ECE 147 HW 2 Andrea Boretta

1) a. $\tilde{\mathcal{L}}(\theta) = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - (x^{(i)} + \delta^{(i)})^T\theta)^2$

$= \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - x^{(i)T}\theta + \delta^{(i)T}\theta)^2$

$= \frac{1}{N}\sum_{i=1}^{N}[(y^{(i)} - x^{(i)T}\theta) + \delta^{(i)T}\theta]^2$

$= \frac{1}{N}\sum_{i=1}^{N}[(y^{(i)} - x^{(i)T}\theta)^2 + 2\delta^{(i)T}\theta(y^{(i)} - x^{(i)T}\theta)$
$+ \theta^T\delta^{(i)}\delta^{(i)T}\theta]$

Then, by the linearity of the expectation function,

$E[\tilde{\mathcal{L}}(\theta)] = \frac{1}{N}\sum_{i=1}^{N}(E[(y^{(i)} - x^{(i)T}\theta)^2] + E[2\delta^{(i)T}\theta(y^{(i)} - x^{(i)T}\theta)]$
$+ E[\theta^T\delta^{(i)}\delta^{(i)T}\theta])$

Since θ, $y^{(i)}$, $x^{(i)}$ are independent of $\delta^{(i)}$,

$= \frac{1}{N}\sum_{i=1}^{N}((y^{(i)} - x^{(i)T}\theta)^2 + 2E[\delta^{(i)T}]\theta(y^{(i)} - x^{(i)T}\theta)$
$+ \theta^T E[\delta^{(i)}\delta^{(i)T}]\theta)$

For a zero centered Gaussian distribution, $E[\delta^{(i)}] = 0$, and we know $E[\delta^{(i)}\delta^{(i)T}] = \sigma^2 I$, so

$= \mathcal{L}(\theta) + \frac{1}{N}\sum_{i=1}^{N}(2 \cdot 0 \cdot \theta(y^{(i)} - x^{(i)T}\theta) + \theta^T(\sigma^2 I)\theta)$

$= \mathcal{L}(\theta) + \sigma^2\theta^T\theta$

$= \mathcal{L}(\theta) + \sigma^2\|\theta\|^2$

b. The addition of noise would regularize by attempting to also minimize the magnitude of θ.

c. If $\sigma \to 0$, $\tilde{\mathcal{L}}(\theta) = \mathcal{L}(\theta) + \sigma^2\|\theta\|^2 \to \mathcal{L}(\theta)$, so there would be no regularization effect

d. if $\sigma \to \infty$, $\tilde{\mathcal{L}}(\theta) = \mathcal{L}(\theta) + \sigma^2\|\theta\|^2 \to \infty$, so it would be impossible to minimize the cost function

3) We have that

$$\text{softmax}_i(x) = \frac{e^{\tilde{w}_i^T \tilde{x}}}{\sum_{j=1}^{c} e^{\tilde{w}_j^T \tilde{x}}}$$

for $\tilde{x} = \begin{bmatrix} \hat{x} \\ 1 \end{bmatrix}$, $\tilde{w} = \begin{bmatrix} w \\ b \end{bmatrix}$.

Then, let $\mathcal{L}_i$ be the gradient of the softmax for the i-th observation in the data set. Since $\mathcal{L}$ is the negative log likelihood, it is additive. Then

$$\mathcal{L}_i(\tilde{x}^{(i)}) = -\log \text{softmax}_{y^{(i)}}(\tilde{x}^{(i)})$$

$$= \log \sum_{j=1}^{c} e^{\tilde{w}_j^T \tilde{x}^{(i)}} - \log e^{\tilde{w}_{y^{(i)}}^T \tilde{x}^{(i)}}$$

$$= \log \sum_{j=1}^{c} e^{\tilde{w}_j^T \tilde{x}^{(i)}} - \tilde{w}_{y^{(i)}}^T \tilde{x}^{(i)}$$

Then

$$\frac{\partial \mathcal{L}_i}{\partial w_k} = \frac{1}{\sum_{j=1}^{c} e^{\tilde{w}_j^T \tilde{x}}} e^{\tilde{w}_k^T \tilde{x}^{(i)}} \tilde{x}^{(i)} - \frac{\partial}{\partial w_k}\left(\tilde{w}_{y^{(i)}}^T \tilde{x}^{(i)}\right)$$

$$= \text{softmax}_k(\tilde{x}^{(i)})\, \tilde{x}^{(i)} - \mathbb{1}_{\{y^{(i)}=k\}} \tilde{x}^{(i)}$$

$$= \left(\text{softmax}_k(\tilde{x}^{(i)}) - \mathbb{1}_{\{y^{(i)}=k\}}\right) \tilde{x}^{(i)}$$

And

$$\frac{\partial \mathcal{L}}{\partial w_k} = \sum_{i=1}^{n} \frac{\partial \mathcal{L}_i}{\partial w_k}$$

4) $\mathcal{L}(w, b) = \frac{1}{k} \sum_{i=1}^{k} \text{hinge}_{y^{(i)}}(x^{(i)})$

$= \frac{1}{k} \sum_{i=1}^{k} \max(0, 1 - y^{(i)}(w^T x^{(i)} + b))$

$\frac{\partial \mathcal{L}}{\partial w} = \frac{1}{k} \sum_{i=1}^{k} \mathbb{1}_{\{y^{(i)}(w^T x^{(i)} + b) < 1\}} \left(\frac{\partial}{\partial w} (1 - y^{(i)}(w^T x^{(i)} + b))\right)$

$= \frac{1}{k} \sum \mathbb{1}_{\{y^{(i)}(w^T x^{(i)} + b) < 1\}} (- y^{(i)} x^{(i)})$

$= \frac{1}{k} X^T \begin{bmatrix} \mathbb{1}_{\{y^{(1)}(w^T x^{(1)} + b)} & \ddots & 0 \\ 0 & & \mathbb{1}_{\{y^{(k)}(w^T x^{(k)} + b)} \end{bmatrix} Y$

Where $X = \begin{bmatrix} — & x^{(1)} & — \\ & \vdots & \\ — & x^{(k)} & — \end{bmatrix}$ and $Y = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(k} \end{bmatrix}$

Also,

$\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{k} \sum_{i=1}^{k} \mathbb{1}_{\{y^{(i)}(w^T x^{(i)} + b) < 1\}} \left(\frac{\partial}{\partial b} (1 - y^{(i)}(w^T x^{(i)} + b))\right)$

$= \frac{1}{k} \sum_{i=1}^{k} \mathbb{1}_{\{y^{(i)}(w^T x^{(i)} + b) < 1\}} (- y^{(i)})$

$= - \frac{1}{k} \sum_{i=1}^{k} \mathbb{1}_{\{y^{(i)}(w^T x^{(i)} + b) < 1\}} y^{(i)}$

knn_nosol

January 30, 2024

0.1 This is the k-nearest neighbors workbook for ECE C147/C247 Assignment #2

Please follow the notebook linearly to implement k-nearest neighbors.

Please print out the workbook entirely when completed.

The goal of this workbook is to give you experience with the data, training and evaluating a simple classifier, k-fold cross validation, and as a Python refresher.

0.2 Import the appropriate libraries

[1]:
```
import numpy as np # for doing most of our calculations
import matplotlib.pyplot as plt# for plotting
from utils.data_utils import load_CIFAR10 # function to load the CIFAR-10␣
 ↪dataset.

# Load matplotlib images inline
%matplotlib inline

# These are important for reloading any code you write in external .py files.
# see http://stackoverflow.com/questions/1907993/
 ↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

[2]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = '/home/andrea/git/UCLA/UCLA_ECE147/Homework2/HW2_code/
 ↪cifar-10-batches-py' # You need to update this line
X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

# As a sanity check, we print out the size of the training and test data.
print('Training data shape: ', X_train.shape)
print('Training labels shape: ', y_train.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
```

```
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

[3]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```

[4]:
```
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]
```

```
num_test = 500
mask = list(range(num_test))
X_test = X_test[mask]
y_test = y_test[mask]

# Reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
print(X_train.shape, X_test.shape)
```

```
(5000, 3072) (500, 3072)
```

1 K-nearest neighbors

In the following cells, you will build a KNN classifier and choose hyperparameters via k-fold cross-validation.

[5]:
```
# Import the KNN class

from nndl import KNN
```

[6]:
```
# Declare an instance of the knn class.
knn = KNN()

# Train the classifier.
#   We have implemented the training of the KNN classifier.
#   Look at the train function in the KNN class to see what this does.
knn.train(X=X_train, y=y_train)
```

1.1 Questions

(1) Describe what is going on in the function knn.train().

(2) What are the pros and cons of this training step?

1.2 Answers

(1) knn.train() is loading the data into the KNN object.

(2) The pros are that it is a very fast, O(1) training step. The cons are that no processing is actually done, and no additional structure or information is gained by the training step. Also, it is very memory intensive, since a lot of data needs to be stored.

1.3 KNN prediction

In the following sections, you will implement the functions to calculate the distances of test points to training points, and from this information, predict the class of the KNN.

[7]:
```
# Implement the function compute_distances() in the KNN class.
# Do not worry about the input 'norm' for now; use the default definition of␣
↪the norm
# in the code, which is the 2-norm.
# You should only have to fill out the clearly marked sections.

import time
time_start =time.time()

dists_L2 = knn.compute_distances(X=X_test)

print('Time to run code: {}'.format(time.time()-time_start))
print('Frobenius norm of L2 distances: {}'.format(np.linalg.norm(dists_L2,␣
↪'fro')))
```

```
Time to run code: 22.351867198944092
Frobenius norm of L2 distances: 7906696.077040902
```

Really slow code Note: This probably took a while. This is because we use two for loops. We could increase the speed via vectorization, removing the for loops.

If you implemented this correctly, evaluating np.linalg.norm(dists_L2, 'fro') should return: ~7906696

1.3.1 KNN vectorization

The above code took far too long to run. If we wanted to optimize hyperparameters, it would be time-expensive. Thus, we will speed up the code by vectorizing it, removing the for loops.

[8]:
```
# Implement the function compute_L2_distances_vectorized() in the KNN class.
# In this function, you ought to achieve the same L2 distance but WITHOUT any␣
↪for loops.
# Note, this is SPECIFIC for the L2 norm.

