softmax

June 7, 2020

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 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
```

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 = 'cs231n/datasets/cifar-10-batches-py'
```

```
    # Cleaning up variables to prevent loading data multiple times (which may cause me
    try:
       del X_train, y_train
       del X_test, y_test
       print('Clear previously loaded data.')
    except:
       pass

    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


# 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 `cs231n/classifiers/softmax.py`.

```
In [3]: # 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(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: 2.378047
sanity check: 2.302585
```

Inline Question 1

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

Your Answer : Since above we're given 10 classes randomly, we can expect that on average each of these classes have a probability of 0.1. By applying the softmax formula, we get the loss to be close to $-log(0.1)$.

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 cs231n.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: -0.300047 analytic: -0.300047, relative error: 1.179609e-07
numerical: 0.455394 analytic: 0.455394, relative error: 2.989037e-08
numerical: 0.965712 analytic: 0.965712, relative error: 5.507721e-08
numerical: 0.229218 analytic: 0.229218, relative error: 2.437353e-07
numerical: 0.560912 analytic: 0.560912, relative error: 7.948891e-08
numerical: 0.539819 analytic: 0.539819, relative error: 1.719193e-08
numerical: 1.721108 analytic: 1.721108, relative error: 3.625317e-08
numerical: -1.164813 analytic: -1.164813, relative error: 1.654328e-08
numerical: 0.231073 analytic: 0.231073, relative error: 4.762176e-08
numerical: 0.664233 analytic: 0.664233, relative error: 1.590655e-08
numerical: 0.059044 analytic: 0.059044, relative error: 1.011073e-07
numerical: -0.048454 analytic: -0.048454, relative error: 2.232189e-07
numerical: -0.921155 analytic: -0.921155, relative error: 2.591088e-08
numerical: 4.228444 analytic: 4.228444, relative error: 1.149242e-08
numerical: 0.151069 analytic: 0.151069, relative error: 1.752625e-07
numerical: -0.598337 analytic: -0.598337, relative error: 1.920326e-08
numerical: -0.867658 analytic: -0.867658, relative error: 1.012421e-08
numerical: -0.009551 analytic: -0.009551, relative error: 5.697257e-08
numerical: -0.781547 analytic: -0.781547, relative error: 5.682646e-09
numerical: 0.323526 analytic: 0.323526, relative error: 8.074568e-08
```

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))
# print(grad_naive)
from cs231n.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))
        # print(grad_vectorized)
        # 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: 2.378047e+00 computed in 0.171589s
vectorized loss: 2.378047e+00 computed in 0.005893s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [9]:
```
# 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

################################################################################
# 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.                          #
################################################################################

learning_rates = [1e-8, 5e-8, 1e-7, 5e-7, 1e-6, 5e-6]
regularization_strengths = [1e2, 5e2, 1e3, 5e3, 1e4, 5e4, 1e5, 5e5]

for lr in learning_rates:
    for r in regularization_strengths:
        classifier = Softmax()
        classifier.train(X_train, y_train, learning_rate=lr, reg=r, num_iters=1000)
        train_pred = classifier.predict(X_train)
        val_pred = classifier.predict(X_val)
        train_acc = np.mean(y_train == train_pred)
        val_acc = np.mean(y_val == val_pred)
        if val_acc > best_val:
            best_val = val_acc
            best_softmax = classifier
        results[(lr, r)] = (train_acc, val_acc)
```

```
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        # 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)
```

```
lr 1.000000e-08 reg 1.000000e+02 train accuracy: 0.126878 val accuracy: 0.130000
lr 1.000000e-08 reg 5.000000e+02 train accuracy: 0.150980 val accuracy: 0.152000
lr 1.000000e-08 reg 1.000000e+03 train accuracy: 0.110918 val accuracy: 0.108000
lr 1.000000e-08 reg 5.000000e+03 train accuracy: 0.135408 val accuracy: 0.119000
lr 1.000000e-08 reg 1.000000e+04 train accuracy: 0.139551 val accuracy: 0.145000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.156837 val accuracy: 0.171000
lr 5.000000e-08 reg 1.000000e+02 train accuracy: 0.191102 val accuracy: 0.199000
lr 5.000000e-08 reg 5.000000e+02 train accuracy: 0.218694 val accuracy: 0.227000
lr 5.000000e-08 reg 1.000000e+03 train accuracy: 0.201408 val accuracy: 0.203000
lr 5.000000e-08 reg 5.000000e+03 train accuracy: 0.220714 val accuracy: 0.215000
lr 5.000000e-08 reg 1.000000e+04 train accuracy: 0.245490 val accuracy: 0.268000
lr 5.000000e-08 reg 5.000000e+04 train accuracy: 0.307796 val accuracy: 0.322000
lr 1.000000e-07 reg 1.000000e+02 train accuracy: 0.226367 val accuracy: 0.242000
lr 1.000000e-07 reg 5.000000e+02 train accuracy: 0.236612 val accuracy: 0.246000
lr 1.000000e-07 reg 1.000000e+03 train accuracy: 0.241796 val accuracy: 0.238000
lr 1.000000e-07 reg 5.000000e+03 train accuracy: 0.288694 val accuracy: 0.299000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.327551 val accuracy: 0.339000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.308143 val accuracy: 0.324000
lr 5.000000e-07 reg 1.000000e+02 train accuracy: 0.307490 val accuracy: 0.308000
lr 5.000000e-07 reg 5.000000e+02 train accuracy: 0.336143 val accuracy: 0.333000
lr 5.000000e-07 reg 1.000000e+03 train accuracy: 0.358000 val accuracy: 0.356000
lr 5.000000e-07 reg 5.000000e+03 train accuracy: 0.373429 val accuracy: 0.370000
lr 5.000000e-07 reg 1.000000e+04 train accuracy: 0.349367 val accuracy: 0.354000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.296673 val accuracy: 0.308000
lr 1.000000e-06 reg 1.000000e+02 train accuracy: 0.344429 val accuracy: 0.324000
lr 1.000000e-06 reg 5.000000e+02 train accuracy: 0.379612 val accuracy: 0.377000
lr 1.000000e-06 reg 1.000000e+03 train accuracy: 0.395184 val accuracy: 0.378000
lr 1.000000e-06 reg 5.000000e+03 train accuracy: 0.368061 val accuracy: 0.366000
lr 1.000000e-06 reg 1.000000e+04 train accuracy: 0.348551 val accuracy: 0.349000
lr 1.000000e-06 reg 5.000000e+04 train accuracy: 0.280980 val accuracy: 0.292000
lr 5.000000e-06 reg 1.000000e+02 train accuracy: 0.388184 val accuracy: 0.365000
lr 5.000000e-06 reg 5.000000e+02 train accuracy: 0.389408 val accuracy: 0.392000
lr 5.000000e-06 reg 1.000000e+03 train accuracy: 0.339102 val accuracy: 0.331000
lr 5.000000e-06 reg 5.000000e+03 train accuracy: 0.316735 val accuracy: 0.341000
lr 5.000000e-06 reg 1.000000e+04 train accuracy: 0.271102 val accuracy: 0.270000
lr 5.000000e-06 reg 5.000000e+04 train accuracy: 0.136000 val accuracy: 0.141000
best validation accuracy achieved during cross-validation: 0.392000
```

```
In [10]: # 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: 0.376000
```

Inline Question 2 - *True or False*

Suppose the overall training loss is defined as the sum of the per-datapoint loss over all training examples. It is 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.

YourAnswer : Yes

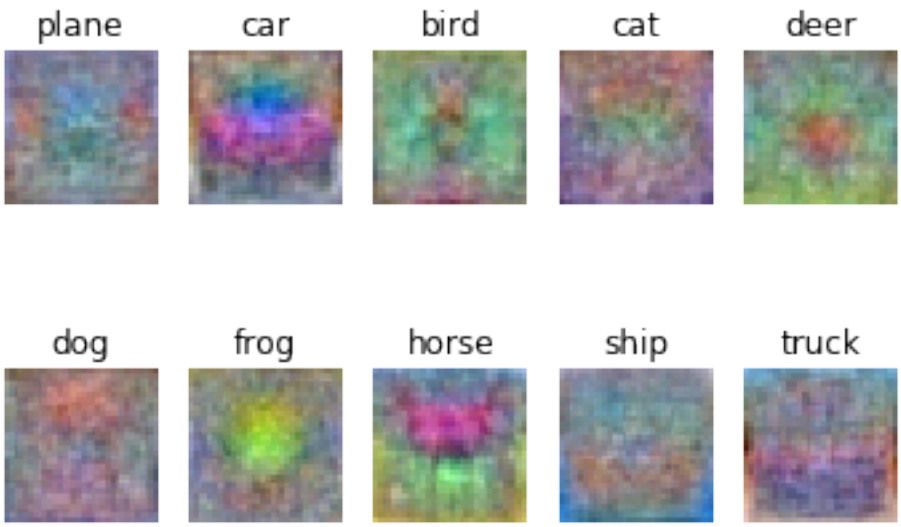
YourExplanation : If the new data point has a score that is out of the margin range from the correct class score, SVM loss wouldn't change. However, in Softmax loss, if the score of the new added datapoint be close to a very large number, it will adversely affect the loss, but definitely the loss of Softmax will change.

```
In [11]: # 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])
```



---

2 IMPORTANT

This is the end of this question. Please do the following:

1. Click `File -> Save` to make sure the latest checkpoint of this notebook is saved to your Drive.
2. Execute the cell below to download the modified `.py` files back to your drive.