softmax

September 29, 2021

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

[1]:
```
# Setup Colab connection with Drive
from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)

# Define working directory
FOLDERNAME = 'Coursework/Fall 2021/682/Assignments/assignment1'
assert FOLDERNAME is not None, "[!] Enter the foldername."

import sys
sys.path.append(f'/content/gdrive/My Drive/{FOLDERNAME}')
```

```
Mounted at /content/gdrive
```

[2]:
```
from __future__ import print_function
import random
import numpy as np
from cs682.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
```

[3]:
```
#Switch to working directory
%cd gdrive/My\ Drive/$FOLDERNAME
```

```
/content/gdrive/My Drive/Coursework/Fall 2021/682/Assignments/assignment1
```

[4]:
```
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 = '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
```

```python
    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 issue)
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**.

[5]:
```python
# 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: 2.323482
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: We expect loss to be close to -log(0.1) as we are not learning any parameters at this point and we expect each class (out of 10) to be equally likely when using a randomly initialized initial W.

[6]:
```python
# 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: -0.284203 analytic: -0.284203, relative error: 8.187001e-08
numerical: 0.725570 analytic: 0.725570, relative error: 1.115359e-07
numerical: 0.915656 analytic: 0.915656, relative error: 4.604942e-08
numerical: 3.544753 analytic: 3.544753, relative error: 1.267183e-08
numerical: 0.609784 analytic: 0.609784, relative error: 1.791180e-07
numerical: 0.472553 analytic: 0.472553, relative error: 1.343816e-07
numerical: 1.911679 analytic: 1.911679, relative error: 5.556583e-08
numerical: -0.878860 analytic: -0.878860, relative error: 3.859701e-08
```

```
numerical: 1.211142 analytic: 1.211142, relative error: 5.585657e-08
numerical: 1.645632 analytic: 1.645632, relative error: 2.717919e-08
numerical: 0.447980 analytic: 0.447980, relative error: 7.554675e-08
numerical: -1.055898 analytic: -1.055898, relative error: 3.890911e-10
numerical: -0.960308 analytic: -0.960308, relative error: 1.570917e-08
numerical: -2.028626 analytic: -2.028626, relative error: 9.224981e-09
numerical: 2.628991 analytic: 2.628991, relative error: 2.425265e-08
numerical: 2.341801 analytic: 2.341801, relative error: 1.486300e-08
numerical: 0.507949 analytic: 0.507949, relative error: 1.452742e-07
numerical: 2.391970 analytic: 2.391970, relative error: 1.223744e-08
numerical: -2.085640 analytic: -2.085640, relative error: 2.415905e-08
numerical: -1.401349 analytic: -1.401349, relative error: 3.581999e-08
```

[7]:
```
# 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 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: 2.323482e+00 computed in 0.113843s
vectorized loss: 2.323482e+00 computed in 0.017536s
Loss difference: 0.000000
Gradient difference: 0.000000
```

[8]:
```
# 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
```

```python
results = {}
best_val = -1
best_softmax = None
learning_rates = [1e-7, 2e-7, 3e-7, 4e-7, 5e-7]
regularization_strengths = [2.5e4, 3e4, 3.5e4, 4e4, 4.5e4, 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.                          ␣
↪#
################################################################################
# Your code
with np.errstate(all='ignore'):
  for lr in learning_rates:
    for r in regularization_strengths:
      sm = Softmax()
      loss_hist = sm.train(X_train, y_train, learning_rate=lr, reg=r,␣
↪num_iters=1500, verbose=False)
      y_train_pred = sm.predict(X_train)
      train_acc = np.mean(y_train == y_train_pred)
      y_val_pred = sm.predict(X_val)
      val_acc = np.mean(y_val == y_val_pred)
      results[(lr,r)] = (train_acc, val_acc)
      if val_acc > best_val:
        best_val = val_acc
        best_softmax = sm
################################################################################
#                              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)
```

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.335592 val accuracy: 0.347000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.324735 val accuracy: 0.333000
```

```
lr 1.000000e-07 reg 3.500000e+04 train accuracy: 0.316980 val accuracy: 0.336000
lr 1.000000e-07 reg 4.000000e+04 train accuracy: 0.317735 val accuracy: 0.329000
lr 1.000000e-07 reg 4.500000e+04 train accuracy: 0.310122 val accuracy: 0.326000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.308143 val accuracy: 0.318000
lr 2.000000e-07 reg 2.500000e+04 train accuracy: 0.329653 val accuracy: 0.343000
lr 2.000000e-07 reg 3.000000e+04 train accuracy: 0.318816 val accuracy: 0.339000
lr 2.000000e-07 reg 3.500000e+04 train accuracy: 0.323020 val accuracy: 0.336000
lr 2.000000e-07 reg 4.000000e+04 train accuracy: 0.318163 val accuracy: 0.336000
lr 2.000000e-07 reg 4.500000e+04 train accuracy: 0.304531 val accuracy: 0.320000
lr 2.000000e-07 reg 5.000000e+04 train accuracy: 0.293571 val accuracy: 0.316000
lr 3.000000e-07 reg 2.500000e+04 train accuracy: 0.333571 val accuracy: 0.343000
lr 3.000000e-07 reg 3.000000e+04 train accuracy: 0.316816 val accuracy: 0.333000
lr 3.000000e-07 reg 3.500000e+04 train accuracy: 0.328102 val accuracy: 0.340000
lr 3.000000e-07 reg 4.000000e+04 train accuracy: 0.311367 val accuracy: 0.329000
lr 3.000000e-07 reg 4.500000e+04 train accuracy: 0.317673 val accuracy: 0.323000
lr 3.000000e-07 reg 5.000000e+04 train accuracy: 0.298061 val accuracy: 0.308000
lr 4.000000e-07 reg 2.500000e+04 train accuracy: 0.328469 val accuracy: 0.343000
lr 4.000000e-07 reg 3.000000e+04 train accuracy: 0.321224 val accuracy: 0.338000
lr 4.000000e-07 reg 3.500000e+04 train accuracy: 0.308816 val accuracy: 0.328000
lr 4.000000e-07 reg 4.000000e+04 train accuracy: 0.314102 val accuracy: 0.329000
lr 4.000000e-07 reg 4.500000e+04 train accuracy: 0.310143 val accuracy: 0.324000
lr 4.000000e-07 reg 5.000000e+04 train accuracy: 0.305918 val accuracy: 0.310000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.324776 val accuracy: 0.330000
lr 5.000000e-07 reg 3.000000e+04 train accuracy: 0.322633 val accuracy: 0.330000
lr 5.000000e-07 reg 3.500000e+04 train accuracy: 0.318041 val accuracy: 0.334000
lr 5.000000e-07 reg 4.000000e+04 train accuracy: 0.311837 val accuracy: 0.333000
lr 5.000000e-07 reg 4.500000e+04 train accuracy: 0.312224 val accuracy: 0.328000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.307531 val accuracy: 0.321000
best validation accuracy achieved during cross-validation: 0.347000
```

[9]:

```
# 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: 0.343000
```

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: It is possible to add a data point which is outside the SVM margin, and hence would contribute 0 to the loss. However addition of a data point would change the data probability distribution and hence would affect the softmax loss.

[10]:

```
# Visualize the learned weights for each class
w = best_softmax.W[:-1,:] # strip out the bias
```

```python
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])
```