softmax

September 28, 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 [1]: import random
        import numpy as np
        from cs682.data_utils import load_CIFAR10
        import matplotlib.pyplot as plt

        from __future__ import print_function

        %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_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 = 'cs682/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 memory
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)
```

```
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 **cs682/classifiers/softmax.py**.

In [3]:
```
# First implement the naive softmax loss function with nested loops.
# Open the file cs682/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cs682.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: 2.355978
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 of each class 0.1, because there are 10 classes. Hence on an average, we can expect the loss to be -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 cs682.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: 1.866588 analytic: 1.866588, relative error: 2.870578e-08
numerical: -0.777731 analytic: -0.777731, relative error: 4.913355e-08
numerical: 3.048122 analytic: 3.048122, relative error: 4.932950e-09
numerical: 0.378959 analytic: 0.378958, relative error: 1.656307e-07
numerical: -0.976891 analytic: -0.976891, relative error: 3.352302e-08
numerical: -1.707841 analytic: -1.707841, relative error: 1.244064e-08
numerical: -0.748268 analytic: -0.748268, relative error: 2.655352e-08
numerical: 0.381892 analytic: 0.381892, relative error: 1.218675e-07
numerical: 0.044736 analytic: 0.044736, relative error: 9.387103e-08
numerical: 0.417072 analytic: 0.417072, relative error: 3.609779e-09
numerical: 4.627194 analytic: 4.626463, relative error: 7.895356e-05
numerical: -0.706533 analytic: -0.707054, relative error: 3.688338e-04
numerical: 0.229622 analytic: 0.234915, relative error: 1.139498e-02
numerical: 0.643648 analytic: 0.642792, relative error: 6.650322e-04
numerical: -2.861869 analytic: -2.868403, relative error: 1.140199e-03
numerical: 1.028699 analytic: 1.023023, relative error: 2.766471e-03
numerical: 3.339658 analytic: 3.338466, relative error: 1.784581e-04
numerical: -1.248134 analytic: -1.256357, relative error: 3.283001e-03
numerical: -0.407527 analytic: -0.404488, relative error: 3.742294e-03
numerical: -0.759588 analytic: -0.763481, relative error: 2.555840e-03
```

In [5]:
```
# Now that we have a naive implementation of the softmax loss function and its gradien
# 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 cs682.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: 2.355978e+00 computed in 0.140424s
vectorized loss: 2.355978e+00 computed in 0.006468s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [6]:
```
# 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 cs682.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None
learning_rates = [1e-7, 5e-7]
regularization_strengths = [2.5e4, 5e4]

softmax1 = Softmax()
for i in learning_rates:
    for j in regularization_strengths:
        loss_history = softmax1.train(X_train, y_train, learning_rate=i, reg=j, num_ite
        y_train_prediction = softmax1.predict(X_train)
        accuracy_train = np.mean(y_val == y_train_prediction)
        y_val_prediction = softmax1.predict(X_val)
        accuracy_validation = np.mean(y_val == y_val_prediction)

        results[(i,j)] = accuracy_train, accuracy_validation
        if accuracy_validation > best_val:
            best_val=accuracy_validation
            best_softmax = softmax1
################################################################################
# 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.                          #
################################################################################
# Your code
################################################################################
```

```
    #                              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 761.533042
iteration 100 / 1500: loss 460.528496
iteration 200 / 1500: loss 279.488481
iteration 300 / 1500: loss 169.764818
iteration 400 / 1500: loss 103.437518
iteration 500 / 1500: loss 63.507726
iteration 600 / 1500: loss 39.245693
iteration 700 / 1500: loss 24.554745
iteration 800 / 1500: loss 15.727254
iteration 900 / 1500: loss 10.341743
iteration 1000 / 1500: loss 7.065417
iteration 1100 / 1500: loss 5.082982
iteration 1200 / 1500: loss 3.942867
iteration 1300 / 1500: loss 3.249523
iteration 1400 / 1500: loss 2.783591


/home/akhila/.local/lib/python3.6/site-packages/ipykernel_launcher.py:17: DeprecationWarning: 


iteration 0 / 1500: loss 3.140750
iteration 100 / 1500: loss 2.529964
iteration 200 / 1500: loss 2.324758
iteration 300 / 1500: loss 2.236533
iteration 400 / 1500: loss 2.181742
iteration 500 / 1500: loss 2.148546
iteration 600 / 1500: loss 2.224669
iteration 700 / 1500: loss 2.113984
iteration 800 / 1500: loss 2.085940
iteration 900 / 1500: loss 2.176372
iteration 1000 / 1500: loss 2.196403
iteration 1100 / 1500: loss 2.122962
iteration 1200 / 1500: loss 2.182554
iteration 1300 / 1500: loss 2.224660
iteration 1400 / 1500: loss 2.150173
iteration 0 / 1500: loss 2.113960
iteration 100 / 1500: loss 2.193093
```

```
iteration 200 / 1500: loss 2.140415
iteration 300 / 1500: loss 2.194579
iteration 400 / 1500: loss 2.105710
iteration 500 / 1500: loss 2.055973
iteration 600 / 1500: loss 2.085644
iteration 700 / 1500: loss 2.152067
iteration 800 / 1500: loss 2.146090
iteration 900 / 1500: loss 2.145500
iteration 1000 / 1500: loss 2.063160
iteration 1100 / 1500: loss 2.007901
iteration 1200 / 1500: loss 2.152323
iteration 1300 / 1500: loss 2.196219
iteration 1400 / 1500: loss 2.050577
iteration 0 / 1500: loss 2.344331
iteration 100 / 1500: loss 2.176658
iteration 200 / 1500: loss 2.134563
iteration 300 / 1500: loss 2.111681
iteration 400 / 1500: loss 2.143278
iteration 500 / 1500: loss 2.215188
iteration 600 / 1500: loss 2.184355
iteration 700 / 1500: loss 2.197539
iteration 800 / 1500: loss 2.186359
iteration 900 / 1500: loss 2.190039
iteration 1000 / 1500: loss 2.212629
iteration 1100 / 1500: loss 2.184437
iteration 1200 / 1500: loss 2.244856
iteration 1300 / 1500: loss 2.225584
iteration 1400 / 1500: loss 2.160992
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.000000 val accuracy: 0.358000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.000000 val accuracy: 0.342000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.000000 val accuracy: 0.356000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.000000 val accuracy: 0.336000
best validation accuracy achieved during cross-validation: 0.358000
```

```
In [7]: # 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: 0.328000
```

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, the decision boundary is not significantly effected by one training point. For softmax loss, the exponential of a small value can cause large changes in loss.

In [8]:
```
# 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', 'truc
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])
```