softmax

January 11, 2016

1 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 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 [2]:
```
import random
import numpy as np
from cs231n.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
```

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

In [3]:
```
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 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 = '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]

    # 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))

    # 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

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

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

1.1 Softmax Classifier

Your code for this section will all be written inside **cs231n/classifiers/softmax.py**.

In [4]:
```
# First implement the naive softmax loss function with nested loops.
# Open the file cs231n/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cs231n.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(10, 3073) * 0.0001
loss, grad = softmax_loss_naive(W, X_train, y_train, 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: 2.352954
sanity check: 2.302585
```

1.2 Inline Question 1:

Why do we expect our loss to be close to -log(0.1)? Explain briefly.**
Your answer: *Fill this in*

In [5]:
```
# 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_train, y_train, 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 cs231n.gradient_check import grad_check_sparse
f = lambda w: softmax_loss_naive(w, X_train, y_train, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)
```

```
numerical: -0.363523 analytic: -0.363523, relative error: 4.031149e-08
numerical: -0.947558 analytic: -0.947558, relative error: 2.887980e-08
numerical: 0.406261 analytic: 0.406261, relative error: 1.465731e-08
numerical: 2.773619 analytic: 2.773619, relative error: 1.267133e-08
numerical: 2.400133 analytic: 2.400133, relative error: 3.195586e-08
numerical: 0.578811 analytic: 0.578811, relative error: 5.938612e-08
numerical: -1.193488 analytic: -1.193488, relative error: 2.080628e-08
numerical: -1.995289 analytic: -1.995289, relative error: 1.341803e-08
numerical: 1.977778 analytic: 1.977778, relative error: 1.310777e-08
numerical: 0.849221 analytic: 0.849221, relative error: 6.014215e-09
```

In [6]:
```
# Now that we have a naive implementation of the softmax loss function and its gradient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version should be
# much faster.
tic = time.time()
loss_naive, grad_naive = softmax_loss_naive(W, X_train, y_train, 0.00001)
toc = time.time()
print 'naive loss: %e computed in %fs' % (loss_naive, toc - tic)

from cs231n.classifiers.softmax import softmax_loss_vectorized
tic = time.time()
loss_vectorized, grad_vectorized = softmax_loss_vectorized(W, X_train, y_train, 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: 2.352954e+00 computed in 0.304337s
vectorized loss: 2.352954e+00 computed in 0.275674s
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 cs231n.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None
learning_rates = [7e-8, 3e-7]
regularization_strengths = [1e4, 3e4]

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained softmax classifer in best_softmax.                          #
################################################################################
import itertools as it
import copy

from cs231n.classifiers import Softmax

for l_r, reg in it.product(learning_rates, regularization_strengths):
    softmax = Softmax()
    loss_hist = softmax.train(X_train, y_train, learning_rate=l_r, reg=reg,
                      num_iters=1500, verbose=True)
    y_train_pred = softmax.predict(X_train)
    y_val_pred = softmax.predict(X_val)

    train_acc = np.mean(y_train == y_train_pred)
    val_acc = np.mean(y_val == y_val_pred)

    if val_acc > best_val:
        best_val = val_acc
        best_softmax = copy.copy(softmax)

    results[(l_r, reg)] = (train_acc, val_acc)

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

# 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 160.079160
iteration 100 / 1500: loss 137.928913
iteration 200 / 1500: loss 119.774292
iteration 300 / 1500: loss 104.158282
iteration 400 / 1500: loss 90.664304
iteration 500 / 1500: loss 78.647715
iteration 600 / 1500: loss 68.583049
iteration 700 / 1500: loss 59.801941
iteration 800 / 1500: loss 52.253669
iteration 900 / 1500: loss 45.581820
iteration 1000 / 1500: loss 39.765775
iteration 1100 / 1500: loss 34.840295
iteration 1200 / 1500: loss 30.392724
iteration 1300 / 1500: loss 26.617322
iteration 1400 / 1500: loss 23.483753
iteration 0 / 1500: loss 466.043294
iteration 100 / 1500: loss 305.406777
iteration 200 / 1500: loss 200.728899
iteration 300 / 1500: loss 132.227693
iteration 400 / 1500: loss 87.567721
iteration 500 / 1500: loss 58.078397
iteration 600 / 1500: loss 38.844220
iteration 700 / 1500: loss 26.039527
iteration 800 / 1500: loss 17.856277
iteration 900 / 1500: loss 12.331856
iteration 1000 / 1500: loss 8.808517
iteration 1100 / 1500: loss 6.470089
iteration 1200 / 1500: loss 4.918743
iteration 1300 / 1500: loss 3.945048
iteration 1400 / 1500: loss 3.345241
iteration 0 / 1500: loss 159.190927
iteration 100 / 1500: loss 86.733656
iteration 200 / 1500: loss 48.267235
iteration 300 / 1500: loss 27.033462
iteration 400 / 1500: loss 15.642430
iteration 500 / 1500: loss 9.404781
iteration 600 / 1500: loss 6.070221
iteration 700 / 1500: loss 4.159821
iteration 800 / 1500: loss 3.203543
iteration 900 / 1500: loss 2.601708
iteration 1000 / 1500: loss 2.264816
iteration 1100 / 1500: loss 2.103841
iteration 1200 / 1500: loss 2.033820
iteration 1300 / 1500: loss 2.050385
iteration 1400 / 1500: loss 1.949405
iteration 0 / 1500: loss 464.241736
iteration 100 / 1500: loss 77.085006
iteration 200 / 1500: loss 14.259585
iteration 300 / 1500: loss 4.059058
iteration 400 / 1500: loss 2.340973
iteration 500 / 1500: loss 2.091586
iteration 600 / 1500: loss 2.039393
iteration 700 / 1500: loss 2.089262
iteration 800 / 1500: loss 2.027608
```

```
iteration 900 / 1500: loss 1.991941
iteration 1000 / 1500: loss 2.062760
iteration 1100 / 1500: loss 2.065556
iteration 1200 / 1500: loss 2.008092
iteration 1300 / 1500: loss 2.034195
iteration 1400 / 1500: loss 1.991874
lr 7.000000e-08 reg 1.000000e+04 train accuracy: 0.293714 val accuracy: 0.298000
lr 7.000000e-08 reg 3.000000e+04 train accuracy: 0.343755 val accuracy: 0.361000
lr 3.000000e-07 reg 1.000000e+04 train accuracy: 0.376490 val accuracy: 0.389000
lr 3.000000e-07 reg 3.000000e+04 train accuracy: 0.340000 val accuracy: 0.352000
best validation accuracy achieved during cross-validation: 0.389000
```

In [8]:
```
# evaluate on test set
# Evaluate the best svm 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, )
```

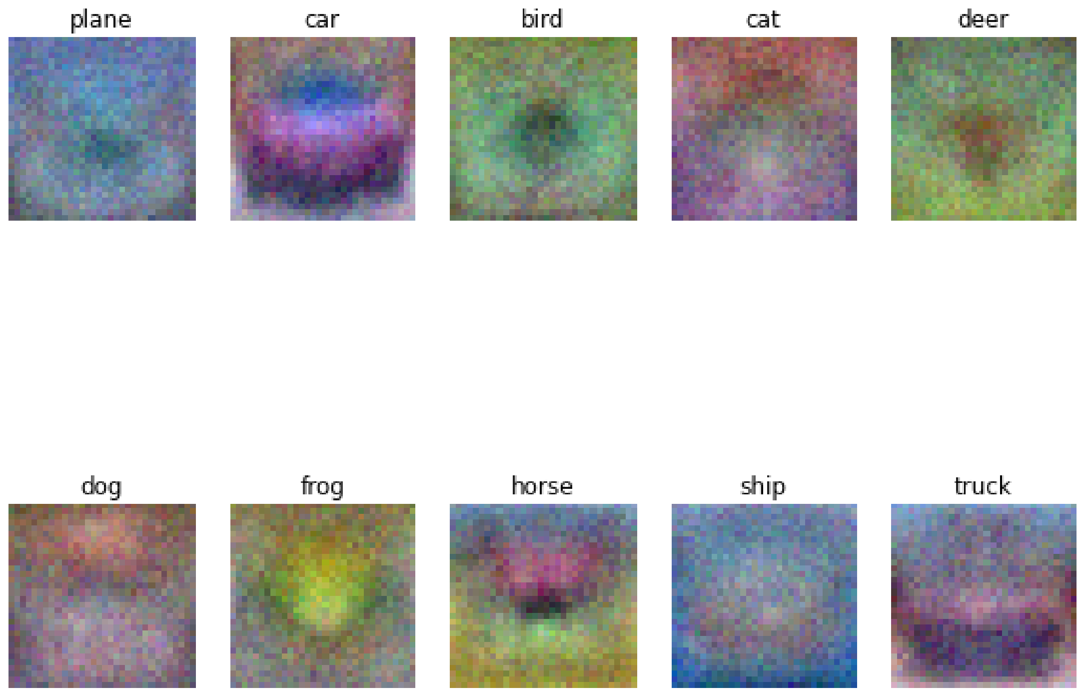
```
softmax on raw pixels final test set accuracy: 0.361000
```

In [9]:
```
# Visualize the learned weights for each class
w = best_softmax.W[:,:-1] # strip out the bias
w = w.reshape(10, 32, 32, 3)

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

classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
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 [ ]:
```