Q1.

# EE 247 Hw 2

(a) $\delta^{(i)} \sim N(0, \sigma^2 I)$ $\qquad$ $\delta^{(i)}$ iid

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

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

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

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

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

$$E[(y^{(i)} - x^{(i)T}\theta)^2] = (y^{(i)} - x^{(i)T}\theta)^2$$

$\uparrow$ deterministic

$$E[(y^{(i)} - x^{(i)T}\theta)(\delta^{(i)T}\theta)] = E[y^{(i)}\delta^{(i)T}\theta] - E[x^{(i)T}\theta\cdot\delta^{(i)T}\theta]$$

$$= y^{(i)} E[\delta^{(i)T}]\theta - x^{(i)T}\theta\, E[\delta^{(i)T}]\theta$$

$$= 0$$

$$E[(\delta^{(i)T}\theta)^2] = E[(\delta^{(i)T}\theta)^T(\delta^{(i)T}\theta)] = E[\theta^T\delta^{(i)}\cdot\delta^{(i)T}\theta]$$

$$= \theta^T E[\delta^{(i)}\delta^{(i)T}]\cdot\theta$$

$$= \theta^T\cdot\sigma^2 I\,\theta$$

$$= \sigma^2\cdot\theta^T\theta$$

$$= \sigma^2\cdot\|\theta\|_2^2$$

$$\therefore E_{\delta\sim N}[\tilde{\mathcal{L}}(\theta)] = \frac{1}{N}\cdot\sum_{i=1}^{N}\left((y^{(i)} - x^{(i)T}\theta)^2 + \sigma^2\cdot\|\theta\|_2^2\right)$$

$$= \frac{1}{N}\cdot\sum_{i=1}^{N}(y^{(i)} - x^{(i)T}\theta)^2 + \frac{1}{\cancel{N}}\cdot\cancel{N}\cdot\sigma^2\|\theta\|_2^2$$

$\therefore E_\delta[\tilde{\mathcal{L}}(\theta)] = \mathcal{L}(\theta) + \sigma^2 \|\theta\|_2^2$

$\therefore \quad R = \sigma \|\theta\|_2^2$

(b) a L-2 Regularization is implicitly applied through noise

(c) $\sigma \to 0 \Rightarrow E_\delta[\tilde{\mathcal{L}}(\theta)] \to \mathcal{L}(\theta)$, which is the Original Mean Squared Error

It would tend to Overfit data with no Regularization

(d) $\sigma \to \infty \Rightarrow E_\delta[\tilde{\mathcal{L}}(\theta)] \to \sigma \|\theta\|_2^2$

Param $\theta$ would be $0$ To Minimize our Loss Function

$\therefore$ Model would Underfit

Q3.

$$\log \text{softmax}_i(x) = \log\left(\frac{e^{a_i(x)}}{\sum_{k=1}^{C} e^{a_k(x)}}\right)$$

$$= a_i(x) - \log \sum_{k=1}^{C} e^{a_k(x)}$$

Let $a_i(x) = w_i^T x + b_i = \tilde{w}_i x = z_i$

$$\text{softmax}_i(x) = f(z_i) = \frac{e^{z_i}}{\sum_{k=1}^{C} e^{z_k}} = r_i$$

$-\log(\text{softmax}_i(x))$

$= -\log(r_i)$

$= \mathcal{L}(r_i)$

$$\nabla_{w_i}\mathcal{L} = \frac{\partial \mathcal{L}(r_i)}{\partial w_i} = \underset{①}{\frac{\partial r_i}{\partial w_i}} \cdot \underset{②}{\frac{\partial \mathcal{L}(r_i)}{\partial r_i}} \quad (i=j)$$

Step ① : $\frac{\partial r_i}{\partial w_i} = \frac{\partial z_i}{\partial w_i} \cdot \frac{\partial r_i}{\partial z_i}$

$$\frac{\partial z_i}{\partial w_i} = \frac{\partial(w_i^T x + b_i)}{\partial w_j} = \frac{\partial w_i^T x}{\partial w_i} = x$$

$$\frac{\partial r_i}{\partial z_i} = \frac{\frac{\partial(e^{z_i})}{\partial z_i} \cdot \sum_k e^{z_k} - \frac{\partial \sum_k e^{z_k}}{\partial z_i} \cdot e^{z_i}}{(\sum_k e^{z_k})^2}$$

$$= \frac{e^{z_i} \cdot \sum_k e^{z_k} - e^{z_i} \cdot e^{z_i}}{(\sum_k e^{z_k})^2}$$

$$= \frac{e^{z_i}}{\sum_k e^{z_k}} - \left(\frac{e^{z_i}}{\sum_k e^{z_k}}\right)^2$$

$$= r_i - r_i^2$$

$$= r_i(1-r_i)$$

$$\therefore \frac{\partial r_i}{\partial w_i} = x \cdot r_i(1-r_i)$$

Step ② : $\frac{\partial \mathcal{L}(r_i)}{\partial r_j}$

$= -\frac{1}{r_i}$

☆

$\therefore \nabla_{w_i}\mathcal{L} \quad i=1,\dots,C$

$= x \cdot r_i(1-r_i)\left(-\frac{1}{r_i}\right)$

$= (r_i - 1)x$

$(i=j)$

if $i \neq j$. using similar approach

$$\frac{\partial r_j}{\partial w_i} = \underset{②}{\frac{\partial z_i}{\partial w_i}} \; \underset{①}{\frac{\partial r_j}{\partial z_i}}$$

① $$\frac{\partial r_j}{\partial z_i} = \frac{\partial}{\partial z_i}\left[\frac{e^{z_j}}{\sum_p e^{z_p}}\right] = \frac{0 - e^{z_j} e^{z_i}}{(\sum_p e^{z_p})^2}$$

$$= -\frac{e^{z_j}}{\sum_p e^{z_p}} \cdot \frac{e^{z_i}}{\sum_p e^{z_p}}$$

$$= -r_j r_i$$

② $$\frac{\partial z_i}{\partial w_i} = x$$

$$\therefore \frac{\partial r_j}{\partial w_i} = -r_j r_i x$$

$$\therefore \frac{\partial \mathcal{L}_j}{\partial w_i} = \nabla_{w_i} C(r_j) = \frac{\partial r_j}{\partial w_i} \frac{\partial C(r_j)}{\partial r_j}$$

$$= -r_j r_i x \cdot -\frac{1}{r_j}$$

$$= r_i x$$

$(i \neq j)$

∴ when $y(d) = i$ ( Match w/ Label for $d$th Data) ( In SoftMax )

$$\frac{\partial \mathcal{L}_d(\theta)}{\partial w_i} = \left[ \frac{e^{w_{y(d)}^T x(d)}}{\sum_{p=1}^{k} e^{w_p^T x(d)}} - 1 \right] x(d)$$

when $y(d) \neq i$

$$\frac{\partial \mathcal{L}_d(\theta)}{\partial w_i} = \left[ \frac{e^{w_i^T x(d)}}{\sum_{p=1}^{k} e^{w_p^T x(d)}} \right] x(d)$$

$$\frac{\partial \mathcal{L}}{\partial w_i} = \sum_d \frac{\partial \mathcal{L}_d}{\partial w_i}$$

② $i = j$

$$\nabla_{b_i} \mathcal{L}_i = \frac{\partial z_i}{\partial b_i} \cdot \frac{\partial r_i}{\partial z_i} \cdot \frac{\partial \mathcal{L}_i}{\partial r_i}$$

$$= 1 \cdot r_i(1-r_i) \cdot \left(-\frac{1}{r_i}\right)$$

$$= r_i - 1$$

$i \neq j$

$$\nabla_{b_i} \mathcal{L}_j = \frac{\partial z_i}{\partial b_i} \frac{\partial r_j}{\partial z_i} \frac{\partial \mathcal{L}_j}{\partial r_j}$$

$$= 1 \cdot (-r_j r_i) \cdot \left(-\frac{1}{r_j}\right)$$

$$= r_i$$

$$\therefore \nabla_{b_i} \mathcal{L}_j = \begin{cases} r_i - 1 & (i = j) \\ r_i & (i \neq j) \end{cases}$$

(3)

$$\tilde{w}_i = \begin{bmatrix} w_i \\ b_i \end{bmatrix}$$

$$\nabla_{\tilde{w}_i} \mathcal{L}_j = \begin{bmatrix} \partial \mathcal{L} / \partial w_i \\ \partial \mathcal{L} / \partial b_i \end{bmatrix} = \begin{cases} \begin{bmatrix} (r_i - 1) x \\ r_i - 1 \end{bmatrix} & (i = j) \\ \begin{bmatrix} r_i x \\ r_i \end{bmatrix} & (i \neq j) \end{cases}$$

Q4.

$$\mathcal{L}(w,b) = \frac{1}{K}\sum_{i=1}^{K} hinge_{y^{(i)}}(x^{(i)})$$

$w \in \mathbb{R}^d$
$b \in \mathbb{R}$

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

$$\nabla_w \mathcal{L} = \frac{\partial \mathcal{L}(w,b)}{\partial w}$$

$$= \frac{1}{K}\sum_{i=1}^{K} \nabla_w \underline{hinge_{y^{(i)}}(x^{(i)})}$$

$$\Downarrow$$

$$\max(0, 1-y^{(i)}(w^T x^{(i)}+b)) = \begin{cases} 0 & \text{if } y^{(i)}(w^T x^{(i)}+b) > 1 \\ 1-y^{(i)}(w^T x^{(i)}+b) & \text{if } y^{(i)}(w^T x^{(i)}+b) < 1 \end{cases}$$

if $y^{(i)}(w^T x^{(i)}+b) > 1 \Rightarrow \nabla_w hinge_{y^{(i)}}(x^{(i)}) = \underset{d\times 1}{\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}}$ $(\in \mathbb{R}^d)$

if $y^{(i)}(w^T x^{(i)}+b) < 1 \Rightarrow \nabla_w hinge_{y^{(i)}}(x^{(i)}) = \frac{\partial(1-y^{(i)}w^T x^{(i)} - y^{(i)}b)}{\partial w}$

$$= 0 - y^{(i)}x^{(i)} - 0$$

$$= -y^{(i)}x^{(i)} \quad (\in \mathbb{R}^d)$$

$$\therefore \nabla_w \mathcal{L} = \frac{1}{K}\sum_{i=1}^{K} \mathbb{I}_{\{y^{(i)}(w^T x^{(i)}+b)<1\}} \odot (-y^{(i)}x^{(i)})$$

$$\nabla_b \mathcal{L} = \frac{\partial \mathcal{L}}{\partial b} = \frac{1}{K} \sum_{i=1}^{K} \nabla_b \, hinge_{y^{(i)}}(x^{(i)})$$

if $y^{(i)}(w^T x^{(i)} + b) > 1 \Rightarrow \nabla_b \, hinge_{y^{(i)}}(x^{(i)}) = 0 \quad (\in \mathbb{R})$

if $y^{(i)}(w^T x^{(i)} + b) < 1 \Rightarrow \nabla_b \, hinge_{y^{(i)}}(x^{(i)}) = \dfrac{\partial(1 - y^{(i)} w^T x - y^{(i)} b)}{\partial b}$

$$= 0 - 0 - y^{(i)}$$

$$= -y^{(i)} \quad (\in \mathbb{R})$$

$$\therefore \nabla_b \mathcal{L} = \frac{1}{K} \sum_{i=1}^{K} \mathbb{I}_{\{y^{(i)}(w^T x^{(i)} + b) < 1\}} \cdot (-y^{(i)})$$

# knn_nosol

January 30, 2024

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

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

Please print out the workbook entirely when completed.

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

## 0.2 Import the appropriate libraries

[16]:
```
import numpy as np # for doing most of our calculations
import matplotlib.pyplot as plt# for plotting
from utils.data_utils import load_CIFAR10 # function to load the CIFAR-10␣
 ↪dataset.

# Load matplotlib images inline
%matplotlib inline

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

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

[17]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = 'C:/Users/User/Desktop/EE247/HW2_Code/hw2_Questions/data/
 ↪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,)
```

[18]:
```
# 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()
```

[19]:
```
# 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.

[20]:
```
# Import the KNN class

from nndl import KNN
```

[21]:
```
# 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) The training process for KNN is just saving data-label pairs

(2)

Pros: The training process is simple and fast since it merely involves remembering the datas

Cons: The process is memory intensive. The more input data we have, the more data we need to store.

### 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.

[22]:
```
# 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')))
```

```
100%|
   | 500/500 [00:21<00:00, 23.39it/s]

Time to run code: 21.377026081085205
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.

[23]:
```
# 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: 0.25002264976501465
Difference in L2 distances between your KNN implementations (should be 0):
1.558978797056501e-09
```

**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.

[24]:
```
# 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.
# ================================================================ #
y_predict = knn.predict_labels(dists_L2_vectorized)
error = np.count_nonzero(y_test-y_predict)/num_test
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

print(error)
```

```
100%|
  | 500/500 [00:00<00:00, 3268.67it/s]

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.

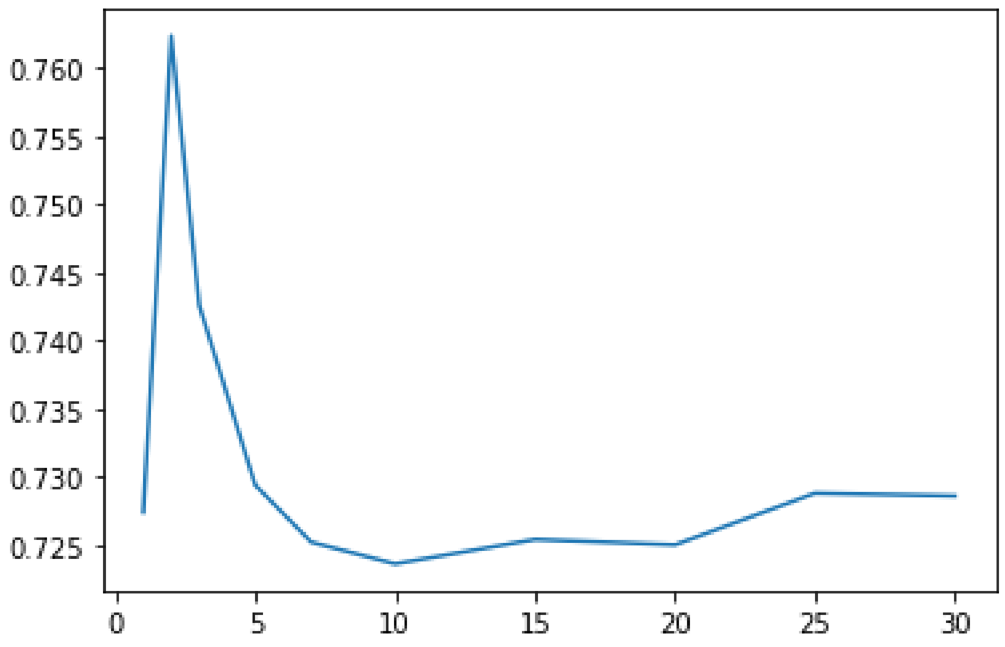
[29]:
```
# Create the dataset folds for cross-valdiation.
num_folds = 5

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]
# ================================================================ #

# randomly assign training img to folds
index_ = np.arange(num_training)
np.random.seed(247)
np.random.shuffle(index_)
index_fold = np.split(index_,num_folds)
train_folds = []

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

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

### 2.0.2 Optimizing the number of nearest neighbors hyperparameter.

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

[31]:
```
from tqdm import tqdm
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.
# ================================================================ #
k_err = np.zeros([len(ks), num_folds])
data_fold = X_train_folds
label_fold = y_train_folds

for cross_val in range(num_folds):
    #first 4 folds train, cross-validate using the last fold
    val_datas = data_fold[-1]
    val_labels = label_fold[-1]
    train_datas = np.vstack(data_fold[0:num_folds-1])
    train_labels = np.asarray(label_fold[0:num_folds-1]).reshape(train_datas.
 ↪shape[0])

    for i, my_k in enumerate(ks):
        knn.train(X=train_datas, y=train_labels)
        dists_L2_vectorized = knn.compute_L2_distances_vectorized(X=val_datas)
        val_predict = knn.predict_labels(dists_L2_vectorized, k=my_k)
        error = np.count_nonzero(val_labels-val_predict)/1000
        k_err[i, cross_val] = error

    # roll as circular shift to select folders for cross validation
    data_fold = np.roll(data_fold, np.prod(data_fold[0].shape))
    label_fold = np.roll(label_fold, data_fold[0].shape[0])

knn_err = np.mean(k_err, axis=1)
plt.plot(ks, knn_err)
print("Lowest k-fold cross validation error: k=", ks[np.argmin(knn_err)])
print("Cross validation error when k=", ks[np.argmin(knn_err)], ": ",␣
 ↪knn_err[np.argmin(knn_err)])

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

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

```
100%|
| 1000/1000 [00:00<00:00, 3804.33it/s]
100%|
| 1000/1000 [00:00<00:00, 3521.13it/s]
100%|
| 1000/1000 [00:00<00:00, 4009.35it/s]
100%|
| 1000/1000 [00:00<00:00, 3960.66it/s]
100%|
| 1000/1000 [00:00<00:00, 3999.56it/s]
100%|
| 1000/1000 [00:00<00:00, 3983.54it/s]
100%|
| 1000/1000 [00:00<00:00, 3999.41it/s]
100%|
| 1000/1000 [00:00<00:00, 4000.00it/s]
100%|
| 1000/1000 [00:00<00:00, 3983.61it/s]
100%|
| 1000/1000 [00:00<00:00, 4000.01it/s]
100%|
| 1000/1000 [00:00<00:00, 4031.81it/s]
100%|
| 1000/1000 [00:00<00:00, 3876.03it/s]
100%|
| 1000/1000 [00:00<00:00, 3999.47it/s]
100%|
| 1000/1000 [00:00<00:00, 3936.61it/s]
100%|
| 1000/1000 [00:00<00:00, 3983.01it/s]
100%|
| 1000/1000 [00:00<00:00, 3920.95it/s]
100%|
| 1000/1000 [00:00<00:00, 4015.56it/s]
100%|
| 1000/1000 [00:00<00:00, 3983.31it/s]
100%|
| 1000/1000 [00:00<00:00, 3750.52it/s]
100%|
| 1000/1000 [00:00<00:00, 3927.03it/s]
100%|
| 1000/1000 [00:00<00:00, 3999.39it/s]
100%|
| 1000/1000 [00:00<00:00, 3968.22it/s]
100%|
| 1000/1000 [00:00<00:00, 3968.27it/s]
100%|
| 1000/1000 [00:00<00:00, 4016.39it/s]
100%|
| 1000/1000 [00:00<00:00, 3905.87it/s]
100%|
| 1000/1000 [00:00<00:00, 3906.61it/s]
100%|
| 1000/1000 [00:00<00:00, 4016.02it/s]
100%|
| 1000/1000 [00:00<00:00, 3952.47it/s]
100%|
 | 1000/1000 [00:00<00:00, 4031.79it/s]
100%|
 | 1000/1000 [00:00<00:00, 3999.99it/s]
100%|
 | 1000/1000 [00:00<00:00, 3983.85it/s]
100%|
 | 1000/1000 [00:00<00:00, 4016.18it/s]
100%|
 | 1000/1000 [00:00<00:00, 3816.80it/s]
100%|
 | 1000/1000 [00:00<00:00, 3968.23it/s]
100%|
 | 1000/1000 [00:00<00:00, 3984.04it/s]
100%|
 | 1000/1000 [00:00<00:00, 3952.13it/s]
100%|
 | 1000/1000 [00:00<00:00, 3860.60it/s]
100%|
 | 1000/1000 [00:00<00:00, 3759.40it/s]
100%|
 | 1000/1000 [00:00<00:00, 3967.86it/s]
100%|
 | 1000/1000 [00:00<00:00, 3891.04it/s]
100%|
 | 1000/1000 [00:00<00:00, 4048.59it/s]
100%|
 | 1000/1000 [00:00<00:00, 3952.85it/s]
100%|
 | 1000/1000 [00:00<00:00, 3983.58it/s]
100%|
 | 1000/1000 [00:00<00:00, 4006.74it/s]
100%|
 | 1000/1000 [00:00<00:00, 3977.60it/s]
100%|
 | 1000/1000 [00:00<00:00, 3968.00it/s]
100%|
 | 1000/1000 [00:00<00:00, 3921.02it/s]
100%|
 | 1000/1000 [00:00<00:00, 3936.55it/s]
100%|
 | 1000/1000 [00:00<00:00, 3983.54it/s]
100%|
 | 1000/1000 [00:00<00:00, 3891.03it/s]

Lowest k-fold cross validation error: k= 10
Cross validation error when k= 10 :  0.7236
Computation time: 29.21
```



### 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) Lowest k-fold cross validation error when k= 10

(2) Cross validation error when k=10 is 0.7236

#### 2.2.1 Optimizing the norm

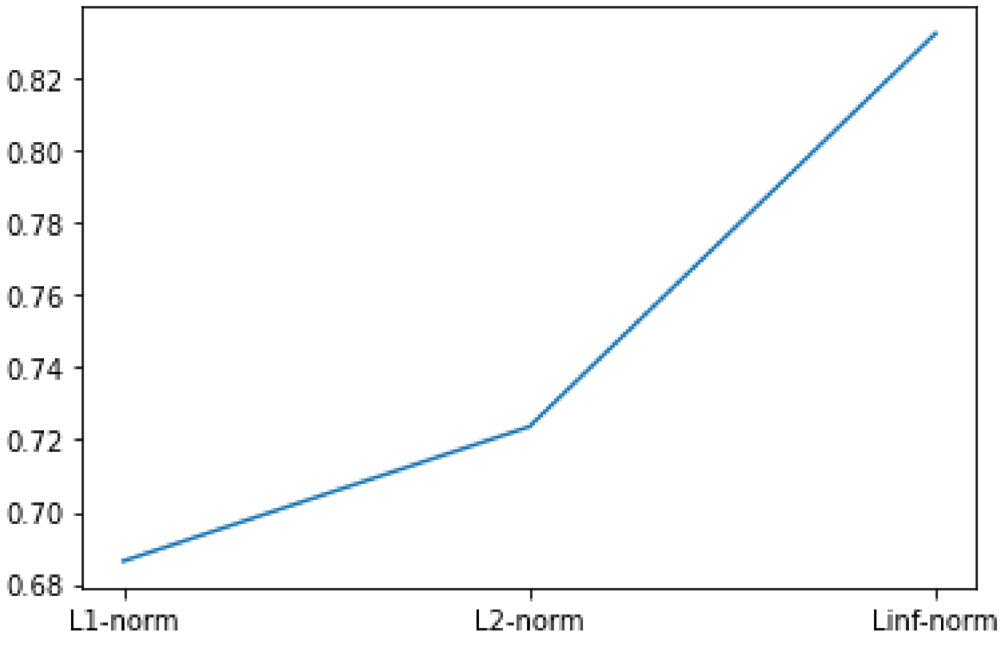
Next, we test three different norms (the 1, 2, and infinity norms) and see which distance metric results in the best cross-validation performance.