time_start =time.time()
dists_L2_vectorized = knn.compute_L2_distances_vectorized(X=X_test)
print('Time to run code: {}'.format(time.time()-time_start))
print('Difference in L2 distances between your KNN implementations (should be␣
↪0): {}'.format(np.linalg.norm(dists_L2 - dists_L2_vectorized, 'fro')))
```

```
Time to run code: 0.19461321830749512
Difference in L2 distances between your KNN implementations (should be 0): 0.0
```

Speedup Depending on your computer speed, you should see a 10-100x speed up from vectorization. On our computer, the vectorized form took 0.36 seconds while the naive implementation took 38.3 seconds.

1.3.2 Implementing the prediction

Now that we have functions to calculate the distances from a test point to given training points, we now implement the function that will predict the test point labels.

[9]:
```
# Implement the function predict_labels in the KNN class.
# Calculate the training error (num_incorrect / total_samples)
#   from running knn.predict_labels with k=1

error = 1

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the error rate by calling predict_labels on the test
#   data with k = 1.  Store the error rate in the variable error.
# ================================================================ #
predictions = knn.predict_labels(dists_L2_vectorized, k=1)
error = (predictions != y_test).mean()

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

print(error)
```

```
0.726
```

If you implemented this correctly, the error should be: 0.726.

This means that the k-nearest neighbors classifier is right 27.4% of the time, which is not great, considering that chance levels are 10%.

2 Optimizing KNN hyperparameters

In this section, we'll take the KNN classifier that you have constructed and perform cross-validation to choose a best value of k, as well as a best choice of norm.

2.0.1 Create training and validation folds

First, we will create the training and validation folds for use in k-fold cross validation.

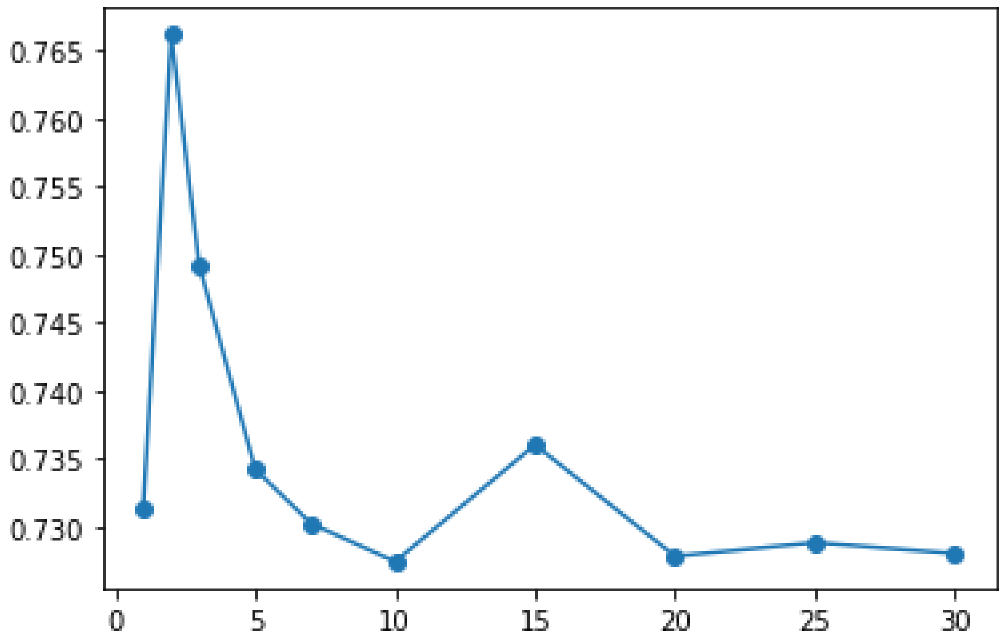
[10]:
```
# Create the dataset folds for cross-valdiation.
num_folds = 5

X_train_folds = []
y_train_folds =  []

# ================================================================ #
# YOUR CODE HERE:
#   Split the training data into num_folds (i.e., 5) folds.
#   X_train_folds is a list, where X_train_folds[i] contains the
#      data points in fold i.
#   y_train_folds is also a list, where y_train_folds[i] contains
#      the corresponding labels for the data in X_train_folds[i]
# ================================================================ #

split = np.arange(5000)
np.random.shuffle(split)
split_mat = (split[:,np.newaxis] % num_folds == np.arange(5))

for i in range(0, num_folds):
    X_train_folds.append(X_train[split_mat[:,i]])
    y_train_folds.append(y_train[split_mat[:,i]])

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

2.0.2 Optimizing the number of nearest neighbors hyperparameter.

In this section, we select different numbers of nearest neighbors and assess which one has the lowest k-fold cross validation error.

[12]:
```
%matplotlib inline
import matplotlib.pyplot as plt
time_start =time.time()

ks = [1, 2, 3, 5, 7, 10, 15, 20, 25, 30]

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the cross-validation error for each k in ks, testing
#   the trained model on each of the 5 folds.  Average these errors
#   together and make a plot of k vs. cross-validation error. Since
#   we are assuming L2 distance here, please use the vectorized code!
#   Otherwise, you might be waiting a long time.
# ================================================================ #

err_data = np.zeros((len(ks), num_folds))
for i in range(0, num_folds):
    knn2 = KNN()
    knn2.train(X_train[~split_mat[:,i]], y_train[~split_mat[:,i]])
    dists = knn2.compute_L2_distances_vectorized(X_train_folds[i])
    # print("i-th fold:", X_train_folds[i].shape)
    # print("dists.shape:", dists.shape)
    # print(np.unique(dists, return_counts=True))
    for j, k in enumerate(ks):
        predictions = knn2.predict_labels(dists, k=k)
        error = (predictions != y_train_folds[i]).mean()
        # print(error)
        # print(predictions)
        # print(y_train_folds[i])
        err_data[j, i] = error

# print(err_data)
mean_err = err_data.mean(axis=1)
print(mean_err)
plt.plot(ks, err_data.mean(axis=1), 'o-')


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

print('Computation time: %.2f'%(time.time()-time_start))
```

```
[0.7314 0.7662 0.7492 0.7342 0.7302 0.7274 0.736  0.7278 0.7288 0.728 ]
Computation time: 4.12
```



2.1 Questions:

(1) What value of k is best amongst the tested k's?

(2) What is the cross-validation error for this value of k?

2.2 Answers:

(1) The best value of k was 10,

(2) The cross validation error for k = 10 was 0.7274.

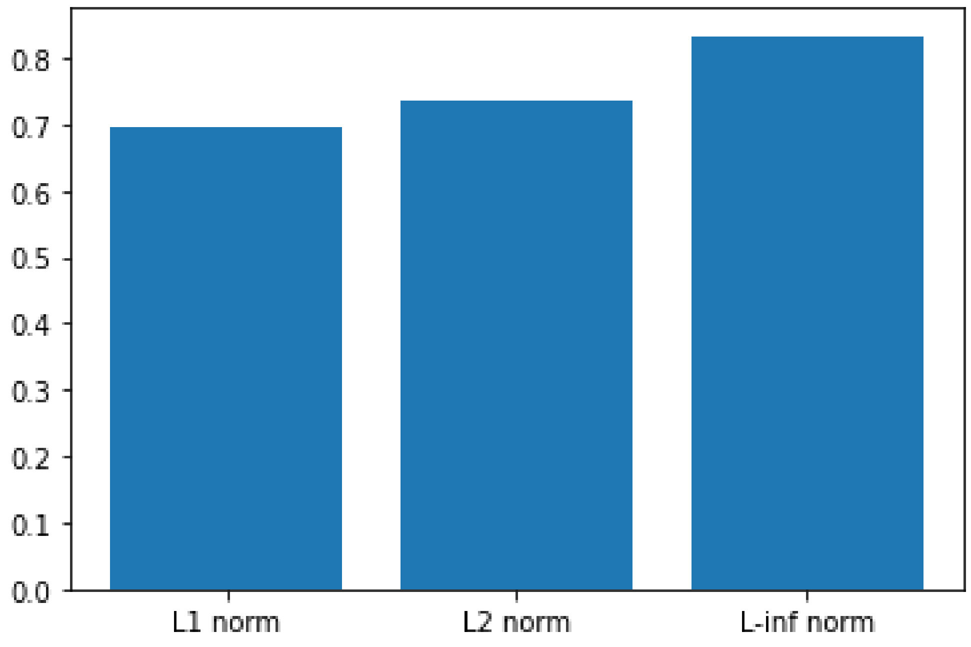
2.2.1 Optimizing the norm

Next, we test three different norms (the 1, 2, and infinity norms) and see which distance metric results in the best cross-validation performance.

[13]:
```
time_start =time.time()

L1_norm = lambda x: np.linalg.norm(x, ord=1)
L2_norm = lambda x: np.linalg.norm(x, ord=2)
Linf_norm = lambda x: np.linalg.norm(x, ord= np.inf)
norms = [L1_norm, L2_norm, Linf_norm]

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the cross-validation error for each norm in norms, testing
#   the trained model on each of the 5 folds.  Average these errors
#   together and make a plot of the norm used vs the cross-validation error
#   Use the best cross-validation k from the previous part.
#
#   Feel free to use the compute_distances function.  We're testing just
#   three norms, but be advised that this could still take some time.
#   You're welcome to write a vectorized form of the L1- and Linf- norms
#   to speed this up, but it is not necessary.
# ================================================================ #

err_data = np.zeros((len(norms), num_folds))
for i in range(0, num_folds):
    knn2 = KNN()
    knn2.train(X_train[~split_mat[:,i]], y_train[~split_mat[:,i]])
    for j, norm in enumerate(norms):
        dists = knn2.compute_distances(X_train_folds[i], norm)
        predictions = knn2.predict_labels(dists, k=15)
        error = (predictions != y_train_folds[i]).mean()
        # print(error)
        # print(predictions)
        # print(y_train_folds[i])
        err_data[j, i] = error

# print(err_data)
mean_err = err_data.mean(axis=1)
print(mean_err)
plt.bar(["L1 norm", "L2 norm", "L-inf norm"], mean_err)
```

```
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
print('Computation time: %.2f'%(time.time()-time_start))
```

```
[0.6944 0.736  0.834 ]
Computation time: 525.52
```



2.3 Questions:

(1) What norm has the best cross-validation error?

(2) What is the cross-validation error for your given norm and k?

2.4 Answers:

(1) The L1 norm had the best cross-validation error for k=10.

(2) The cross validation error it achieved is 0.6944.

3 Evaluating the model on the testing dataset.

Now, given the optimal k and norm you found in earlier parts, evaluate the testing error of the k-nearest neighbors model.

[27]:
```
error = 1
```

```
# ================================================================ #
# YOUR CODE HERE:
#   Evaluate the testing error of the k-nearest neighbors classifier
#   for your optimal hyperparameters found by 5-fold cross-validation.
# ================================================================ #

dists = knn.compute_distances(X_test, norm=L1_norm)
predictions = knn.predict_labels(dists, k=10)
error = (predictions != y_test).mean()

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

print('Error rate achieved: {}'.format(error))
```

```
Error rate achieved: 0.722
```

3.1 Question:

How much did your error improve by cross-validation over naively choosing $k = 1$ and using the L2-norm?

3.2 Answer:

Very little. The error rate fell from 0.726 to 0.722, a whopping 1.5% improvement in success rate.

```
import numpy as np
import pdb


class KNN(object):

  def __init__(self):
    pass

  def train(self, X, y):
    """
    Inputs:
    - X is a numpy array of size (num_examples, D)
    - y is a numpy array of size (num_examples, )
    """
    self.X_train = X
    self.y_train = y

  def compute_distances(self, X, norm=None):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train.

    Inputs:
    - X: A numpy array of shape (num_test, D) containing test data.
    - norm: the function with which the norm is taken.

    Returns:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
      is the Euclidean distance between the ith test point and the jth training
      point.
    """
    if norm is None:
      norm = lambda x: np.sqrt(np.sum(x**2))
      #norm = 2

    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    # print("X.shape:", X.shape)
    # print("X_train.shape:", self.X_train.shape)

    for i in np.arange(num_test):

      for j in np.arange(num_train):
        # ================================================================ #
        # YOUR CODE HERE:
        #   Compute the distance between the ith test point and the jth
        #   training point using norm(), and store the result in dists[i, j].
        # ================================================================ #

        dists[i, j] = norm(X[i] - self.X_train[j])

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

    return dists

  # def compute_Ln_distances_vectorized(self, X, norm=None):
  #   if norm is None:
  #     norm = lambda x: np.sqrt(np.sum(x**2))
  #     #norm = 2
  #   num_test = X.shape[0]
  #   num_train = self.X_train.shape[0]

  #   dists = norm(X[:,np.newaxis,:] - self.X_train)

  #   return dists

  def compute_L2_distances_vectorized(self, X):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train WITHOUT using any for loops.

    Inputs:
    - X: A numpy array of shape (num_test, D) containing test data.

    Returns:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
      is the Euclidean distance between the ith test point and the jth training
      point.
    """
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))

    # ================================================================ #
    # YOUR CODE HERE:
    #   Compute the L2 distance between the ith test point and the jth
    #   training point and store the result in dists[i, j].  You may
    #   NOT use a for loop (or list comprehension).  You may only use
    #   numpy operations.
    #
    #   HINT: use broadcasting.  If you have a shape (N,1) array and
    #   a shape (M,) array, adding them together produces a shape (N, M)
    #   array.
    # ================================================================ #

    dists = np.sqrt(np.sum(np.square(X)[:,np.newaxis,:], axis=2) - 2 * X@self.X_train.T + np.s
um(np.square(self.X_train), axis=1))

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

    return dists


  def predict_labels(self, dists, k=1):
    """
    Given a matrix of distances between test points and training points,
    predict a label for each test point.

    Inputs:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
      gives the distance betwen the ith test point and the jth training point.

    Returns:
    - y: A numpy array of shape (num_test,) containing predicted labels for the
      test data, where y[i] is the predicted label for the test point X[i].
    """
    num_test = dists.shape[0]
    y_pred = np.zeros(num_test)
    for i in np.arange(num_test):
      # A list of length k storing the labels of the k nearest neighbors to
      # the ith test point.
      closest_y = []
      # ================================================================ #
      # YOUR CODE HERE:
      #   Use the distances to calculate and then store the labels of
      #   the k-nearest neighbors to the ith test point.  The function
      #   numpy.argsort may be useful.
      #
      #   After doing this, find the most common label of the k-nearest
      #   neighbors.  Store the predicted label of the ith training example
      #   as y_pred[i].  Break ties by choosing the smaller label.
      # ================================================================ #

      closest_y = self.y_train[np.argpartition(dists[i,:], k)[:k]]
      vals, counts = np.unique(closest_y, return_counts=True)
      y_pred[i] = vals[np.argmax(counts)]

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

    return y_pred
```

softmax_nosol

January 30, 2024

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
```

[107]:
```
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 = '/home/andrea/git/UCLA/UCLA_ECE147/Homework2/HW2_code/
↪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)
    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.

[108]:
```
from nndl import Softmax
```

[109]:
```
# 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

[129]:
```
## Implement the loss function of the softmax using a for loop over
#  the number of examples

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

[120]:
```
print(loss)
```

```
2.3277607028048966
```

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 computed loss is the average loss per observation. The loss is -2.3, so the softmax per observation is $e^{-2.3}$, or about $\frac{1}{10}$. This makes perfect sense since W is approximately 0, so the softmax evaluates to $\text{softmax}_j(x^i) = \frac{e^{w_j^T x}}{\sum_{k=1}^{c} e^{w_k^T x}} \approx \frac{e^0}{\sum_{k=1}^{c} e^0} = \frac{1}{c} = \frac{1}{10}$ since we have $c = 10$ classes.

Softmax gradient

[261]:
```
## 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: -0.412513 analytic: -0.412513, relative error: 3.100247e-08
numerical: 1.449105 analytic: 1.449105, relative error: 5.650975e-09
numerical: 2.770640 analytic: 2.770640, relative error: 2.425229e-09
numerical: -1.671566 analytic: -1.671566, relative error: 6.438381e-09
numerical: 2.540387 analytic: 2.540387, relative error: 9.276612e-09
numerical: 1.122858 analytic: 1.122858, relative error: 3.049509e-08
numerical: -0.832407 analytic: -0.832407, relative error: 2.164703e-08
numerical: 0.362969 analytic: 0.362969, relative error: 1.164066e-07
numerical: 0.110281 analytic: 0.110281, relative error: 2.015857e-09
numerical: 1.501076 analytic: 1.501076, relative error: 4.027766e-09
```

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.

[262]:
```
import time
```

[263]:
```
## 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.
```

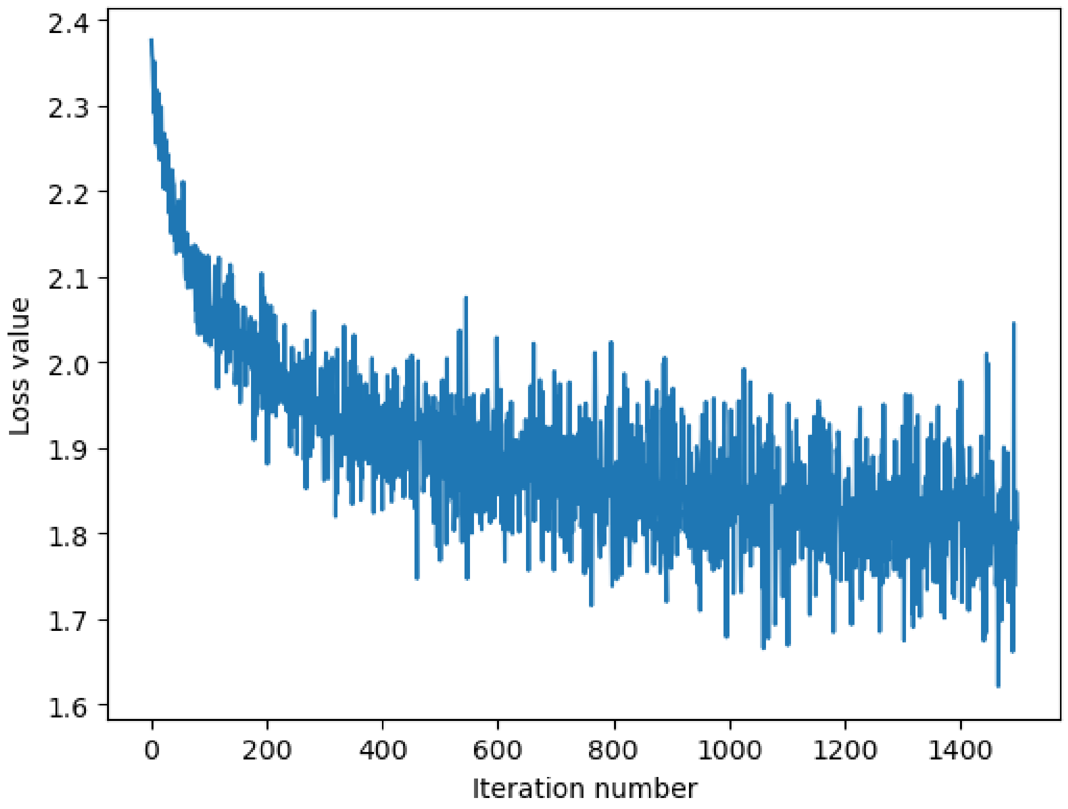
```
Normal loss / grad_norm: 2.3271336756111975 / 316.2043429344621 computed in
0.06460165977478027s
Vectorized loss / grad: 2.327133675611197 / 316.2043429344621 computed in
0.004245758056640625s
difference in loss / grad: 4.440892098500626e-16 /2.359453427692796e-13
```

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).

[269]:
```
# 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 2.3762751114287943
iteration 100 / 1500: loss 2.076086118619406
iteration 200 / 1500: loss 2.067041703893229
iteration 300 / 1500: loss 1.9796769840619515
iteration 400 / 1500: loss 1.8270526124815358
iteration 500 / 1500: loss 1.7680983550326121
iteration 600 / 1500: loss 1.8455566105812704
iteration 700 / 1500: loss 1.960219026122504
iteration 800 / 1500: loss 1.8611293417615247
iteration 900 / 1500: loss 1.7590826893658758
iteration 1000 / 1500: loss 1.821829237861987
iteration 1100 / 1500: loss 1.8012093950570955
iteration 1200 / 1500: loss 1.810574438016967
iteration 1300 / 1500: loss 1.924711064381258
iteration 1400 / 1500: loss 1.8759793554305062
That took 6.072973012924194s
```



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

[270]:
```
## 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: 0.3793061224489796
validation accuracy: 0.384
```

0.7 Optimize the softmax classifier

[276]:
```
# ================================================================ #
# 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.
# ================================================================ #

learning_rates = [1e-9, 1e-8, 1e-7, 1e-6, 1e-5]
W = np.random.normal(size=[num_classes, num_features]) * 0.0001
results = np.zeros((len(learning_rates), 2))

for i, e in enumerate(learning_rates):
    softmax.W = W
    loss_history = softmax.train(X_train, y_train, learning_rate=e,
                      num_iters=1500, verbose=False)

    y_train_pred = softmax.predict(X_train)
    print('training accuracy for learning rate {}: {}'.format(e, 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)), ))
    results[i][0] = np.mean(np.equal(y_val, y_val_pred))
    results[i][1] = loss_history[-1]

best_rate_idx = np.argmin(results[:,1])
print("The best learning rate was:", learning_rates[best_rate_idx])
print("It achieved a final loss of:", results[best_rate_idx][1])
print("The corresponding validation accuracy of:", results[best_rate_idx][0])
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
training accuracy for learning rate 1e-09: 0.17146938775510204
validation accuracy: 0.166
training accuracy for learning rate 1e-08: 0.2841836734693878
validation accuracy: 0.301
training accuracy for learning rate 1e-07: 0.38124489795918365
validation accuracy: 0.378
training accuracy for learning rate 1e-06: 0.4179795918367347
validation accuracy: 0.4
training accuracy for learning rate 1e-05: 0.3409387755102041
validation accuracy: 0.322
The best learning rate was: 1e-06
It achieved a final loss of: 1.6598466343248017
The corresponding validation accuracy of: 0.4
```

```
import numpy as np


class Softmax(object):

  def __init__(self, dims=[10, 3073]):
    self.init_weights(dims=dims)

  def init_weights(self, dims):
    """
    Initializes the weight matrix of the Softmax classifier.
    Note that it has shape (C, D) where C is the number of
    classes and D is the feature size.
    """
    self.W = np.random.normal(size=dims) * 0.0001

  def loss(self, X, y):
    """
    Calculates the softmax loss.

    Inputs have dimension D, there are C classes, and we operate on minibatches
    of N examples.

    Inputs:
    - X: A numpy array of shape (N, D) containing a minibatch of data.
    - y: A numpy array of shape (N,) containing training labels; y[i] = c means
      that X[i] has label c, where 0 <= c < C.

    Returns a tuple of:
    - loss as single float
    """

    # Initialize the loss to zero.
    loss = 0.0

    unit_loss = np.zeros(X.shape[0])
    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the normalized softmax loss.  Store it as the variable loss.
    #   (That is, calculate the sum of the losses of all the training
    #   set margins, and then normalize the loss by the number of
    #   training examples.)
    # ================================================================ #
    assert X.shape[1] == self.W.shape[1], f"{X.shape[1] =} != {self.W.shape[1] =}"
    def softmax(c, x):
        return np.exp(self.W[c]@x)/(np.exp(self.W@x)).sum()

#     i = 0
    for y_i, x_i in zip(y, X):
        loss -= np.log(softmax(y_i, x_i))
#         unit_loss[i] = -np.log(softmax(y_i, x_i))
#         i += 1

    loss /= X.shape[0]
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

#     return loss, unit_loss
    return loss

  def loss_and_grad(self, X, y):
    """
    Same as self.loss(X, y), except that it also returns the gradient.

    Output: grad -- a matrix of the same dimensions as W containing
      the gradient of the loss with respect to W.
    """

    # Initialize the loss and gradient to zero.
    loss = 0.0
    grad = np.zeros_like(self.W)

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the softmax loss and the gradient. Store the gradient
    #   as the variable grad.
    # ================================================================ #
    assert X.shape[1] == self.W.shape[1], f"{X.shape[1] =} != {self.W.shape[1] =}"
    def softmax(c, x):
        return np.exp(self.W[c]@x)/(np.exp(self.W@x)).sum()

    for y_i, x_i in zip(y, X):
        loss -= np.log(softmax(y_i, x_i))

    loss /= X.shape[0]


    for i in range(0, X.shape[0]):
        for k in range(0, self.W.shape[0]):
            dL_i_dw_k = (softmax(k, X[i]) - (k == y[i]))*X[i]
            assert grad[k,:].shape == dL_i_dw_k.shape, f"{grad[k,:].shape =} != {dL_i_dw_k.sha
pe =}"
            grad[k,:] += dL_i_dw_k

    grad /= X.shape[0]

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

    return loss, grad

  def grad_check_sparse(self, X, y, your_grad, num_checks=10, h=1e-5):
    """
    sample a few random elements and only return numerical
    in these dimensions.
    """

    for i in np.arange(num_checks):
      ix = tuple([np.random.randint(m) for m in self.W.shape])

      oldval = self.W[ix]
      self.W[ix] = oldval + h # increment by h
      fxph = self.loss(X, y)
      self.W[ix] = oldval - h # decrement by h
      fxmh = self.loss(X,y) # evaluate f(x - h)
      self.W[ix] = oldval # reset

      grad_numerical = (fxph - fxmh) / (2 * h)
      grad_analytic = your_grad[ix]
      rel_error = abs(grad_numerical - grad_analytic) / (abs(grad_numerical) + abs(grad_analyt
ic))
      print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical, grad_analytic,
 rel_error))
```

```
  def fast_loss_and_grad(self, X, y):
    """
    A vectorized implementation of loss_and_grad. It shares the same
    inputs and ouptuts as loss_and_grad.
    """
    loss = 0.0
    grad = np.zeros(self.W.shape) # initialize the gradient as zero

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the softmax loss and gradient WITHOUT any for loops.
    # ================================================================ #

    assert X.shape[1] == self.W.shape[1], f"{X.shape[1] =} != {self.W.shape[1] =}"
    loss = (-np.log((np.exp((self.W[y]*X).sum(axis=1))/(np.exp(X@self.W.T).sum(axis=1))))).mea
n()


    softmax_mat = ((np.exp(X@self.W.T))/(np.exp(X@self.W.T).sum(axis=1)[:,np.newaxis]))
    indicator_func = 1*(y[:,np.newaxis] == np.arange(10))
    grad = (1.0/X.shape[0])*(softmax_mat - indicator_func).T@X

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

    return loss, grad

  def train(self, X, y, learning_rate=1e-3, num_iters=100,
            batch_size=200, verbose=False):
    """
    Train this linear classifier using stochastic gradient descent.

    Inputs:
    - X: A numpy array of shape (N, D) containing training data; there are N
      training samples each of dimension D.
    - y: A numpy array of shape (N,) containing training labels; y[i] = c
      means that X[i] has label 0 <= c < C for C classes.
    - learning_rate: (float) learning rate for optimization.
    - num_iters: (integer) number of steps to take when optimizing
    - batch_size: (integer) number of training examples to use at each step.
    - verbose: (boolean) If true, print progress during optimization.

    Outputs:
    A list containing the value of the loss function at each training iteration.
    """
    num_train, dim = X.shape
    num_classes = np.max(y) + 1 # assume y takes values 0...K-1 where K is number of classes

    self.init_weights(dims=[np.max(y) + 1, X.shape[1]]) # initializes the weights of self.W

    # Run stochastic gradient descent to optimize W
    loss_history = []

    for it in np.arange(num_iters):
        X_batch = None
        y_batch = None

        # ================================================================ #
        # YOUR CODE HERE:
        #   Sample batch_size elements from the training data for use in
        #     gradient descent.  After sampling,
        #     - X_batch should have shape: (batch_size, dim)
        #     - y_batch should have shape: (batch_size,)
        #   The indices should be randomly generated to reduce correlations
        #   in the dataset.  Use np.random.choice.  It's okay to sample with
        #   replacement.
        # ================================================================ #
        idxs = np.random.choice(np.arange(num_train), size=batch_size, replace=False)
        X_batch = X[idxs]
        y_batch = y[idxs]
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

        # evaluate loss and gradient
        loss, grad = self.fast_loss_and_grad(X_batch, y_batch)
        loss_history.append(loss)

        # ================================================================ #
        # YOUR CODE HERE:
        #   Update the parameters, self.W, with a gradient step
        # ================================================================ #
        self.W -= learning_rate*grad

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

        if verbose and it % 100 == 0:
            print('iteration {} / {}: loss {}'.format(it, num_iters, loss))

    return loss_history

  def predict(self, X):
    """
    Inputs:
    - X: N x D array of training data. Each row is a D-dimensional point.

    Returns:
    - y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional
      array of length N, and each element is an integer giving the predicted
      class.
    """
    y_pred = np.zeros(X.shape[1])
    # ================================================================ #
    # YOUR CODE HERE:
    #   Predict the labels given the training data.
    # ================================================================ #
    y_pred = np.argmax(X@self.W.T, axis=1)
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return y_pred
```