softmax

October 21, 2019

c. Q3: Implement a Softmax classifier의 결과를 작성한 코드와 함께 출력하세요.

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=50
    """
    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: 2.374962
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 초기에 weight에서는 모든 class가 같은 비율로 선택되어 질 수 있기 때문에 -log(0.1)을 기대한다. 그리고 CIFAR-10는 10개의 class를 가지고 있기 때문에 correct class의 probability는 초기에는 0.1의 값으로 예상할 수 있기 때문에이다. 여기서 softmax는 correct class의 negative log probability이다.

```
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: -4.923950 analytic: -4.923950, relative error: 1.052514e-08
numerical: 1.046927 analytic: 1.046927, relative error: 5.461082e-09
numerical: 2.108643 analytic: 2.108643, relative error: 2.563264e-08
numerical: 1.238681 analytic: 1.238681, relative error: 2.901188e-08
numerical: 0.821542 analytic: 0.821542, relative error: 1.566416e-08
numerical: 1.766678 analytic: 1.766678, relative error: 3.110967e-08
numerical: 1.544118 analytic: 1.544118, relative error: 4.672810e-08
numerical: 0.630696 analytic: 0.630696, relative error: 7.762893e-08
numerical: -0.575142 analytic: -0.575142, relative error: 1.330414e-08
numerical: 2.480511 analytic: 2.480511, relative error: 1.270990e-08
numerical: 0.586478 analytic: 0.586478, relative error: 1.332123e-07
numerical: -1.164067 analytic: -1.164067, relative error: 2.412108e-08
numerical: 0.325806 analytic: 0.325806, relative error: 2.252981e-07
numerical: 2.517754 analytic: 2.517753, relative error: 1.874770e-08
numerical: 0.550184 analytic: 0.550184, relative error: 1.925119e-07
numerical: 2.034174 analytic: 2.034174, relative error: 2.456839e-08
numerical: -3.080956 analytic: -3.080956, relative error: 8.182032e-09
numerical: -0.577468 analytic: -0.577469, relative error: 1.846150e-07
numerical: 0.730723 analytic: 0.730723, relative error: 3.750990e-08
numerical: 1.129648 analytic: 1.129648, relative error: 4.834641e-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))

        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))

# 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: 2.374962e+00 computed in 0.091153s
vectorized loss: 2.374962e+00 computed in 0.011305s
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 cs231n.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.                          #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

grid_search = [ (lr, rg) for lr in learning_rates for rg in regularization_strengths]

for lr, rg in grid_search:
    # Create a new Softmax instance
    softmax_model = Softmax()
    # Train the model with current parameters
    softmax_model.train(X_train, y_train, learning_rate=lr, reg=rg, num_iters=1000)
    # Predict values for training set
    y_train_pred = softmax_model.predict(X_train)
    # Calculate accuracy
    train_accuracy = np.mean(y_train_pred == y_train)
    # Predict values for validation set
    y_val_pred = softmax_model.predict(X_val)
```

```
            # Calculate accuracy
            val_accuracy = np.mean(y_val_pred == y_val)
            # Save results
            results[(lr,rg)] = (train_accuracy, val_accuracy)
            if best_val < val_accuracy:
                best_val = val_accuracy
                best_softmax = softmax_model

        # *****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-07 reg 2.500000e+04 train accuracy: 0.332041 val accuracy: 0.360000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.307551 val accuracy: 0.328000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.319061 val accuracy: 0.324000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.300347 val accuracy: 0.319000
best validation accuracy achieved during cross-validation: 0.360000
```

```
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: 0.350000
```

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.

$Your Answer :$ True

$Your Explanation :$ 예를 들어, scores [9, 8, 7]를 출력하는 새로운 데이터를 포함한다면, margin을 1로 하고 correct class가 0이면 SVM의 loss값은 0이 되어 SVM의 loss는 변하지 않는다. 하지만 softmax는 상대적인 새로운 데이터가 넣어지만 항상 확률 값을 내놓아 softmax의 loss가 변한다.

```
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)
```

```python
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])
```