Sunay Bhat - softmax_nosol

January 22, 2022

0.1 This is the softmax workbook for ECE C147/C247 Assignment #2

Please follow the notebook linearly to implement a softmax classifier.

Please print out the workbook entirely when completed.

The goal of this workbook is to give you experience with training a softmax classifier.

[1]:
```
import random
import numpy as np
from utils.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

%matplotlib inline
%load_ext autoreload
%autoreload 2
```

[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 = 'cifar-10-batches-py' # You need to update this line
    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)
```

```python
    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,)
```

0.2 Training a softmax classifier.

The following cells will take you through building a softmax classifier. You will implement its loss function, then subsequently train it with gradient descent. Finally, you will choose the learning rate of gradient descent to optimize its classification performance.

[3]:
```
from nndl import Softmax
```

[4]:
```
# Declare an instance of the Softmax class.
# Weights are initialized to a random value.
# Note, to keep people's first solutions consistent, we are going to use a␣
↪random seed.

np.random.seed(1)

num_classes = len(np.unique(y_train))
num_features = X_train.shape[1]

softmax = Softmax(dims=[num_classes, num_features])
```

Softmax loss

[5]:
```
## Implement the loss function of the softmax using a for loop over
#  the number of examples

loss = softmax.loss(X_train, y_train)
```

[6]:
```
print(loss)
```

```
2.327760702804874
```

0.3 Question:

You'll notice the loss returned by the softmax is about 2.3 (if implemented correctly). Why does this make sense?

0.4 Answer:

Without any training, our loss should be equivalent to the **natural log of random chance**. In this case with 10 classes, that is $\ln(10)$ which is ≈ 2.3.

Softmax gradient

[7]:
```
## Calculate the gradient of the softmax loss in the Softmax class.
# For convenience, we'll write one function that computes the loss
#   and gradient together, softmax.loss_and_grad(X, y)
# You may copy and paste your loss code from softmax.loss() here, and then
#   use the appropriate intermediate values to calculate the gradient.

loss, grad = softmax.loss_and_grad(X_dev,y_dev)
```

```
# Compare your gradient to a gradient check we wrote.
# You should see relative gradient errors on the order of 1e-07 or less if you␣
↪implemented the gradient correctly.
softmax.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: -0.613715 analytic: -0.613715, relative error: 2.040999e-08
numerical: 1.704663 analytic: 1.704662, relative error: 1.935561e-08
numerical: 2.194071 analytic: 2.194071, relative error: 1.067169e-08
numerical: 2.129312 analytic: 2.129311, relative error: 1.520441e-08
numerical: 0.843848 analytic: 0.843848, relative error: 1.152302e-07
numerical: 1.913921 analytic: 1.913921, relative error: 2.743260e-08
numerical: -1.958870 analytic: -1.958870, relative error: 3.339779e-08
numerical: -0.758563 analytic: -0.758563, relative error: 2.882746e-08
numerical: 0.429284 analytic: 0.429284, relative error: 4.920839e-08
numerical: -1.769676 analytic: -1.769676, relative error: 4.309108e-08
```

0.5 A vectorized version of Softmax

To speed things up, we will vectorize the loss and gradient calculations. This will be helpful for stochastic gradient descent.

[8]:
```
import time
```

[9]:
```
## Implement softmax.fast_loss_and_grad which calculates the loss and gradient
#    WITHOUT using any for loops.

# Standard loss and gradient
tic = time.time()
loss, grad = softmax.loss_and_grad(X_dev, y_dev)
toc = time.time()
print('Normal loss / grad_norm: {} / {} computed in {}s'.format(loss, np.linalg.
↪norm(grad, 'fro'), toc - tic))

tic = time.time()
loss_vectorized, grad_vectorized = softmax.fast_loss_and_grad(X_dev, y_dev)
toc = time.time()
print('Vectorized loss / grad: {} / {} computed in {}s'.format(loss_vectorized,␣
↪np.linalg.norm(grad_vectorized, 'fro'), toc - tic))

# The losses should match but your vectorized implementation should be much␣
↪faster.
print('difference in loss / grad: {} /{} '.format(loss - loss_vectorized, np.
↪linalg.norm(grad - grad_vectorized)))

# You should notice a speedup with the same output.
```

```
Normal loss / grad_norm: 2.3124758066980844 / 332.7585134313542 computed in
0.22177624702453613s
```

