

softmax

October 27, 2019

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 external 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=500)
    """
    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 memo
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: 2.348464
sanity check: 2.302585

```

Inline Question 1

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

Your Answer: There are ten classes here, so if the scores are random and mostly equal, we expect the ratio in the softmax formula to be ≈ 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: -0.465569 analytic: -0.465569, relative error: 1.148105e-07
numerical: 1.734552 analytic: 1.734552, relative error: 9.144844e-09
numerical: 1.993410 analytic: 1.993410, relative error: 2.555822e-10
numerical: 2.034620 analytic: 2.034620, relative error: 2.061479e-09
numerical: 6.974875 analytic: 6.974874, relative error: 7.055669e-09
numerical: 0.807704 analytic: 0.807704, relative error: 1.547988e-08
numerical: 0.878877 analytic: 0.878877, relative error: 4.585847e-08
numerical: 0.284163 analytic: 0.284163, relative error: 2.087217e-07
numerical: -0.481440 analytic: -0.481440, relative error: 3.016805e-08
numerical: 1.940488 analytic: 1.940488, relative error: 3.649122e-08
numerical: 2.249105 analytic: 2.249105, relative error: 3.846784e-08
numerical: 2.570815 analytic: 2.570815, relative error: 1.149731e-08
numerical: 2.126309 analytic: 2.126309, relative error: 5.753572e-09
numerical: 1.865551 analytic: 1.865551, relative error: 1.260893e-08
numerical: -0.209247 analytic: -0.209248, relative error: 1.335191e-07
numerical: -2.224795 analytic: -2.224795, relative error: 5.236434e-09
numerical: 0.885811 analytic: 0.885811, relative error: 3.763822e-08
numerical: 0.776535 analytic: 0.776535, relative error: 9.893845e-08
numerical: 0.445314 analytic: 0.445314, relative error: 1.090203e-07
numerical: -4.220672 analytic: -4.220672, relative error: 1.494104e-08

In [5]: # 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_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: 2.348464e+00 computed in 0.163326s
vectorized loss: 2.348464e+00 computed in 0.020444s
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 classifier in best_softmax.                         #
#####
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

```

```

i = 0
for lr in learning_rates:
    for reg in regularization_strengths:
        i+=1
        model = Softmax()
        model.train(X_train, y_train, learning_rate=lr, reg=reg, num_iters=1500, verbose=False)
        y_train_pred = model.predict(X_train)
        train_acc = np.mean(y_train == y_train_pred)
        y_val_pred = model.predict(X_val)
        val_acc = np.mean(y_val == y_val_pred)
        results[(lr, reg)] = (train_acc, val_acc)
        if best_val < val_acc:
            best_val = val_acc
            best_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: 0.322551 val accuracy: 0.336000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.311755 val accuracy: 0.325000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.316571 val accuracy: 0.335000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.303429 val accuracy: 0.305000
best validation accuracy achieved during cross-validation: 0.336000

```

```

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: 0.332000

```

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 : Your Explanation : Hinge (or SVM) loss can be strictly equal to zero for data points with big enough margin. But logarithmic loss (Softmax classifier loss) is always positive.

```

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)

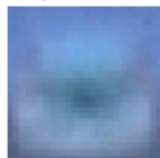
w_min, w_max = np.min(w), np.max(w)

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

```

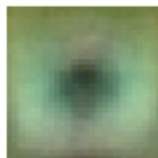
plane



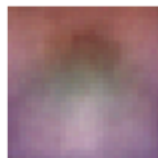
car



bird



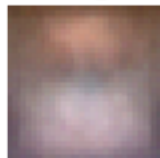
cat



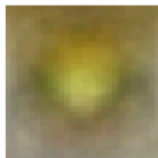
deer



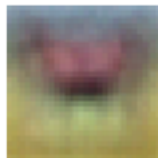
dog



frog



horse



ship



truck

