knn

January 27, 2021

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.

We thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu). These are the functions in the cs231n folders and code in the jupyer notebook to preprocess and show the images. The classifiers used are based off of code prepared for CS 231n as well.

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

[8]:
```
import numpy as np # for doing most of our calculations
import matplotlib.pyplot as plt# for plotting
from cs231n.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
```

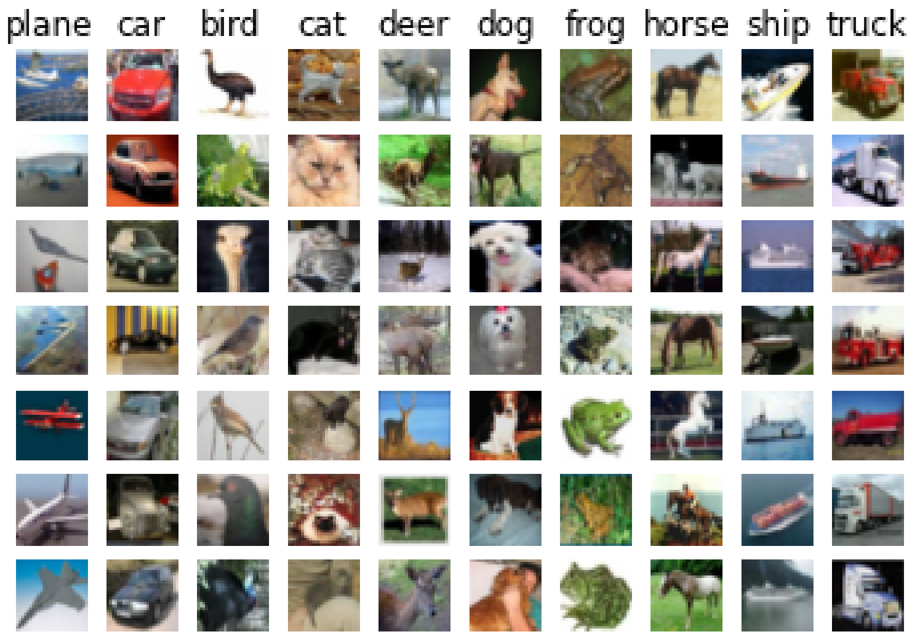
```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

[11]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = 'C:/Users/Ashwin/Desktop/UCLA/current classes/ece 247/hw2/
 ↪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,)
```

[12]:
```
# 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()
```



[13]:
```
# 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.

[14]:
```
# Import the KNN class

from nndl import KNN
```

[15]:
```
# 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) In knn.train(), we are just declaring and initializing the instance variables X_train and y_train of the knn object and caching them in memory.

(2) The pros is that it's fast, O(1), and very simple. The biggest con is that it takes a lot of memory to store all the training and test data in the object.

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.

[47]:
```
# 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: 105.75354480743408
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.

[30]:
```
# 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: 0.6139991283416748
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.

[61]:
```python
# 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



# ================================================================ #
# 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.
# ================================================================ #

predicted_labels = knn.predict_labels(dists_L2_vectorized, 1)
num_incorrect = np.sum(y_test != predicted_labels)
error = num_incorrect / num_test

# ================================================================ #
# 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.

[66]:
```
# 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]
# ================================================================ #
fold_size = num_training // num_folds
for i in range(num_folds):
    start, end = fold_size*i, fold_size*(i+1)
    X_train_folds.append(X_train[start:end])
    y_train_folds.append(y_train[start:end])

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

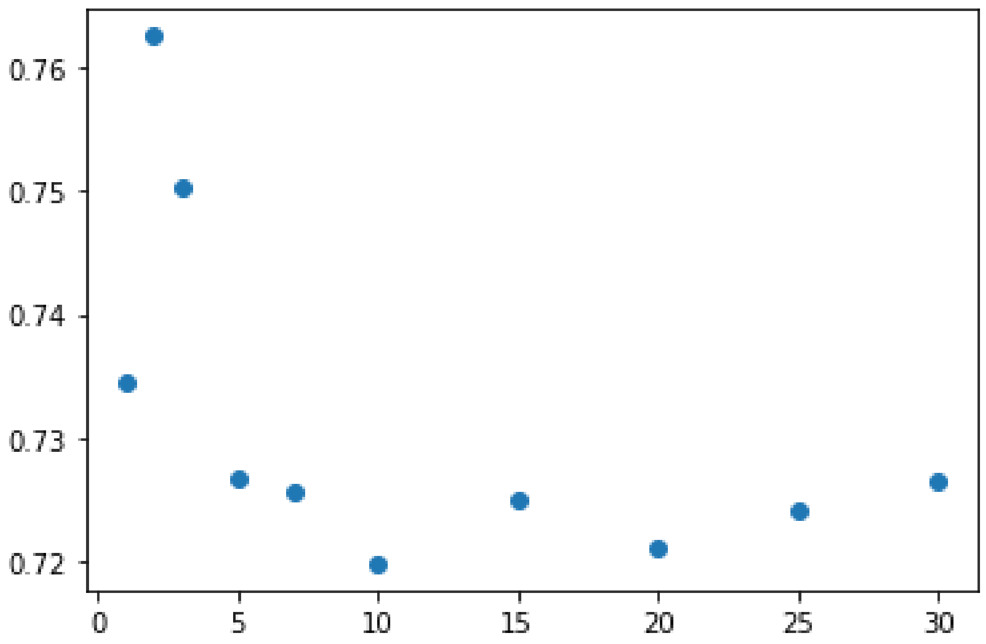
2.0.2 Optimizing the number of nearest neighbors hyperparameter.

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

[103]:
```
time_start =time.time()

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

# ================================================================ #
# 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.
# ================================================================ #

def calculate_error_per_k (k):
    errors = []

    for i, xy in enumerate(zip(X_train_folds, y_train_folds)):
        xtest, ytest = xy[0], xy[1]

        #training data for this fold = all training data - this fold
        start, end = fold_size*i, fold_size*(i+1)

        #kept getting errors when trying to use X_train and splicing, so I'll␣
 ↪just do it manually
        xtrain_list = []
        ytrain_list = []

        for j in range(num_folds):
            if i == j:
                continue
            xtrain_list.extend(X_train_folds[j])
            ytrain_list.extend(y_train_folds[j])

        #convert to numpy array since lists are not compatible
        xtrain, ytrain = np.array(xtrain_list), np.array(ytrain_list)

        #train model on the other k-1 folds
        model = KNN()
        model.train(X=xtrain, y=ytrain)

        #test model on this specific fold
        predicted_labels = model.predict_labels(model.
 ↪compute_L2_distances_vectorized(xtest), k)
        num_incorrect = np.sum(ytest != predicted_labels)
        error = num_incorrect / len(xtest)
        errors.append(error)
    return np.mean(errors)

error_per_ks = [calculate_error_per_k(x) for x in ks]

for x,y in zip(ks,error_per_ks):
    print("k:", x, "error:", y)

plt.scatter(ks, error_per_ks)
plt.show()
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

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

```
k: 1 error: 0.7344
k: 2 error: 0.7626000000000002
k: 3 error: 0.7504000000000001
```

```
k: 5 error: 0.7267999999999999
k: 7 error: 0.7256
k: 10 error: 0.7198
k: 15 error: 0.725
k: 20 error: 0.721
k: 25 error: 0.7242
k: 30 error: 0.7266
```



```
Computation time: 93.00
```

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:

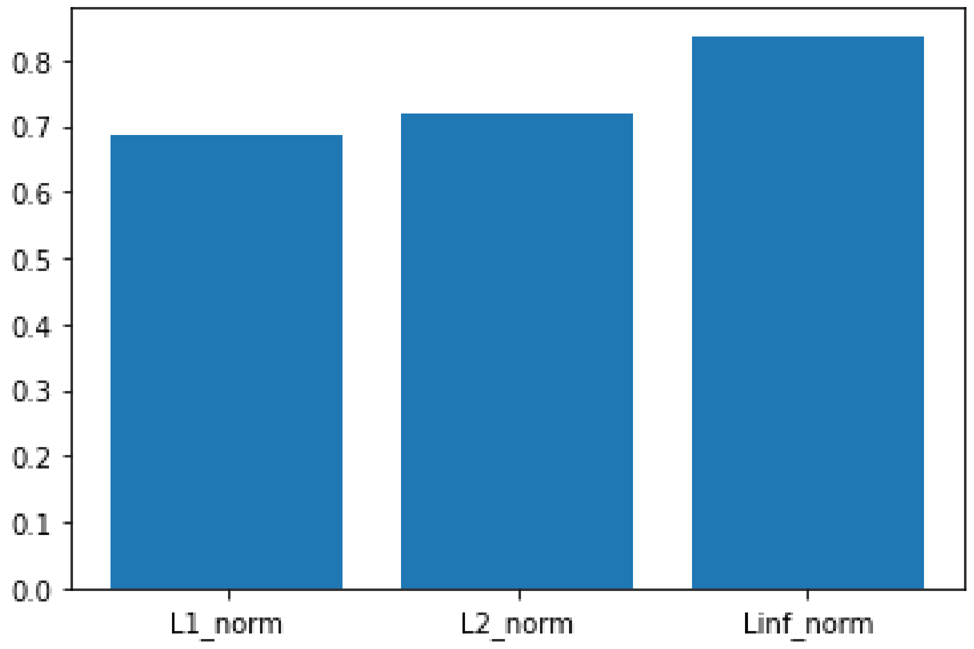
(1) Looking at the results, it looks like k = 10 is the best among the tested k values.

(2) The cross-validation error is 0.7198.

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.

```
[104]: 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.
# ================================================================ #

def calculate_error_per_k_norm (k, norm):
    errors = []

    for i, xy in enumerate(zip(X_train_folds, y_train_folds)):

        xtest, ytest = xy[0], xy[1]

        #training data for this fold = all training data - this fold
        start, end = fold_size*i, fold_size*(i+1)

        #kept getting errors when trying to use X_train and splicing, so I'll␣
 ↪just do it manually
        xtrain_list = []
        ytrain_list = []

        for j in range(num_folds):
            if i == j:
                continue
            xtrain_list.extend(X_train_folds[j])
            ytrain_list.extend(y_train_folds[j])

        #convert to numpy array since lists are not compatible
        xtrain, ytrain = np.array(xtrain_list), np.array(ytrain_list)

        #train model on the other k-1 folds
        model = KNN()
        model.train(X=xtrain, y=ytrain)
        #test model on this specific fold
        predicted_labels = model.predict_labels(model.compute_distances(xtest,␣
↪norm), k)
        num_incorrect = np.sum(ytest != predicted_labels)
        error = num_incorrect / len(xtest)
        errors.append(error)
    return np.mean(errors)

errors_per_norm = [calculate_error_per_k_norm(10,norm) for norm in norms]

plt.bar(["L1_norm", "L2_norm", "Linf_norm"], errors_per_norm)
plt.show()

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



```
Computation time: 1941.78
```

[105]:
```
print(errors_per_norm)
```

```
[0.6886000000000001, 0.7198, 0.8370000000000001]
```

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) L1 norm – hopefully this is correct b/c it took me 30 min to run oops.

(2) 0.6886000000000001

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.

[107]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Evaluate the testing error of the k-nearest neighbors classifier
#   for your optimal hyperparameters found by 5-fold cross-validation.
# ================================================================ #

dist_l1 = knn.compute_distances(X=X_test, norm = L1_norm)
predicted_l1 = knn.predict_labels(dist_l1, 10)
error = np.sum(y_test != predicted_l1) / num_test

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

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

#difference between old error and new error
print("Improvement:", 0.726 - error)
```

```
Error rate achieved: 0.722
Improvement: 0.0040000000000000036
```

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:

The error rate improved slightly by about 0.004.

3.3 KNN Notebook Code Below:

```
[ ]: import numpy as np
     import pdb
     from scipy.stats import mode

     """
     This code was based off of code from cs231n at Stanford University, and␣
      ↪modified for ECE C147/C247 at UCLA.
     """

     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))
         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])
        pass

              #␣
↪================================================================ #
              # END YOUR CODE HERE
              #␣
↪================================================================ #

    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.
        # ================================================================ #
    #let X be a N x 1 column vector, we want to expand it to a N x M matrix
    #let X_train be a 1 x M row vector, we want to expand it to a N x M matrix

    #optimized pairwise distances from here: https://www.pythonlikeyoumeanit.
↪com/Module3_IntroducingNumpy/Broadcasting.
↪html#Pairwise-Distances-Using-Broadcasting-%28Unoptimized%29

    x_squared = np.sum(X**2, axis=1)[:, np.newaxis]
    y_squared = np.sum(self.X_train**2, axis=1)
    two_x_y = 2*np.matmul(X, self.X_train.T)

    dists = np.sqrt(x_squared + y_squared - two_x_y)

        # ================================================================ #
        # 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.
            # ================================================================ #

        k_nearest = dists[i].argsort()[:k] #finds the indices

        #convert indices -> labels
        closest_y = self.y_train[k_nearest]

        #find mode, aka the label that appears the most; pick the smaller label␣
↪to break ties
        y_pred[i] = mode(closest_y)[0][0]

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

    return y_pred
```

svm

January 27, 2021

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

Please follow the notebook linearly to implement a linear support vector machine.

Please print out the workbook entirely when completed.

We thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu). These are the functions in the cs231n folders and includes code to preprocess and show the images. The classifiers used are based off of code prepared for CS 231n as well.

The goal of this workbook is to give you experience with training an SVM classifier via gradient descent.

0.2 Importing libraries and data setup

[33]:
```
import numpy as np # for doing most of our calculations
import matplotlib.pyplot as plt# for plotting
from cs231n.data_utils import load_CIFAR10 # function to load the CIFAR-10␣
 ↪dataset.
import pdb

# 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
```

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

[34]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = 'C:/Users/Ashwin/Desktop/UCLA/current classes/ece 247/hw2/
 ↪hw2_code/cifar-10-batches-py/'
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,)
```

[35]:
```
# 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()
```

[36]:
```python
# Split the data into train, val, and test sets. In addition we will
# create a small development set as a subset of the training data;
# we can use this for development so our code runs faster.
num_training = 49000
num_validation = 1000
num_test = 1000
num_dev = 500

# Our validation set will be num_validation points from the original
# training set.
mask = range(num_training, num_training + num_validation)
X_val = X_train[mask]
y_val = y_train[mask]

# Our training set will be the first num_train points from the original
# training set.
mask = range(num_training)
X_train = X_train[mask]
y_train = y_train[mask]

# We will also make a development set, which is a small subset of
# the training set.
mask = np.random.choice(num_training, num_dev, replace=False)
X_dev = X_train[mask]
y_dev = y_train[mask]

# We use the first num_test points of the original test set as our
# test set.
mask = range(num_test)
X_test = X_test[mask]
y_test = y_test[mask]

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, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

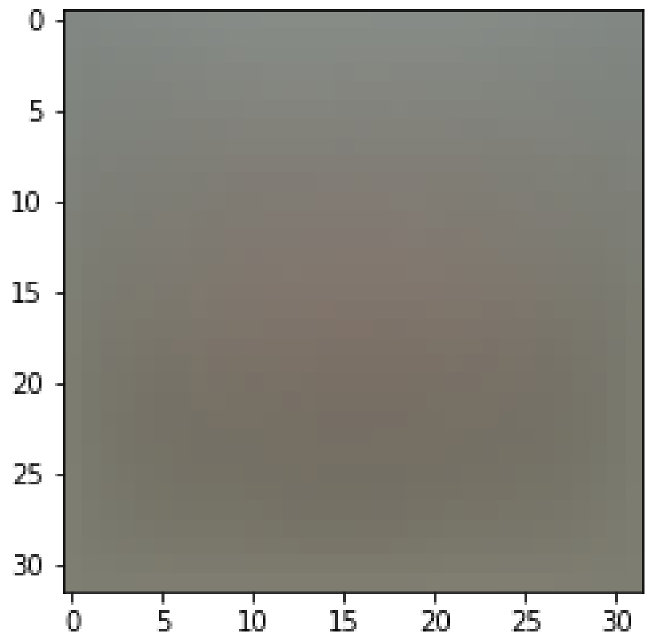
```
Dev data shape:  (500, 32, 32, 3)
Dev labels shape:  (500,)
```

[37]:
```
# 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))

# As a sanity check, print out the shapes of the data
print('Training data shape: ', X_train.shape)
print('Validation data shape: ', X_val.shape)
print('Test data shape: ', X_test.shape)
print('dev data shape: ', X_dev.shape)
```

```
Training data shape:  (49000, 3072)
Validation data shape:  (1000, 3072)
Test data shape:  (1000, 3072)
dev data shape:  (500, 3072)
```

[38]:
```
# Preprocessing: subtract the mean image
# first: compute the image mean based on the training data
mean_image = np.mean(X_train, axis=0)
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean␣
 ↪image
plt.show()
```

```
[130.64189796 135.98173469 132.47391837 130.05569388 135.34804082
 131.75402041 130.96055102 136.14328571 132.47636735 131.48467347]
```



[39]:
```
# second: subtract the mean image from train and test data
X_train -= mean_image
X_val -= mean_image
X_test -= mean_image
X_dev -= mean_image
```

[40]:
```
# third: append the bias dimension of ones (i.e. bias trick) so that our SVM
# only has to worry about optimizing a single weight matrix W.
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))])

print(X_train.shape, X_val.shape, X_test.shape, X_dev.shape)
```

```
(49000, 3073) (1000, 3073) (1000, 3073) (500, 3073)
```

0.3 Question:

(1) For the SVM, we perform mean-subtraction on the data. However, for the KNN notebook, we did not. Why?

0.4 Answer:

(1) SVM's can converge much faster on data with mean 0. However, KNN calculates based on other points, not on the overall distribution, and hence would not speed up if we made this change. Hence, this change is not needed for KNN. Additionally, mean-subtraction helps

reduce features with large magnitudes, which could potentially dominate other features when calculating the distance between the separating plane and the support vectors for SVM.

The reason we don't do normalization is since image pixels are usually homogeneous; hence, data normalization is not really needed.

0.5 Training an SVM

The following cells will take you through building an SVM. 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.

[41]:
```
from nndl.svm import SVM
```

[42]:
```
# Declare an instance of the SVM class.
# Weights are initialized to a random value.
# Note, to keep people's initial 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]

svm = SVM(dims=[num_classes, num_features])
```

SVM loss

[43]:
```
## Implement the loss function for in the SVM class(nndl/svm.py), svm.loss()

loss = svm.loss(X_train, y_train)
print('The training set loss is {}.'.format(loss))

# If you implemented the loss correctly, it should be 15569.98
```

```
The training set loss is 15569.977915410193.
```

SVM gradient

[44]:
```
## Calculate the gradient of the SVM class.
# For convenience, we'll write one function that computes the loss
#   and gradient together. Please modify svm.loss_and_grad(X, y).
# You may copy and paste your loss code from svm.loss() here, and then
#   use the appropriate intermediate values to calculate the gradient.

loss, grad = svm.loss_and_grad(X_dev,y_dev)

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

```
numerical: 1.139153 analytic: 1.139153, relative error: 1.205192e-07
numerical: 3.136700 analytic: 3.136700, relative error: 6.917619e-08
numerical: -6.469076 analytic: -6.469076, relative error: 1.453638e-08
numerical: 8.848929 analytic: 8.848930, relative error: 2.128730e-08
numerical: -4.041552 analytic: -4.041553, relative error: 7.274893e-08
numerical: -4.804845 analytic: -4.804845, relative error: 1.561937e-08
numerical: 1.420281 analytic: 1.420281, relative error: 1.180564e-07
numerical: -4.114251 analytic: -4.114252, relative error: 3.423090e-08
numerical: -3.228525 analytic: -3.228525, relative error: 4.994892e-08
numerical: -13.368588 analytic: -13.368587, relative error: 3.684456e-08
```

0.6 A vectorized version of SVM

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

[45]:
```
import time
```

[48]:
```
## Implement svm.fast_loss_and_grad which calculates the loss and gradient
#    WITHOUT using any for loops.

# Standard loss and gradient
tic = time.time()
loss, grad = svm.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 = svm.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, i.e., differences on the␣
↪order of 1e-12
```

```
Normal loss / grad_norm: 15264.509941347484 / 2005.2446248571475 computed in
0.09596419334411621s
```

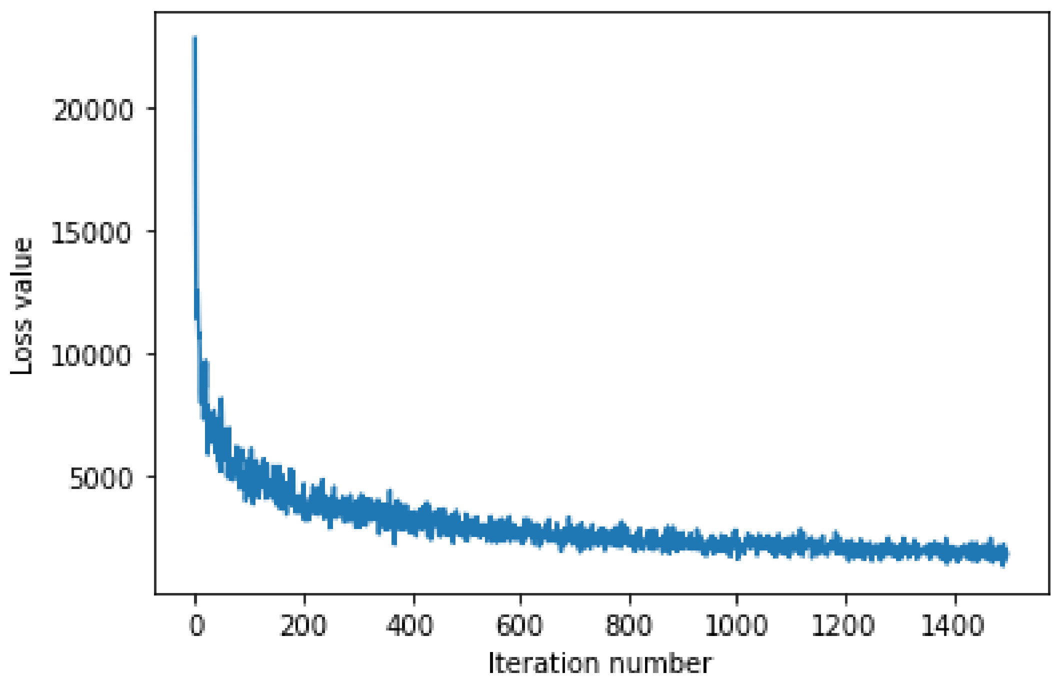
```
Vectorized loss / grad: 15264.509941347476 / 2005.2446248571475 computed in
0.0070035457611083984s
difference in loss / grad: 7.275957614183426e-12 / 3.2481758762084007e-12
```

0.7 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).

[51]:
```
# Implement svm.train() by filling in the code to extract a batch of data
# and perform the gradient step.

tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=5e-4,
                      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 22770.756607864787
iteration 100 / 1500: loss 4505.767134293777
iteration 200 / 1500: loss 3196.6474088671994
iteration 300 / 1500: loss 3179.892535847857
iteration 400 / 1500: loss 3120.17599694821
iteration 500 / 1500: loss 3413.5936203839497
iteration 600 / 1500: loss 2331.54384795122
iteration 700 / 1500: loss 2235.9108382401664
iteration 800 / 1500: loss 2519.3944914825474
iteration 900 / 1500: loss 2618.8846084258166
iteration 1000 / 1500: loss 2322.2631218538977
iteration 1100 / 1500: loss 1752.6067446319437
iteration 1200 / 1500: loss 2077.292143244585
iteration 1300 / 1500: loss 1704.1347667002772
iteration 1400 / 1500: loss 1858.1419171516927
That took 14.698964595794678s
```



0.7.1 Evaluate the performance of the trained SVM on the validation data.

[61]:
```
## Implement svm.predict() and use it to compute the training and testing error.

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

```
training accuracy: 0.28489795918367344
validation accuracy: 0.292
```

0.8 Optimize the SVM

Note, to make things faster and simpler, we won't do k-fold cross-validation, but will only optimize the hyperparameters on the validation dataset (X_val, y_val).

[67]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Train the SVM 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.
#   Note: You do not need to modify SVM class for this section
# ================================================================ #
learning_rates = [0.0001, 0.00025, 0.001, 0.0025, 0.01, 0.025, 0.1, 0.5, 1]

#generates error based on learning rate on validation test
def generateError (rate):
    svm.train(X_train, y_train, rate)
    y_test_pred = svm.predict(X_test)
    y_val_pred = svm.predict(X_val)

    test_error_rate = 1 - np.mean(np.equal(y_test,y_test_pred))
    validation_accuracy = np.mean(np.equal(y_val, y_val_pred))
    print("learning rate:", rate, "validation_accuracy", validation_accuracy,␣
 ↪"test_error_rate", test_error_rate)
    return (rate, validation_accuracy, test_error_rate)

tuples = [generateError(x) for x in learning_rates]

print(max(tuples, key = lambda x : x[1]))

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

```
learning rate: 0.0001 validation_accuracy 0.204 test_error_rate 0.787
learning rate: 0.00025 validation_accuracy 0.245 test_error_rate 0.8
learning rate: 0.001 validation_accuracy 0.252 test_error_rate 0.761
learning rate: 0.0025 validation_accuracy 0.23 test_error_rate 0.767
learning rate: 0.01 validation_accuracy 0.271 test_error_rate 0.748
learning rate: 0.025 validation_accuracy 0.276 test_error_rate 0.727
learning rate: 0.1 validation_accuracy 0.271 test_error_rate 0.732
learning rate: 0.5 validation_accuracy 0.237 test_error_rate 0.758
learning rate: 1 validation_accuracy 0.274 test_error_rate 0.757
(0.025, 0.276, 0.727)
```

As we can see, the best validation_accuracy is 0.276, which means the best validation error is 1 - 0.276 = 0.724. This corresponds to a learning rate of 0.025 and a test error rate of 0.727.

0.9 SVM Notebook Code:

[ ]:
```
import numpy as np
import pdb

"""
This code was based off of code from cs231n at Stanford University, and␣
 ↪modified for ECE C147/C247 at UCLA.
```

```
"""
class SVM(object):

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

  def init_weights(self, dims):
    """
        Initializes the weight matrix of the SVM.  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)

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

    # compute the loss and the gradient
    num_classes = self.W.shape[0]
    num_train = X.shape[0]
    loss = 0.0

    for i in np.arange(num_train):
    # ================================================================ #
    # YOUR CODE HERE:
          #   Calculate the normalized SVM loss, and store it as '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.)
    # ================================================================ #

    #using https://mlxai.github.io/2017/01/06/
↪vectorized-implementation-of-svm-loss-and-gradient-update.html for a␣
↪reference (Piazza @200)
```

```
        scores = np.dot(self.W, X[i,:]).T
        correct_class_score = scores[y[i]]

        for j in np.arange(num_classes):
            #there will be no loss if we are in the same class
            if j == y[i]:
                continue
            loss += max(0, scores[j] - correct_class_score + 1) #delta = 1 in␣
↪this case

    #normalize
    loss = loss / num_train

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


    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.
        """

    # compute the loss and the gradient
    num_classes = self.W.shape[0]
    num_train = X.shape[0]
    loss = 0.0
    grad = np.zeros_like(self.W)

    for i in np.arange(num_train):
    # ================================================================ #
    # YOUR CODE HERE:
        #   Calculate the SVM loss and the gradient.  Store the gradient in
    #   the variable grad.
    # ================================================================ #
        scores = np.dot(self.W, X[i,:]).T
        correct_class_score = scores[y[i]]

        for j in np.arange(num_classes):
            #there will be no loss if we are in the same class
            if j == y[i]:
                continue
            #delta = 1 in this case; according to https://cs231n.github.io/
↪linear-classify/, it should always be 1?
            margin = scores[j] - correct_class_score + 1

            if margin > 0:
                loss += margin
                grad[y[i],:] -= X[i,:]
                grad[j,:] += X[i,:]

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

    loss /= num_train
    grad /= num_train

    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_analytic))
      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 SVM loss WITHOUT any for loops.
    # ================================================================ #

    #using the same reference as above

    num_classes = self.W.shape[0]
    num_train = X.shape[0]

    scores = X.dot(self.W.T)
    yi_scores = scores[np.arange(scores.shape[0]), y]
    margins = np.maximum(0, (scores.T - np.matrix(yi_scores)).T + 1) #delta = 1␣
↪always
    margins[np.arange(num_train),y] = 0
    loss = np.mean(np.sum(margins, axis=1))
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #




        # ================================================================ #
    # YOUR CODE HERE:
        #   Calculate the SVM grad WITHOUT any for loops.
    # ================================================================ #
     #use Guangyuan's discussion as reference:https://colab.research.google.com/
↪drive/1ugOJc2qZHMQYh2sKf3QXz9-QbH5t5FIW and the other reference mentioned␣
↪above

    binary = margins
    binary[margins > 0] = 1
    row_sum = np.sum(binary, axis=1)
    binary[np.arange(num_train), y] = -row_sum.T
    grad = np.dot(binary.T, X)

    #average
    grad /= num_train

    # ================================================================ #
    # 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: (dim, batch_size)
            #     - 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.
      # ================================================================ #

      #pick batch_size random indices from {0, num_train)
      indices = np.random.choice(np.arange(num_train), batch_size)
      X_batch = X[indices]
      y_batch = y[indices]

      # ================================================================ #
      # 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 = 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 with the parameter self.W.
    # ================================================================ #

    #np.dot(X, W^T) = np.matmul(XW)^T
    #argmax = indices of max values along axis
    product = np.dot(self.W, X.T)
    y_pred = np.argmax(product, axis=0)

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

    return y_pred
```

softmax

January 27, 2021

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.

We thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu). These are the functions in the cs231n folders and code in the jupyer notebook to preprocess and show the images. The classifiers used are based off of code prepared for CS 231n as well.

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

[1]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

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

[3]:
```
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 = 'C:/Users/Ashwin/Desktop/UCLA/current classes/ece 247/hw2/
 ↪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.

[4]:
```
from nndl import Softmax
```

[5]:
```
# 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

[19]:
```
## Implement the loss function of the softmax using a for loop over
#   the number of examples

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

[20]:
```
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:

No training has been performed, so each class will be classified with a probability of 0.1. If we take the negative log of 0.1 (-ln(0.1)), we get ~2.3, which corresponds to our loss.

Softmax gradient

[23]:
```
## 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: -0.722136 analytic: -0.722136, relative error: 3.738540e-09
numerical: 0.819143 analytic: 0.819143, relative error: 2.177321e-08
numerical: 1.817788 analytic: 1.817788, relative error: 1.423580e-09
numerical: -0.422552 analytic: -0.422552, relative error: 5.372484e-09
numerical: -0.161966 analytic: -0.161966, relative error: 3.918769e-07
numerical: 0.868902 analytic: 0.868902, relative error: 3.689266e-08
numerical: 0.286321 analytic: 0.286321, relative error: 8.394643e-08
numerical: 0.270276 analytic: 0.270276, relative error: 1.540718e-08
numerical: -0.302617 analytic: -0.302617, relative error: 9.591102e-09
numerical: -1.046201 analytic: -1.046201, relative error: 5.848762e-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.

[21]:
```
import time
```

[29]:
```
## 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: 2.3235766236006388 / 332.5967097296429 computed in
0.24800753593444824s
Vectorized loss / grad: 2.3235766236006383 / 332.5967097296429 computed in
0.008000373840332031s
difference in loss / grad: 4.440892098500626e-16 /2.198458141915654e-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:

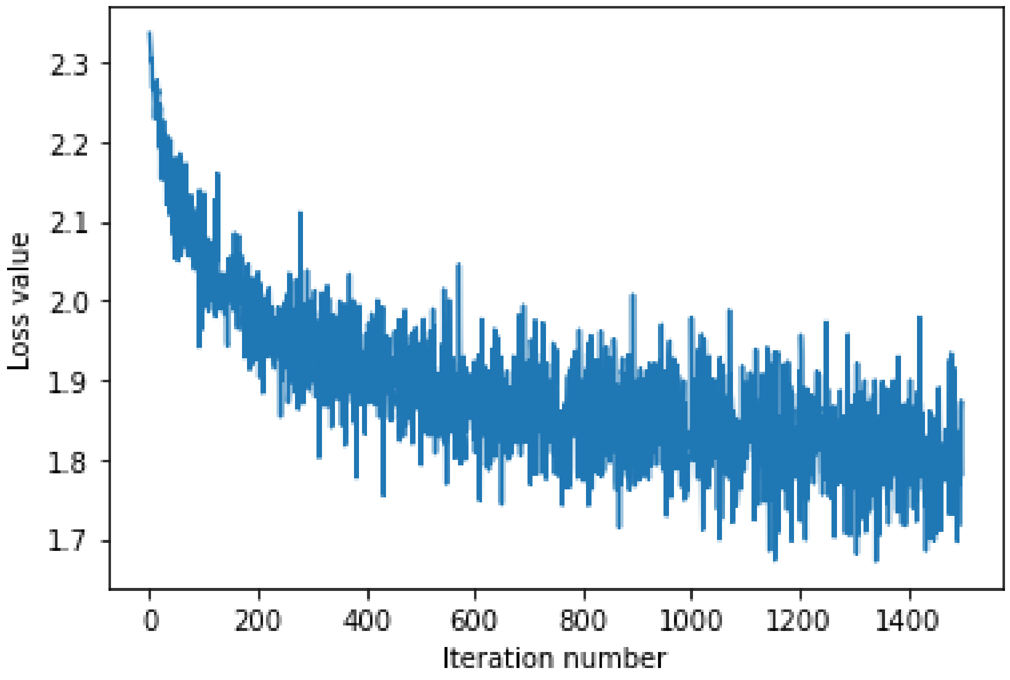
How should the softmax gradient descent training step differ from the svm training step, if at all?

0.8 Answer:

Since the only difference between the 2 classifiers are the loss functions, the training steps will be the same.

[30]:
```
# 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 2.336592660663754
iteration 100 / 1500: loss 2.0557222613850827
iteration 200 / 1500: loss 2.035774512066282
iteration 300 / 1500: loss 1.9813348165609885
iteration 400 / 1500: loss 1.9583142443981612
iteration 500 / 1500: loss 1.8622653073541355
iteration 600 / 1500: loss 1.8532611454359382
iteration 700 / 1500: loss 1.8353062223725825
iteration 800 / 1500: loss 1.829389246882764
iteration 900 / 1500: loss 1.899215853035748
iteration 1000 / 1500: loss 1.9783503540252303
iteration 1100 / 1500: loss 1.8470797913532635
iteration 1200 / 1500: loss 1.8411450268664085
iteration 1300 / 1500: loss 1.79104024957921
iteration 1400 / 1500: loss 1.870580302938226
That took 17.281003713607788s
```



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

[31]:
```
## 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.

[33]:
```
np.finfo(float).eps
```

[33]:
```
2.220446049250313e-16
```

[37]:
```
# ================================================================ #
# 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 = [0.000001, 0.00001, 0.001, 0.0025, 0.01, 0.025, 0.1, 0.5, 1]

#generates error based on learning rate on validation test
def generateError (rate):
    softmax.train(X_train, y_train, rate)
    y_test_pred = softmax.predict(X_test)
    y_val_pred = softmax.predict(X_val)

    test_error_rate = 1 - np.mean(np.equal(y_test,y_test_pred))
    validation_accuracy = np.mean(np.equal(y_val, y_val_pred))
    print("learning rate:", rate, "validation_accuracy", validation_accuracy,␣
 ↪"test_error_rate", test_error_rate)
    return (rate, validation_accuracy, test_error_rate)

tuples = [generateError(x) for x in learning_rates]

print(max(tuples, key = lambda x : x[1]))
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
learning rate: 1e-06 validation_accuracy 0.374 test_error_rate 0.632
learning rate: 1e-05 validation_accuracy 0.327 test_error_rate
0.6970000000000001
learning rate: 0.001 validation_accuracy 0.087 test_error_rate 0.897
learning rate: 0.0025 validation_accuracy 0.087 test_error_rate 0.897
learning rate: 0.01 validation_accuracy 0.087 test_error_rate 0.897
learning rate: 0.025 validation_accuracy 0.087 test_error_rate 0.897
learning rate: 0.1 validation_accuracy 0.087 test_error_rate 0.897
learning rate: 0.5 validation_accuracy 0.087 test_error_rate 0.897
learning rate: 1 validation_accuracy 0.087 test_error_rate 0.897
(1e-06, 0.374, 0.632)
```

As we can see, the learning rate that performs the best is 1e-06, with a validation_accuracy of 0.374, and hence a validation error rate of 1 - 0.374 = 0.626, and a test error rate of 0.632 (which is also the lowest in all of the samples, showing that our hyperparameter tuning worked as expected).

0.10 Softmax Notebook:

```
[ ]: import numpy as np
     import math

     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

         # ================================================================ #
         # 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.)
    # ================================================================ #

    product = np.dot(X, self.W.T)

    for i, row in enumerate(product):
        row -= np.max(row)
        inner_sum = np.exp(row[y[i]]) / np.sum(np.exp(row)) #the inner sum of␣
↪softmax loss
        logsum = -np.log(inner_sum) #applying negative log to the sum
        loss += logsum

    #normalize
    loss = loss / len(X)

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

    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.
    # ================================================================ #
    product = np.dot(X, self.W.T)

    for i, row in enumerate(product):
        row -= np.max(row)
        inner_sum = np.exp(row[y[i]]) / np.sum(np.exp(row)) #the inner sum of␣
↪softmax loss
        logsum = -np.log(inner_sum) #applying negative log to the sum
        loss += logsum

        #for each class, calculating the gradient using the Softmax Derivative␣
↪hint on CCLE
        for j in range(10):
            j_softmax = np.exp(row[j]) / np.sum(np.exp(row))
            if j == y[i]:
                grad[j] += X[i] * (j_softmax - 1)
            else:
                grad[j] += X[i] * j_softmax

    #normalize
    loss = loss / len(X)
    grad = grad / len(X)

    # ================================================================ #
    # 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_analytic))
      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.
    # ================================================================ #

    #used this as a resource: https://mlxai.github.io/2017/01/09/
↪implementing-softmax-classifier-with-vectorized-operations.html

    N = X.shape[0]

    product = np.dot(X, self.W.T)
    sum_j = np.sum(np.exp(product), axis=1, keepdims=True)
    coefs = np.exp(product) / sum_j
    coefs[np.arange(N), y] -= 1
    loss = np.log(sum_j) - product[np.arange(N), y]
    loss = np.mean(loss)
    grad = coefs.T.dot(X)
    grad /= len(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: (dim, batch_size)
          #     - 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.
      # ================================================================ #
      indices = np.random.choice(np.arange(num_train), batch_size)
      X_batch = X[indices]
      y_batch = y[indices]
      # ================================================================ #
      # 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 = 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.
    # ================================================================ #
    product = np.dot(self.W, X.T)
    y_pred = np.argmax(product, axis=0)
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return y_pred
```