```
Vectorized loss / grad: 2.312475806698085 / 332.7585134313541 computed in
0.019501924514770508s
difference in loss / grad: -4.440892098500626e-16 /2.1250669706601915e-13
```

0.6 Stochastic gradient descent

We now implement stochastic gradient descent. This uses the same principles of gradient descent we discussed in class, however, it calculates the gradient by only using examples from a subset of the training set (so each gradient calculation is faster).

0.7 Question:

How should the softmax gradient descent training step differ from the svm training step, if at all?

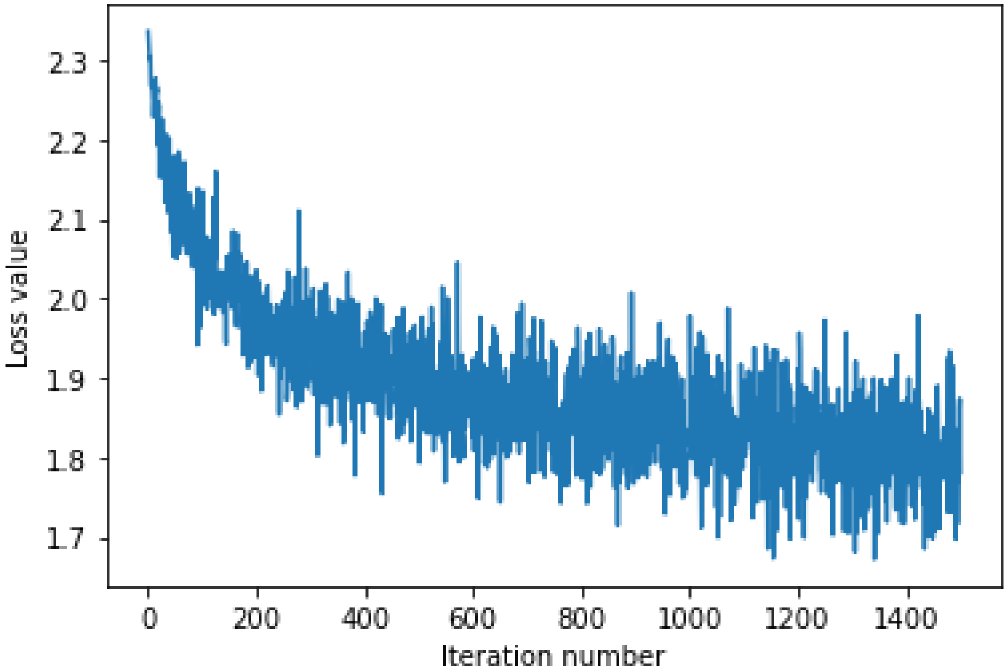
0.8 Answer:

Ignoring as per instructor guidance (Piazza question 125)

[10]:
```python
# Implement softmax.train() by filling in the code to extract a batch of data
# and perform the gradient step.
import time


tic = time.time()
loss_hist = softmax.train(X_train, y_train, learning_rate=1e-7,
                          num_iters=1500, verbose=True)
toc = time.time()
print('That took {}s'.format(toc - tic))

plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```

```
iteration 0 / 1500: loss 2.3365926606637544
iteration 100 / 1500: loss 2.0557222613850827
iteration 200 / 1500: loss 2.0357745120662813
iteration 300 / 1500: loss 1.9813348165609888
iteration 400 / 1500: loss 1.9583142443981612
iteration 500 / 1500: loss 1.862265307354135
iteration 600 / 1500: loss 1.8532611454359382
iteration 700 / 1500: loss 1.8353062223725827
iteration 800 / 1500: loss 1.829389246882764
iteration 900 / 1500: loss 1.899215853035748
iteration 1000 / 1500: loss 1.9783503540252299
iteration 1100 / 1500: loss 1.8470797913532633
iteration 1200 / 1500: loss 1.8411450268664082
iteration 1300 / 1500: loss 1.7910402495792102
```

```
iteration 1400 / 1500: loss 1.8705803029382257
That took 7.113604784011841s
```



0.8.1 Evaluate the performance of the trained softmax classifier on the validation data.

[11]:
```
## Implement softmax.predict() and use it to compute the training and testing␣
↪error.

y_train_pred = softmax.predict(X_train)
print('training accuracy: {}'.format(np.mean(np.equal(y_train,y_train_pred), )))
y_val_pred = softmax.predict(X_val)
print('validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val_pred)), ))
```

```
training accuracy: 0.3811428571428571
validation accuracy: 0.398
```

0.9 Optimize the softmax classifier

You may copy and paste your optimization code from the SVM here.

[12]:
```
np.finfo(float).eps
```

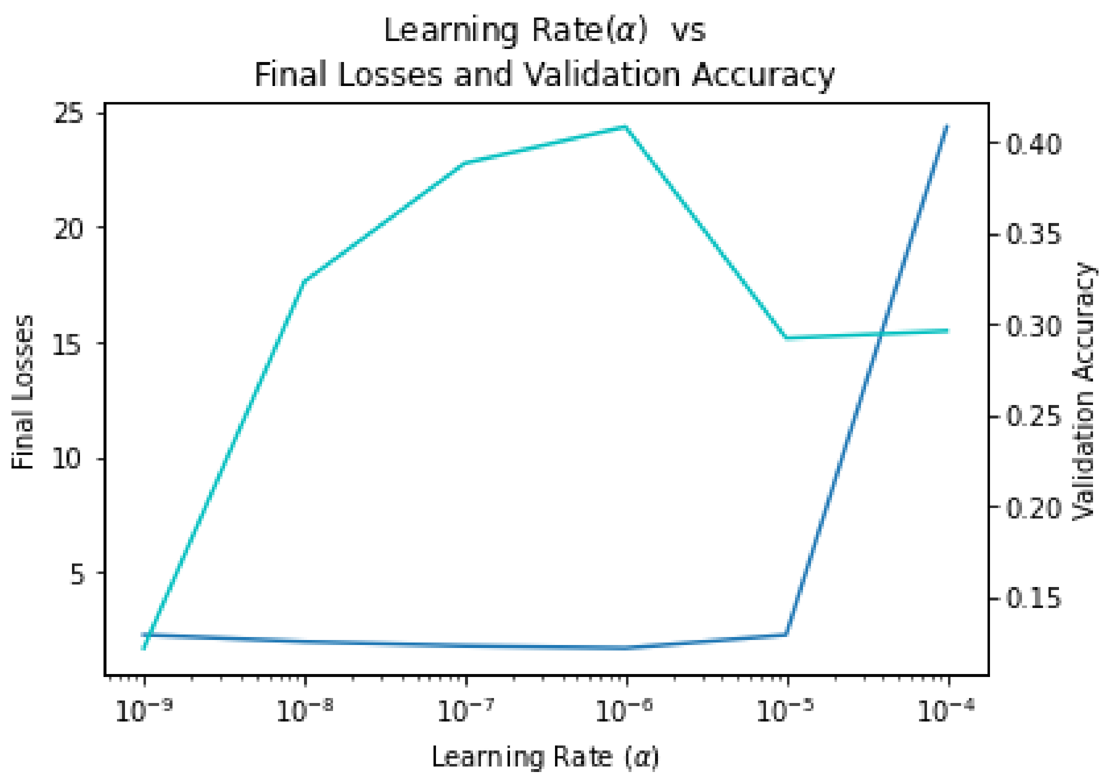
[12]:
```
2.220446049250313e-16
```

[14]:

```
# ================================================================ #
# YOUR CODE HERE:
#   Train the Softmax classifier with different learning rates and
#     evaluate on the validation data.
#   Report:
#     - The best learning rate of the ones you tested.
#     - The best validation accuracy corresponding to the best validation error.
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
# ================================================================ #
final_losses = []
val_accuracy = []
alphas = [1e-9, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4]

for i,ALPHA in enumerate([1e-9, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4]):
    loss_hist =  softmax.train(X_train, y_train,␣
↪learning_rate=ALPHA,num_iters=1500, verbose=False)
    print('learning rate: {}, final loss {:.5f}'.format(ALPHA,loss_hist[-1]))
    y_val_pred = softmax.predict(X_val)
    final_losses.append(loss_hist[-1])
    val_accuracy.append(np.mean(np.equal(y_val, y_val_pred)))
    print('validation accuracy: {:.5f}'.format(val_accuracy[i]))

fig, ax1 = plt.subplots()
ax2 = ax1.twinx()
ax1.set_title(r'Learning Rate($\alpha$)  vs ' + '\nFinal Losses and Validation␣
↪Accuracy')
ax1.set_xlabel(r'Learning Rate ($\alpha$)')
ax1.set_ylabel('Final Losses')
ax2.set_ylabel('Validation Accuracy')
ax1.plot(alphas,final_losses)
ax2.plot(alphas,val_accuracy,'c')
ax1.set_xscale('log')

print('The best learning rate by validation accruacy is {}'.format(alphas[np.
↪argmax(val_accuracy)]))
print('The validation accruacy for {}is {}'.format(alphas[np.
↪argmax(val_accuracy)],np.max(val_accuracy)))

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
learning rate: 1e-09, final loss 2.27561
validation accuracy: 0.12200
learning rate: 1e-08, final loss 1.97698
```

```
validation accuracy: 0.32300
learning rate: 1e-07, final loss 1.80075
validation accuracy: 0.38800
learning rate: 1e-06, final loss 1.70058
validation accuracy: 0.40800
learning rate: 1e-05, final loss 2.27160
validation accuracy: 0.29200
learning rate: 0.0001, final loss 24.34482
validation accuracy: 0.29600
The best learning rate by validation accruacy is 1e-06
The validation accruacy for 1e-06is 0.408
```



0.10 Answer:

Best **tested** learning rate is $1 \times e^{-6}$ with a validation accuracy of 0.408

[ ]: