Softmax exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the [assignments page (https://compsci697l.github.io/assignments.html)](https://compsci697l.github.io/assignments.html) on the course website.

This exercise is analogous to the SVM exercise. You will:

- implement a fully-vectorized **loss function** for the Softmax classifier
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** with numerical gradient
- use a validation set to **tune the learning rate and regularization** strength
- **optimize** the loss function with **SGD**
- **visualize** the final learned weights

In [1]:

```
import random
import numpy as np
from asgn1.data_utils import load_CIFAR10
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 extenrnal modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

In [2]:

```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, num_de
  """
  Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
  it for the linear 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 = '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]
  mask = np.random.choice(num_training, num_dev, replace=False)
  X_dev = X_train[mask]
  y_dev = y_train[mask]

  # Preprocessing: reshape the image data into rows
  X_train = np.reshape(X_train, (X_train.shape[0], -1))
  X_val = np.reshape(X_val, (X_val.shape[0], -1))
  X_test = np.reshape(X_test, (X_test.shape[0], -1))
  X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

  # 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
  X_dev -= mean_image

  # add bias dimension and transform into columns
  X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
  X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
  X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
  X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

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


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev = 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
print 'dev data shape: ', X_dev.shape
print 'dev labels shape: ', y_dev.shape
```

```
Train data shape:  (49000, 3073)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3073)
```

```
Validation data shape:  (1000, 3073)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3073)
Test labels shape:  (1000,)
dev data shape:  (500, 3073)
dev labels shape:  (500,)
```

Softmax Classifier

Your code for this section will all be written inside **asgn1/classifiers/softmax.py**.

In [3]:

```python
# First implement the naive softmax loss function with nested loops.
# Open the file asgn1/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from asgn1.classifiers.softmax import softmax_loss_naive
import time

# Generate a random softmax weight matrix and use it to compute the loss.
W = np.random.randn(3073, 10) * 0.0001
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# As a rough sanity check, our loss should be something close to -log(0.1).
print 'loss: %f' % loss
print 'sanity check: %f' % (-np.log(0.1))
```

```
loss: 2.370486
sanity check: 2.302585
```

Inline Question 1:

Why do we expect our loss to be close to -log(0.1)? Explain briefly.**

Your answer: *Fill this in* When we run softmax with weights initialized randomly, the probability of getting the correct class is 1/10. Hence loss = -log(1/10) or -log(0.1)

In [4]:

```
# Complete the implementation of softmax_loss_naive and implement a (naive)
# version of the gradient that uses nested loops.
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# As we did for the SVM, use numeric gradient checking as a debugging tool.
# The numeric gradient should be close to the analytic gradient.
from asgn1.gradient_check import grad_check_sparse
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)

# similar to SVM case, do another gradient check with regularization
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 1e2)
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 1e2)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)
```

```
numerical: -1.236110 analytic: -1.236110, relative error: 1.090696e-08
numerical: 0.668721 analytic: 0.668721, relative error: 5.108578e-09
numerical: -0.472237 analytic: -0.472237, relative error: 3.137702e-08
numerical: 2.034204 analytic: 2.034204, relative error: 3.526026e-08
numerical: 2.181732 analytic: 2.181732, relative error: 9.129213e-09
numerical: -1.825013 analytic: -1.825013, relative error: 2.781728e-08
numerical: -0.440088 analytic: -0.440088, relative error: 1.912808e-08
numerical: -1.722876 analytic: -1.722876, relative error: 6.382400e-09
numerical: 0.414171 analytic: 0.414171, relative error: 4.362239e-09
numerical: -2.568082 analytic: -2.568082, relative error: 5.953708e-09
numerical: 3.576720 analytic: 3.576720, relative error: 8.966118e-09
numerical: 1.354405 analytic: 1.354405, relative error: 5.251024e-08
numerical: -0.533807 analytic: -0.533807, relative error: 1.044905e-07
numerical: -1.030340 analytic: -1.030340, relative error: 4.253525e-08
numerical: 0.835245 analytic: 0.835245, relative error: 8.421429e-09
numerical: 3.123163 analytic: 3.123162, relative error: 9.817540e-09
numerical: -1.827525 analytic: -1.827525, relative error: 3.474506e-08
numerical: 1.229852 analytic: 1.229852, relative error: 4.513306e-08
numerical: 2.577349 analytic: 2.577349, relative error: 1.298665e-08
numerical: -1.667184 analytic: -1.667184, relative error: 4.325253e-08
```

In [6]:

```
# Now that we have a naive implementation of the softmax loss function and its grad
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version shou
# much faster.
tic = time.time()
loss_naive, grad_naive = softmax_loss_naive(W, X_dev, y_dev, 0.00001)
toc = time.time()
print 'naive loss: %e computed in %fs' % (loss_naive, toc - tic)

from asgn1.classifiers.softmax import softmax_loss_vectorized
tic = time.time()
loss_vectorized, grad_vectorized = softmax_loss_vectorized(W, X_dev, y_dev, 0.00001
toc = time.time()
print 'vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic)

# As we did for the SVM, we use the Frobenius norm to compare the two versions
# of the gradient.
grad_difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print 'Loss difference: %f' % np.abs(loss_naive - loss_vectorized)
print 'Gradient difference: %f' % grad_difference
```

```
naive loss: 2.370486e+00 computed in 0.221337s
vectorized loss: 2.370486e+00 computed in 0.013622s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [7]:

```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of over 0.35 on the validation set.
from asgn1.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None
learning_rates = [3e-7,1e-7]
regularization_strengths = [5e5, 5e6]

for i in learning_rates:
    for j in regularization_strengths:
        softmax=Softmax()
        loss_hist = softmax.train(X_train, y_train, learning_rate=i, reg=j,
                      num_iters=1500, verbose=True)
        y_train_pred = softmax.predict(X_train)
        y_val_pred = softmax.predict(X_val)
        results[i,j]=[np.mean(y_train == y_train_pred),np.mean(y_val == y_val_pred)
        print np.mean(y_val == y_val_pred)
        if (np.mean(y_val == y_val_pred))>best_val:
            best_val=np.mean(y_val == y_val_pred)
            best_softmax=softmax

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print 'lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy)

print 'best validation accuracy achieved during cross-validation: %f' % best_val
```

```
iteration 0 / 1500: loss 44.729998
iteration 100 / 1500: loss 35.697032
iteration 200 / 1500: loss 30.535702
iteration 300 / 1500: loss 26.538868
iteration 400 / 1500: loss 22.919176
iteration 500 / 1500: loss 19.819916
iteration 600 / 1500: loss 17.319728
iteration 700 / 1500: loss 15.042559
iteration 800 / 1500: loss 13.234810
iteration 900 / 1500: loss 11.585390
iteration 1000 / 1500: loss 10.096006
iteration 1100 / 1500: loss 9.051851
iteration 1200 / 1500: loss 7.989155
iteration 1300 / 1500: loss 7.059098
iteration 1400 / 1500: loss 6.414644
0.365
iteration 0 / 1500: loss 392.313729
iteration 100 / 1500: loss 87.694722
iteration 200 / 1500: loss 20.948262
iteration 300 / 1500: loss 6.194258
iteration 400 / 1500: loss 2.920511
iteration 500 / 1500: loss 2.236900
iteration 600 / 1500: loss 2.066292
iteration 700 / 1500: loss 1.957672
iteration 800 / 1500: loss 2.004194
iteration 900 / 1500: loss 2.056716
iteration 1000 / 1500: loss 2.032526
```

```
iteration 1100 / 1500: loss 2.032035
iteration 1200 / 1500: loss 1.984740
iteration 1300 / 1500: loss 2.004033
iteration 1400 / 1500: loss 2.042767
0.378
iteration 0 / 1500: loss 43.380395
iteration 100 / 1500: loss 40.359616
iteration 200 / 1500: loss 37.605210
iteration 300 / 1500: loss 35.717250
iteration 400 / 1500: loss 34.460364
iteration 500 / 1500: loss 32.460937
iteration 600 / 1500: loss 31.050138
iteration 700 / 1500: loss 29.308560
iteration 800 / 1500: loss 27.901119
iteration 900 / 1500: loss 26.419422
iteration 1000 / 1500: loss 25.183309
iteration 1100 / 1500: loss 24.141686
iteration 1200 / 1500: loss 22.807778
iteration 1300 / 1500: loss 22.098529
iteration 1400 / 1500: loss 20.957333
0.267
iteration 0 / 1500: loss 385.071073
iteration 100 / 1500: loss 233.121373
iteration 200 / 1500: loss 141.707576
iteration 300 / 1500: loss 86.548437
iteration 400 / 1500: loss 53.141677
iteration 500 / 1500: loss 32.897744
iteration 600 / 1500: loss 20.748783
iteration 700 / 1500: loss 13.382361
iteration 800 / 1500: loss 8.897557
iteration 900 / 1500: loss 6.193017
iteration 1000 / 1500: loss 4.572103
iteration 1100 / 1500: loss 3.498468
iteration 1200 / 1500: loss 2.939168
iteration 1300 / 1500: loss 2.587487
iteration 1400 / 1500: loss 2.369190
0.361
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.266980 val accurac
y: 0.267000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.347694 val accurac
y: 0.361000
lr 3.000000e-07 reg 5.000000e+05 train accuracy: 0.360286 val accurac
y: 0.365000
lr 3.000000e-07 reg 5.000000e+06 train accuracy: 0.351490 val accurac
y: 0.378000
best validation accuracy achieved during cross-validation: 0.378000
```

In [8]:

```
# evaluate on test set
# Evaluate the best softmax on test set
y_test_pred = best_softmax.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print 'softmax on raw pixels final test set accuracy: %f' % (test_accuracy, )
```

```
softmax on raw pixels final test set accuracy: 0.369000
```

In [9]:

```
# Visualize the learned weights for each class
w = best_softmax.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)

w_min, w_max = np.min(w), np.max(w)

classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', '
for i in xrange(10):
  plt.subplot(2, 5, i + 1)

  # Rescale the weights to be between 0 and 255
  wimg = 255.0 * (w[:, :, :, i].squeeze() - w_min) / (w_max - w_min)
  plt.imshow(wimg.astype('uint8'))
  plt.axis('off')
  plt.title(classes[i])
```

In [ ]: