softmax_nosol

January 29, 2023

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.

[134]:
```
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
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

[135]:
```
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 = (
        "/Users/mylesthemonster/Documents/ece_c247/hw2/hw2_code/
 ↪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
    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.

[136]:
```
from nndl import Softmax
```

[137]:
```
# 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

[138]:
```
## Implement the loss function of the softmax using a for loop over
#  the number of examples

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

[139]:
```
print(loss)
```

```
2.327760702804897
```

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:

The loss from the softmax being 2.327760702804897 makes sense because the softmax is a multiclass classifier. The loss is the average of the loss for each class. Since there are 10 classes, the loss for each class is $\frac{2.3}{10} = 0.23$. The loss for each class is the negative log of the probability of the correct class. Since the probability of the correct class is $\frac{1}{10}$, the loss for each class is $-\log(\frac{1}{10}) = 2.3$.

Softmax gradient

[140]:
```
## 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: -1.265198 analytic: -1.265198, relative error: 2.288884e-08
numerical: 0.118820 analytic: 0.118820, relative error: 1.340117e-07
numerical: 0.345094 analytic: 0.345094, relative error: 8.097804e-09
numerical: 1.080327 analytic: 1.080327, relative error: 1.824787e-08
numerical: -1.255915 analytic: -1.255915, relative error: 4.078140e-08
numerical: 1.819504 analytic: 1.819504, relative error: 6.604489e-09
numerical: -0.849827 analytic: -0.849827, relative error: 6.472480e-08
numerical: 0.237496 analytic: 0.237496, relative error: 2.136340e-08
numerical: 0.673580 analytic: 0.673580, relative error: 6.425449e-09
numerical: -2.647151 analytic: -2.647151, relative error: 1.810721e-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.

[141]:
```
import time
```

[142]:
```
## 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: 2.329977339069047 / 341.4968789453495 computed in
0.00615382194519043s
Vectorized loss / grad: 2.3299773550631815 / 341.4968789453495 computed in
0.0028061866760253906s
difference in loss / grad: -1.5994134461294607e-08 /1.0698549924945794e-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).

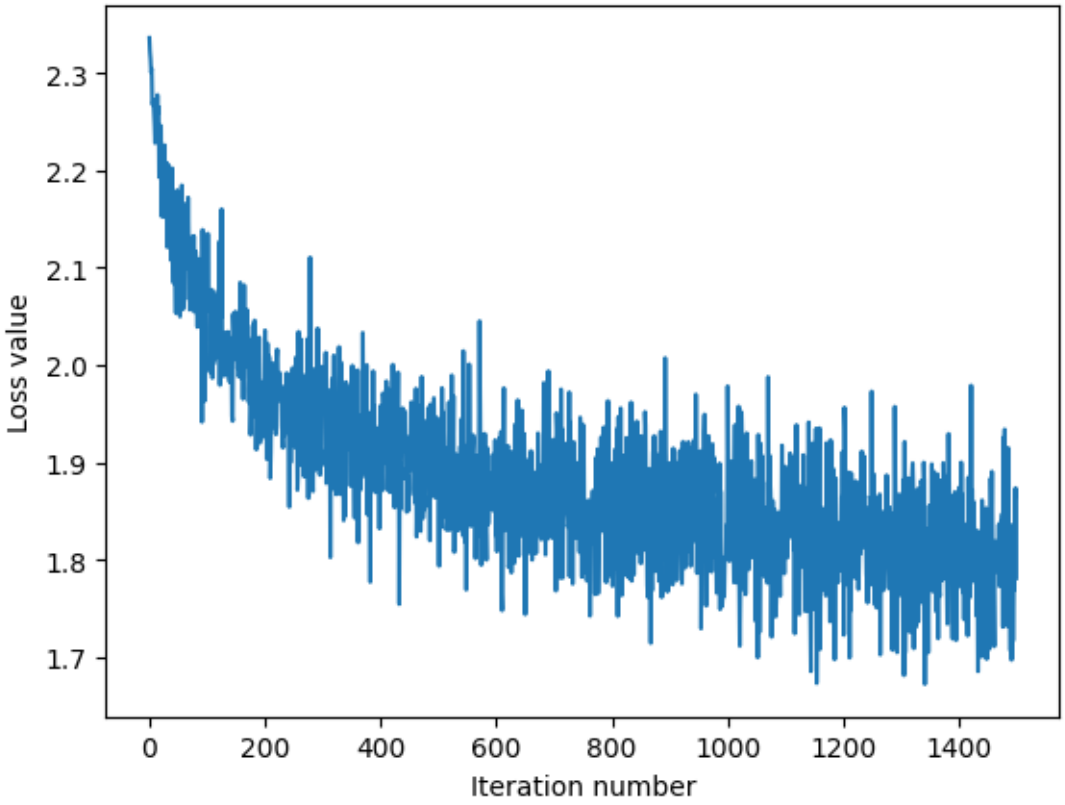
[143]:
```
# Implement softmax.train() by filling in the code to extract a batch of data
# and perform the gradient step.
import time


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 2.3365926768353664
iteration 100 / 1500: loss 2.055722282615281
iteration 200 / 1500: loss 2.0357745361814166
iteration 300 / 1500: loss 1.9813348420788213
iteration 400 / 1500: loss 1.9583142707532786
iteration 500 / 1500: loss 1.8622653355158354
iteration 600 / 1500: loss 1.8532611744112568
iteration 700 / 1500: loss 1.8353062499718755
iteration 800 / 1500: loss 1.829389275758732
iteration 900 / 1500: loss 1.8992158822347505
iteration 1000 / 1500: loss 1.9783503842474557
iteration 1100 / 1500: loss 1.8470798201432712
iteration 1200 / 1500: loss 1.8411450584382825
iteration 1300 / 1500: loss 1.7910402816542166
iteration 1400 / 1500: loss 1.8705803352042043
```

```
That took 1.916717767715454s
```



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

[144]:
```
## 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
```

0.7 Optimize the softmax classifier

[145]:
```
np.finfo(float).eps
```

[145]:
```
2.220446049250313e-16
```

[146]:
```
# ================================================================ #
# 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.
# ================================================================ #

# Define a list of learning rates to test
learning_rates = [10**i for i in range(-15, -1)]

# Initialize variables to keep track of the best learning rate and its␣
 ↪corresponding accuracy
best_rate, best_val_accuracy, best_test_accuracy = 0, 0, 0

# Iterate over all learning rates
for rate in learning_rates:

    # Train the classifier with the current learning rate
    softmax.train(X_train, y_train, learning_rate=rate, num_iters=1500,␣
 ↪verbose=False)

    # Predict the labels of the validation set
    y_val_pred = softmax.predict(X_val)

    # Calculate the accuracy of the predictions
    val_accuracy = np.mean(np.equal(y_val, y_val_pred))

    # Print the current learning rate and its corresponding accuracy
    print(f"Learning rate: {rate}, Validation accuracy: {val_accuracy}")
```

```
    # If the current accuracy is better than the best accuracy so far, update␣
 ↪the best accuracy
    if val_accuracy > best_val_accuracy:
        best_rate = rate
        best_val_accuracy = val_accuracy

# Re-train the classifier with the best learning rate
softmax.train(X_train, y_train, learning_rate=best_rate, num_iters=1500,␣
 ↪verbose=False)

# Predict the labels of the test set
y_test_pred = softmax.predict(X_test)

# Calculate the accuracy of the predictions
best_test_accuracy = np.mean(np.equal(y_test, y_test_pred))

# Print the best learning rate, test accuracy and test error rate
print(
    f"\nThe Best learning rate is {best_rate} which has a Test accuracy of␣
 ↪{best_test_accuracy}, and a Test error rate of {1.0 - best_test_accuracy}"
)

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

```
Learning rate: 1e-15, Validation accuracy: 0.122
Learning rate: 1e-14, Validation accuracy: 0.049
Learning rate: 1e-13, Validation accuracy: 0.068
Learning rate: 1e-12, Validation accuracy: 0.075
Learning rate: 1e-11, Validation accuracy: 0.083
Learning rate: 1e-10, Validation accuracy: 0.08
Learning rate: 1e-09, Validation accuracy: 0.178
Learning rate: 1e-08, Validation accuracy: 0.313
Learning rate: 1e-07, Validation accuracy: 0.392
Learning rate: 1e-06, Validation accuracy: 0.415
Learning rate: 1e-05, Validation accuracy: 0.317
Learning rate: 0.0001, Validation accuracy: 0.278
Learning rate: 0.001, Validation accuracy: 0.305
Learning rate: 0.01, Validation accuracy: 0.243

The Best learning rate is 1e-06 which has a Test accuracy of 0.399, and a Test
error rate of 0.601
```