = affine_forward(a1, W2, b2)

    # Loss functions return data loss and gradients on scores
    data_loss, dscores = svm_loss(scores, y)

    # Compute backward pass
    da1, dW2, db2 = affine_backward(dscores, fc2_cache)
    ds1 = relu_backward(da1, relu_cache)
    dX, dW1, db1 = affine_backward(ds1, fc1_cache)

    # A real network would add regularization here

    # Return loss and gradients
    return loss, dW1, db1, dW2, db2
```

In [1]:
```
# As usual, a bit of setup

import numpy as np
import matplotlib.pyplot as plt
from cs231n.gradient_check import eval_numerical_gradient_array, eval_numerica
l_gradient
from cs231n.layers import *

%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))))
```

Dropout layer: forward

Open the file `cs231n/layers.py` and implement the `dropout_forward` function. You should implement **inverted dropout** rather than regular dropout. We can check the forward pass by looking at the statistics of the outputs in train and test modes.

In [2]:
```
# Check the dropout forward pass

x = np.random.randn(100, 100)
dropout_param_train = {'p': 0.25, 'mode': 'train'}
dropout_param_test = {'p': 0.25, 'mode': 'test'}

out_train, _ = dropout_forward(x, dropout_param_train)
out_test, _ = dropout_forward(x, dropout_param_test)

# Test dropout training mode; about 25% of the elements should be nonzero
print np.mean(out_train != 0) # expected to be ~0.25

# Test dropout test mode; all of the elements should be nonzero
print np.mean(out_test != 0) # expected to be = 1
```

```
0.2511
1.0
```

Dropout layer: backward

Open the file `cs231n/layers.py` and implement the `dropout_backward` function. We can check the backward pass using numerical gradient checking.

In [3]:
```
from cs231n.gradient_check import eval_numerical_gradient_array

# Check the dropout backward pass

x = np.random.randn(5, 4)
dout = np.random.randn(*x.shape)
dropout_param = {'p': 0.8, 'mode': 'train', 'seed': 123}

dx_num = eval_numerical_gradient_array(lambda x: dropout_forward(x, dropout_pa
ram)[0], x, dout)

_, cache = dropout_forward(x, dropout_param)
dx = dropout_backward(dout, cache)

# The error should be around 1e-12
print 'Testing dropout_backward function:'
print 'dx error: ', rel_error(dx_num, dx)
```

```
Testing dropout_backward function:
dx error:  3.38580207168e-12
```

Convolution layer: forward naive

We are now ready to implement the forward pass for a convolutional layer. Implement the function `conv_forward_naive` in the file `cs231n/layers.py`.

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 [4]:
```
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 1e-8
print 'Testing conv_forward_naive'
print 'difference: ', rel_error(out, correct_out)
```

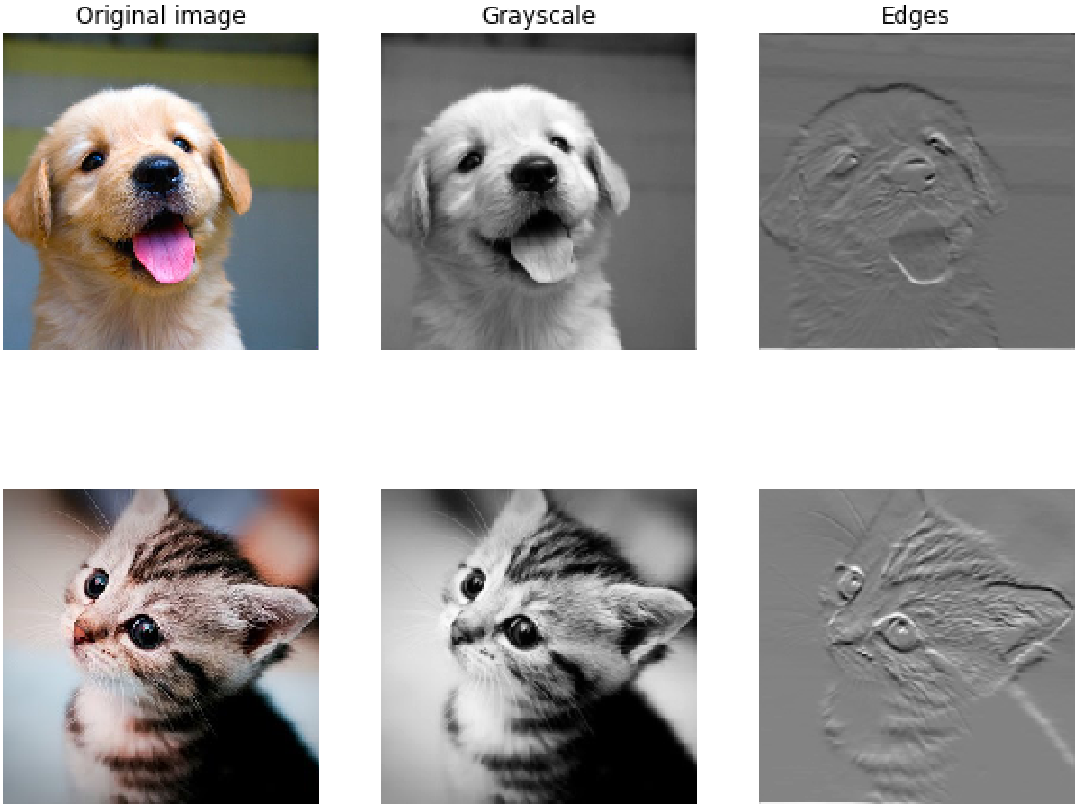
```
Testing conv_forward_naive
difference:  2.21214764175e-08
```

Aside: Image processing via convolutions

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

In [5]:
```
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()
```



Convolution layer: backward naive

Next you need to implement the function `conv_backward_naive` in the file `cs231n/layers.py`. As usual, we will check your implementation with numeric gradient checking.

In [6]:
```
x = np.random.randn(4, 3, 5, 5)
w = np.random.randn(2, 3, 3, 3)
b = np.random.randn(2,)
dout = np.random.randn(4, 2, 5, 5)
conv_param = {'stride': 1, 'pad': 1}

dx_num = eval_numerical_gradient_array(lambda x: conv_forward_naive(x, w, b, conv_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w, b, conv_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w, b, conv_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 1e-9'
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:  1.42918515583e-09
dw error:  3.24894092112e-10
db error:  4.93456965076e-12
```

Max pooling layer: forward naive

The last layer we need for a basic convolutional neural network is the max pooling layer. First implement the forward pass in the function `max_pool_forward_naive` in the file `cs231n/layers.py`.

In [7]:
```
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 around 1e-8.
print 'Testing max_pool_forward_naive function:'
print 'difference: ', rel_error(out, correct_out)
```

```
Testing max_pool_forward_naive function:
difference:  4.16666651573e-08
```

Max pooling layer: backward naive

Implement the backward pass for a max pooling layer in the function `max_pool_backward_naive` in the file `cs231n/layers.py`. As always we check the correctness of the backward pass using numerical gradient checking.

In [8]:
```
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 around 1e-12
print 'Testing max_pool_backward_naive function:'
print 'dx error: ', rel_error(dx, dx_num)
```

```
Testing max_pool_backward_naive function:
dx error:  3.27563758008e-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 [9]:
```
from cs231n.fast_layers import conv_forward_fast, conv_backward_fast
from time import time

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: 8.386237s
Fast: 0.027078s
Speedup: 309.707659x
Difference:  5.76441667407e-11

Testing conv_backward_fast:
Naive: 9.075142s
Fast: 0.023224s
Speedup: 390.763728x
dx difference:  1.37522088985e-11
dw difference:  4.97269198917e-13
db difference:  8.05806780804e-14
```

In [10]:
```
from cs231n.fast_layers import max_pool_forward_fast, max_pool_backward_fast

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 'speedup: %fx' % ((t1 - t0) / (t2 - t1))
print 'dx difference: ', rel_error(dx_naive, dx_fast)
```

```
Testing pool_forward_fast:
Naive: 0.218179s
fast: 0.004214s
speedup: 51.774201x
difference:  0.0

Testing pool_backward_fast:
Naive: 1.068016s
speedup: 40.932619x
dx difference:  0.0
```

Sandwich layers

There are a couple common layer "sandwiches" that frequently appear in ConvNets. For example convolutional layers are frequently followed by ReLU and pooling, and affine layers are frequently followed by ReLU. To make it more convenient to use these common patterns, we have defined several convenience layers in the file `cs231n/layer_utils.py`. Lets grad-check them to make sure that they work correctly:

In [11]:
```
from cs231n.layer_utils import conv_relu_pool_forward, conv_relu_pool_backward

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)

print 'Testing conv_relu_pool_forward:'
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_forward:
dx error:  7.39177599082e-09
dw error:  5.02782991993e-10
db error:  3.50315221012e-11
```

In [12]:
```
from cs231n.layer_utils import conv_relu_forward, conv_relu_backward

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)

print 'Testing conv_relu_forward:'
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_forward:
dx error:  2.26033376732e-09
dw error:  2.02817906612e-10
db error:  2.39239728675e-11
```

In [13]:
```
from cs231n.layer_utils import affine_relu_forward, affine_relu_backward

x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

out, cache = affine_relu_forward(x, w, b)
dx, dw, db = affine_relu_backward(dout, cache)

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

print 'Testing affine_relu_forward:'
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 affine_relu_forward:
dx error:  4.21086180119e-10
dw error:  3.49094679052e-10
db error:  1.70177182826e-11
```

In [ ]: