knn

January 25, 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

[1]:
```
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
```

[2]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = '../cifar-10-batches-py' # You need to update this line
X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

# 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 is simply storing the training dataset in memory

(2) Training takes O(1) time but O(n) space

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: 31.894300937652588
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: 1.3185920715332031
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.
# ================================================================ #
error = np.count_nonzero(knn.predict_labels(dists_L2_vectorized)-y_test)/
 ↪len(y_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.

[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]
# ================================================================ #
train = np.hstack((X_train, y_train[:,np.newaxis]))
np.random.seed(0)
np.random.shuffle(train)
X_train, y_train = train[:, :-1], train[:, -1].astype(int)
X_train_folds = np.split(X_train,num_folds)
y_train_folds = np.split(y_train,num_folds)

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

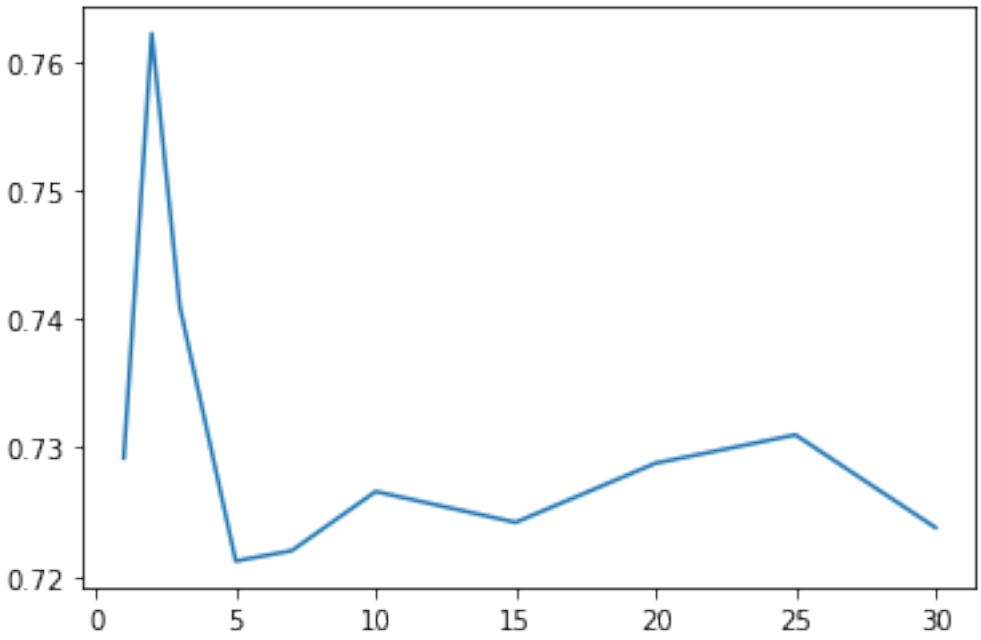
2.0.2 Optimizing the number of nearest neighbors hyperparameter.

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

[11]:
```
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.
# ================================================================ #
from copy import copy

errors = {}
for k in ks:
    curr_errors = []
    for fold in range(num_folds):
        X_train_folds_temp, y_train_folds_temp = copy(X_train_folds),␣
↪copy(y_train_folds)
        X_val_fold, y_val_fold = X_train_folds_temp.pop(fold),␣
↪y_train_folds_temp.pop(fold)
```

```
        knn.train(X=np.concatenate(X_train_folds_temp), y=np.
 ↪concatenate(y_train_folds_temp))
        dists = knn.compute_L2_distances_vectorized(X=X_val_fold)
        curr_errors.append(np.count_nonzero(knn.predict_labels(dists,␣
 ↪k)-y_val_fold)/len(y_val_fold))
    errors[k] = np.average(curr_errors)
print(errors)
plt.plot(ks, errors.values())
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

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

```
{1: 0.7292, 2: 0.7621999999999999, 3: 0.741, 5: 0.7212, 7: 0.722, 10: 0.7266,
15: 0.7242, 20: 0.7288, 25: 0.731, 30: 0.7238}
Computation time: 67.89
```



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) k = 5

(2) CV-error for k=5 is 0.7212

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.

[14]:
```
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.
# ================================================================ #

errors = []
for norm in norms:
    curr_errors = []
    for fold in range(num_folds):
        X_train_folds_temp, y_train_folds_temp = copy(X_train_folds),␣
↪copy(y_train_folds)
        X_val_fold, y_val_fold = X_train_folds_temp.pop(fold),␣
↪y_train_folds_temp.pop(fold)
        knn.train(X=np.concatenate(X_train_folds_temp), y=np.
↪concatenate(y_train_folds_temp))
        dists = knn.compute_distances(X=X_val_fold, norm=norm)
        curr_errors.append(np.count_nonzero(knn.predict_labels(dists,␣
↪k=5)-y_val_fold)/len(y_val_fold))
    errors.append(np.average(curr_errors))
print(errors)
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
print('Computation time: %.2f'%(time.time()-time_start))
```

```
[0.6958, 0.7212, 0.8353999999999999]
Computation time: 643.66
```

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

(2) L1-Norm & k= 25 -> 0.6886

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.

[16]:
```
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.
# ================================================================ #

knn.train(X=X_train, y=y_train)
dists = knn.compute_distances(X=X_test, norm=L1_norm)
error = np.count_nonzero(knn.predict_labels(dists, k=5)-y_test)/len(y_test)

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

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

```
Error rate achieved: 0.698
```

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 dropped from 0.726 to 0.698

[ ]:
```

```

svm

January 25, 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

[1]:
```
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
```

[2]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = '../cifar-10-batches-py' # You need to update this line
X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

# 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]:
```
# 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)
```

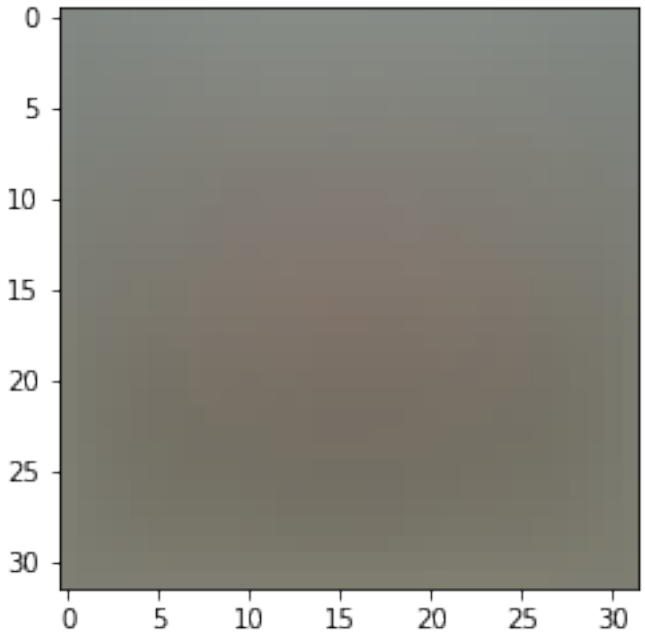
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
Dev data shape:  (500, 32, 32, 3)
Dev labels shape:  (500,)
```

[5]:
```
# 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)
```

[6]:
```
# 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]
```



[7]:
```
# 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
```

[8]:
```
# 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) Mean-Subtraction in KNN would not change the output since we are calculating relative differences between images, but for SVM we have to normalize the data so each image will affect gradient and loss computations evenly

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.

[9]:
```
from nndl.svm import SVM
```

[10]:
```
# 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

[11]:
```
## 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 15569.97791541019.
```

SVM gradient

[12]:
```
## 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: -5.890541 analytic: -5.890542, relative error: 1.032196e-07
numerical: 1.594771 analytic: 1.594771, relative error: 3.688664e-08
numerical: -6.516308 analytic: -6.516307, relative error: 5.936412e-08
numerical: 6.464212 analytic: 6.464212, relative error: 3.593814e-08
numerical: -2.454658 analytic: -2.454658, relative error: 1.728085e-10
numerical: 1.408333 analytic: 1.408332, relative error: 1.359279e-07
numerical: 7.764712 analytic: 7.764711, relative error: 6.596002e-08
numerical: -15.967384 analytic: -15.967382, relative error: 5.455627e-08
numerical: -11.466394 analytic: -11.466393, relative error: 1.629109e-08
numerical: -19.091872 analytic: -19.091872, relative error: 2.986340e-09
```

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.

[13]:
```
import time
```

[14]:
```
## 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
```

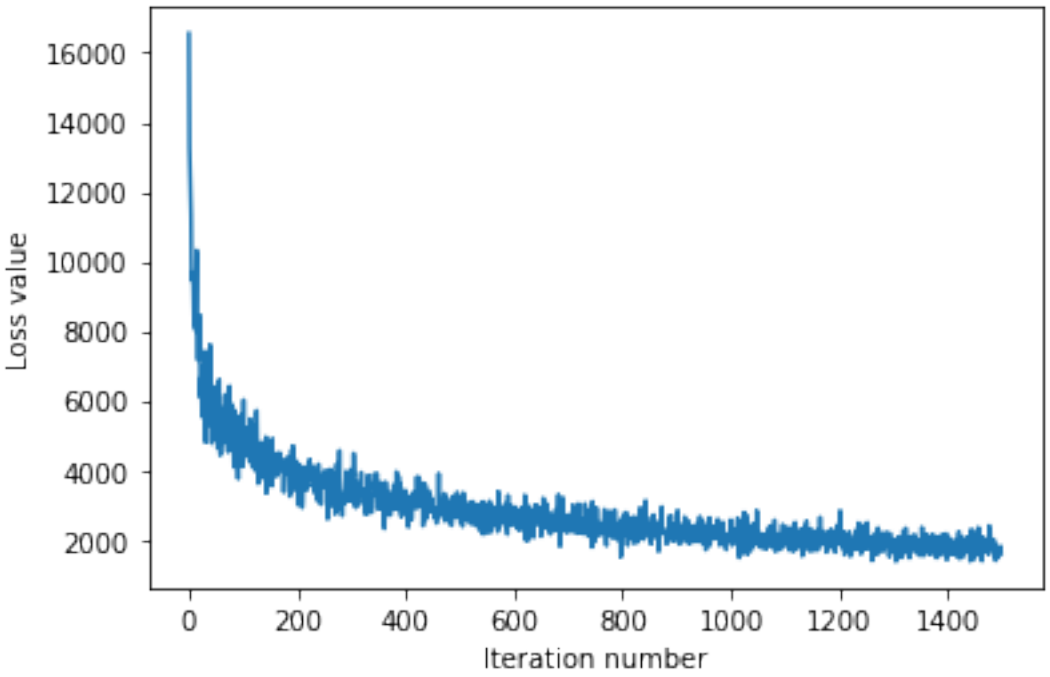
```
Normal loss / grad_norm: 14551.391406280789 / 2385.430795390602 computed in
0.09059929847717285s
Vectorized loss / grad: 14551.391406280767 / 2385.430795390602 computed in
0.0052869319915771484s
difference in loss / grad: 2.1827872842550278e-11 / 5.0972131053994295e-12
```

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).

[15]:
```
# 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 16557.38000190916
iteration 100 / 1500: loss 4701.089451272714
iteration 200 / 1500: loss 4017.333137942789
iteration 300 / 1500: loss 3681.9226471953625
iteration 400 / 1500: loss 2732.6164373988995
iteration 500 / 1500: loss 2786.6378424645054
iteration 600 / 1500: loss 2837.035784278267
iteration 700 / 1500: loss 2206.2348687399317
iteration 800 / 1500: loss 2269.0388241169803
iteration 900 / 1500: loss 2543.237815385921
iteration 1000 / 1500: loss 2566.6921357268275
iteration 1100 / 1500: loss 2182.068905905164
iteration 1200 / 1500: loss 1861.1182244250458
iteration 1300 / 1500: loss 1982.9013858528251
iteration 1400 / 1500: loss 1927.520415858212
That took 5.656845808029175s
```



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

[16]:
```
## 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: 0.28530612244897957
validation accuracy: 0.3
```

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).

[17]:
```
# ================================================================ #
# 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
# ================================================================ #
rates = [1e-3, 1e-4, 1e-5, 1e-6, 1e-7, 1e-8, 1e-9, 1e-10]
max_accuracy = (None, 0)
for rate in rates:
    svm.train(X_train, y_train, learning_rate=rate,
                      num_iters=1500, verbose=False)
    y_val_pred = svm.predict(X_val)
    acc = (rate,np.mean(np.equal(y_val, y_val_pred)))
    max_accuracy = max(max_accuracy, acc, key=lambda x: x[1])
print(f"Best Learning Rate: {max_accuracy[0]}\nValidation Set Accuracy:
 ↪{max_accuracy[1]}\nValidation Set Error:{1-max_accuracy[1]}")
svm.train(X_train, y_train, learning_rate=max_accuracy[0],
                      num_iters=1500, verbose=False)
y_test_pred = svm.predict(X_test)
acc = np.mean(np.equal(y_test, y_test_pred))
print("Test Set Error: ", 1-acc)
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
Best Learning Rate: 0.001
Validation Set Accuracy:0.294
```

```
Validation Set Error:0.706
Test Set Error:  0.758
```

[ ]:

softmax

January 25, 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
```

[2]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000,␣
↪num_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the linear classifier. These are the same steps as we used for the
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = '../cifar-10-batches-py' # You need to update this line
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
```

```
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

    # Preprocessing: reshape the image data into rows
    X_train = np.reshape(X_train, (X_train.shape[0], -1))
    X_val = np.reshape(X_val, (X_val.shape[0], -1))
    X_test = np.reshape(X_test, (X_test.shape[0], -1))
    X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis = 0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image
    X_dev -= mean_image

    # add bias dimension and transform into columns
    X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
    X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
    X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
    X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

    return X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev =␣
 ↪get_CIFAR10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
print('dev data shape: ', X_dev.shape)
print('dev labels shape: ', y_dev.shape)
```

```
Train data shape:  (49000, 3073)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3073)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3073)
Test labels shape:  (1000,)
```

```
dev data shape:  (500, 3073)
dev labels shape:  (500,)
```

0.2 Training a softmax classifier.

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

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

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

np.random.seed(1)

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

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

Softmax loss

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

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

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

```
2.3277607028048966
```

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:

Since all scores are initialized to 0, the loss is the natural log of the number of classes (10)-> 2.3

Softmax gradient

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

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

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

```
numerical: 0.301892 analytic: 0.301892, relative error: 4.713473e-09
numerical: 0.412108 analytic: 0.412108, relative error: 1.306684e-07
numerical: -2.543329 analytic: -2.543329, relative error: 4.141162e-09
numerical: 1.474428 analytic: 1.474427, relative error: 2.611153e-08
numerical: 0.807006 analytic: 0.807006, relative error: 6.768935e-08
numerical: 2.075951 analytic: 2.075951, relative error: 2.178458e-08
numerical: 0.856041 analytic: 0.856040, relative error: 6.441477e-08
numerical: -1.398324 analytic: -1.398324, relative error: 1.355805e-08
numerical: -0.306065 analytic: -0.306065, relative error: 2.836512e-09
numerical: -4.762989 analytic: -4.762989, relative error: 1.079254e-08
```

0.5 A vectorized version of Softmax

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

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

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

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

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

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

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

```
Normal loss / grad_norm: 2.3410310932467624 / 356.1972823983476 computed in
0.08643984794616699s
Vectorized loss / grad: 2.3410310932467615 / 356.1972823983477 computed in
0.005819797515869141s
difference in loss / grad: 8.881784197001252e-16 /2.2939937255122995e-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?

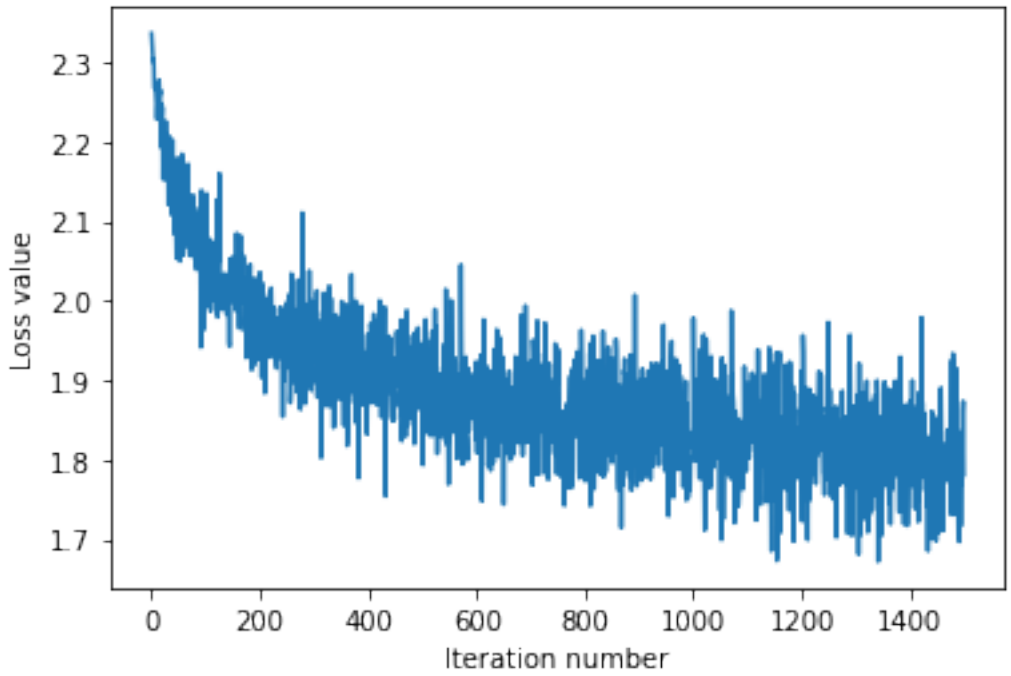
0.8 Answer:

The training step is identical except the gradient calculation and loss will just be computed differently

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


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

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

```
iteration 0 / 1500: loss 2.3365926606637544
iteration 100 / 1500: loss 2.0557222613850827
iteration 200 / 1500: loss 2.0357745120662813
iteration 300 / 1500: loss 1.9813348165609888
iteration 400 / 1500: loss 1.9583142443981612
iteration 500 / 1500: loss 1.8622653073541355
iteration 600 / 1500: loss 1.8532611454359382
iteration 700 / 1500: loss 1.8353062223725827
iteration 800 / 1500: loss 1.829389246882764
iteration 900 / 1500: loss 1.8992158530357484
```

```
iteration 1000 / 1500: loss 1.97835035402523
iteration 1100 / 1500: loss 1.8470797913532633
iteration 1200 / 1500: loss 1.8411450268664082
iteration 1300 / 1500: loss 1.79104024957921
iteration 1400 / 1500: loss 1.8705803029382257
That took 5.022393703460693s
```



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

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

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

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

0.9 Optimize the softmax classifier

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

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

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

[13]:
```
# ================================================================ #
# 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.
# ================================================================ #
rates = [1e-4, 1e-5, 1e-6, 1e-7, 1e-8, 1e-9, 1e-10]
max_accuracy = (None, 0)
for rate in rates:
    softmax.train(X_train, y_train, learning_rate=rate,
                      num_iters=1500, verbose=False)
    y_val_pred = softmax.predict(X_val)
    acc = (rate,np.mean(np.equal(y_val, y_val_pred)))
    max_accuracy = max(max_accuracy, acc, key=lambda x: x[1])
print(f"Best Learning Rate: {max_accuracy[0]}\nValidation Set Accuracy:
 ↪{max_accuracy[1]}\nValidation Set Error:{1-max_accuracy[1]}")
softmax.train(X_train, y_train, learning_rate=max_accuracy[0],
                      num_iters=1500, verbose=False)
y_test_pred = softmax.predict(X_test)
acc = np.mean(np.equal(y_test, y_test_pred))
print("Test Set Error: ", 1-acc)
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
Best Learning Rate: 1e-06
Validation Set Accuracy:0.405
Validation Set Error:0.595
Test Set Error:  0.608
```

[ ]:
```

```