softmax

May 8, 2018

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 [17]:
```
import random
import numpy as np
from hmwk5_2.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

from __future__ import print_function

# Reference: https://github.com/Halfish/cs231n/tree/master/assignment1

%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 [18]: def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, num_dev=5
             """
             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 = 'hmwk5_2/datasets/cifar-10-batches-py'

             X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

             # subsample the data
             mask = list(range(num_training, num_training + num_validation))
             X_val = X_train[mask]
             y_val = y_train[mask]
             mask = list(range(num_training))
             X_train = X_train[mask]
             y_train = y_train[mask]
             mask = list(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


         # Cleaning up variables to prevent loading data multiple times (which may cause memor
```

```
try:
   del X_train, y_train
   del X_test, y_test
   print('Clear previously loaded data.')
except:
   pass

# 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)
```

```
Clear previously loaded data.
Train data shape:  (49000, 3073)
Train labels shape:  (49000,)
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,)
```

1.1 Softmax Classifier

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

In [19]:

```
# 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 hmwk5_2.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: 2.411441
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: The probability for each class is 0.1 since we have 10 classes. Therefore, the log loss is -log(0.1).

In [20]:
```
# 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 hmwk5_2.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, 5e1)
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 5e1)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)
```

```
numerical: -0.814163 analytic: -0.814163, relative error: 2.837687e-09
numerical: 1.002279 analytic: 1.002279, relative error: 3.929047e-09
numerical: 0.594972 analytic: 0.594972, relative error: 1.953488e-08
numerical: 1.797474 analytic: 1.797474, relative error: 2.765331e-08
numerical: 0.063704 analytic: 0.063704, relative error: 1.292642e-07
numerical: 0.188682 analytic: 0.188682, relative error: 1.793670e-07
numerical: -2.539226 analytic: -2.539226, relative error: 1.491037e-09
numerical: 0.815371 analytic: 0.815371, relative error: 5.148228e-08
numerical: -0.741378 analytic: -0.741378, relative error: 6.687431e-09
numerical: 1.023604 analytic: 1.023604, relative error: 1.995192e-08
numerical: 3.163773 analytic: 3.163773, relative error: 1.823740e-08
numerical: -0.437112 analytic: -0.437112, relative error: 3.192093e-09
numerical: -5.682758 analytic: -5.682758, relative error: 5.466139e-09
numerical: -1.724527 analytic: -1.724527, relative error: 2.861011e-08
numerical: 2.520643 analytic: 2.520643, relative error: 4.503707e-09
numerical: -0.240242 analytic: -0.240242, relative error: 7.630763e-08
numerical: -1.677741 analytic: -1.677741, relative error: 1.413622e-08
numerical: -2.649825 analytic: -2.649825, relative error: 7.792913e-10
numerical: 1.273833 analytic: 1.273833, relative error: 6.748062e-09
numerical: -0.652263 analytic: -0.652263, relative error: 2.075119e-08
```

In [21]:
```
# Now that we have a naive implementation of the softmax loss function and its gradie
# implement a vectorized version in softmax_loss_vectorized.
```

```
# The two versions should compute the same results, but the vectorized version should
# much faster.
tic = time.time()
loss_naive, grad_naive = softmax_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from hmwk5_2.classifiers.softmax import softmax_loss_vectorized
tic = time.time()
loss_vectorized, grad_vectorized = softmax_loss_vectorized(W, X_dev, y_dev, 0.000005)
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: 2.411441e+00 computed in 0.098400s
vectorized loss: 2.411441e+00 computed in 0.008782s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [22]:
```
# 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 hmwk5_2.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None
learning_rates = [1e-7, 5e-7]
regularization_strengths = [2.5e4, 5e4]

################################################################################
# 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.                          #
################################################################################
for lr in learning_rates:
    for rs in regularization_strengths:
        softmax = Softmax()
        softmax.train(X_train, y_train, learning_rate = lr, reg=rs, num_iters = 1500,
                      verbose = True)
```

```python
            y_pred_train = softmax.predict(X_train)
            train_accuracy = np.mean(y_pred_train == y_train)

            y_pred_val = softmax.predict(X_val)
            current_accuracy = np.mean(y_pred_val == y_val)
            results[(lr, rs)] = (train_accuracy, current_accuracy)

            if current_accuracy > best_val:
                best_val = current_accuracy
                best_softmax = softmax
    ################################################################################
    #                              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 389.585567
iteration 100 / 1500: loss 235.882251
iteration 200 / 1500: loss 143.151071
iteration 300 / 1500: loss 87.381301
iteration 400 / 1500: loss 53.580661
iteration 500 / 1500: loss 33.268365
iteration 600 / 1500: loss 20.905002
iteration 700 / 1500: loss 13.436617
iteration 800 / 1500: loss 8.994056
iteration 900 / 1500: loss 6.240335
iteration 1000 / 1500: loss 4.556667
iteration 1100 / 1500: loss 3.502107
iteration 1200 / 1500: loss 2.949472
iteration 1300 / 1500: loss 2.544311
iteration 1400 / 1500: loss 2.418960
iteration 0 / 1500: loss 779.154900
iteration 100 / 1500: loss 286.144912
iteration 200 / 1500: loss 105.892203
iteration 300 / 1500: loss 40.068351
iteration 400 / 1500: loss 16.085616
iteration 500 / 1500: loss 7.259269
iteration 600 / 1500: loss 3.952037
iteration 700 / 1500: loss 2.717868
iteration 800 / 1500: loss 2.311429
iteration 900 / 1500: loss 2.142549
iteration 1000 / 1500: loss 2.126397
```

```
iteration 1100 / 1500: loss 2.125953
iteration 1200 / 1500: loss 2.129377
iteration 1300 / 1500: loss 2.082927
iteration 1400 / 1500: loss 2.077699
iteration 0 / 1500: loss 392.871800
iteration 100 / 1500: loss 33.199343
iteration 200 / 1500: loss 4.451756
iteration 300 / 1500: loss 2.162057
iteration 400 / 1500: loss 2.076185
iteration 500 / 1500: loss 2.017442
iteration 600 / 1500: loss 2.011248
iteration 700 / 1500: loss 1.999447
iteration 800 / 1500: loss 2.119726
iteration 900 / 1500: loss 1.980435
iteration 1000 / 1500: loss 2.028366
iteration 1100 / 1500: loss 1.883927
iteration 1200 / 1500: loss 2.058723
iteration 1300 / 1500: loss 2.012429
iteration 1400 / 1500: loss 1.952310
iteration 0 / 1500: loss 767.466308
iteration 100 / 1500: loss 6.868844
iteration 200 / 1500: loss 2.115490
iteration 300 / 1500: loss 2.108528
iteration 400 / 1500: loss 2.123933
iteration 500 / 1500: loss 2.103379
iteration 600 / 1500: loss 2.134838
iteration 700 / 1500: loss 2.059736
iteration 800 / 1500: loss 2.072398
iteration 900 / 1500: loss 2.090008
iteration 1000 / 1500: loss 2.067229
iteration 1100 / 1500: loss 2.091932
iteration 1200 / 1500: loss 2.117153
iteration 1300 / 1500: loss 2.076470
iteration 1400 / 1500: loss 2.099181
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.346367 val accuracy: 0.370000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.333041 val accuracy: 0.349000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.341429 val accuracy: 0.360000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.330980 val accuracy: 0.353000
best validation accuracy achieved during cross-validation: 0.370000
```

In [23]:
```python
# 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: 0.356000
```

Inline Question - *True or False*

It's possible to add a new datapoint to a training set that would leave the SVM loss unchanged, but this is not the case with the Softmax classifier loss.

Your answer: True

Your explanation: In SVM, if the score for new datapoint is out of margin rangem the loss stays unchanged. However, in softmax, the loss will change.

In [24]:
```
# 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', 'tru
for i in range(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])
```