[33]:
```
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.
# ================================================================ #

sel_k = ks[np.argmin(knn_err)]
data_fold = X_train_folds
label_fold = y_train_folds
norm_err = np.zeros([len(norms), num_folds])

for cross_val in range(num_folds):
    val_datas = data_fold[-1]
    val_labels = label_fold[-1]
    train_datas = np.vstack(data_fold[0:num_folds-1])
    train_labels = np.asarray(label_fold[0:num_folds-1]).reshape(train_datas.
 ↪shape[0])

    for i, my_norm in enumerate(norms):
        knn.train(X=train_datas, y=train_labels)
        dists_calc = knn.compute_distances(X=val_datas, norm=my_norm)
        val_predict = knn.predict_labels(dists_calc, k=sel_k)
        error = np.count_nonzero(val_labels-val_predict)/1000
        norm_err[i, cross_val] = error

    data_fold = np.roll(data_fold, np.prod(data_fold[0].shape))
    label_fold = np.roll(label_fold, data_fold[0].shape[0])

norms_err = np.mean(norm_err, axis=1)
norm_label = ['L1-norm', 'L2-norm', 'Linf-norm']
plt.plot(norm_label, norms_err)
print("Lowest error using ", norm_label[np.argmin(norms_err)])
print("Cross validation error when using", norm_label[np.argmin(norms_err)], ":␣
 ↪", norms_err[np.argmin(norms_err)])

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

```
100%|
 | 1000/1000 [00:44<00:00, 22.25it/s]
100%|
| 1000/1000 [00:00<00:00, 4877.36it/s]
```

```
100%|
  | 1000/1000 [00:37<00:00, 26.52it/s]
100%|
 | 1000/1000 [00:00<00:00, 4807.67it/s]
100%|
  | 1000/1000 [00:47<00:00, 21.20it/s]
100%|
 | 1000/1000 [00:00<00:00, 6288.20it/s]
100%|
  | 1000/1000 [00:44<00:00, 22.24it/s]
100%|
 | 1000/1000 [00:00<00:00, 4832.09it/s]
100%|
  | 1000/1000 [00:38<00:00, 26.22it/s]
100%|
 | 1000/1000 [00:00<00:00, 4484.38it/s]
100%|
  | 1000/1000 [00:46<00:00, 21.28it/s]
100%|
 | 1000/1000 [00:00<00:00, 6060.69it/s]
100%|
  | 1000/1000 [00:45<00:00, 21.98it/s]
100%|
 | 1000/1000 [00:00<00:00, 4807.12it/s]
100%|
  | 1000/1000 [00:37<00:00, 26.53it/s]
100%|
 | 1000/1000 [00:00<00:00, 4761.15it/s]
100%|
  | 1000/1000 [00:48<00:00, 20.47it/s]
100%|
 | 1000/1000 [00:00<00:00, 5347.62it/s]
100%|
  | 1000/1000 [00:51<00:00, 19.42it/s]
100%|
 | 1000/1000 [00:00<00:00, 4445.29it/s]
100%|
  | 1000/1000 [00:39<00:00, 25.10it/s]
100%|
 | 1000/1000 [00:00<00:00, 4607.69it/s]
100%|
  | 1000/1000 [00:50<00:00, 19.83it/s]
100%|
 | 1000/1000 [00:00<00:00, 6022.88it/s]
100%|
  | 1000/1000 [00:49<00:00, 20.32it/s]
100%|
 | 1000/1000 [00:00<00:00, 4608.35it/s]
```

```
100%|
  | 1000/1000 [00:44<00:00, 22.34it/s]
100%|
 | 1000/1000 [00:00<00:00, 4148.94it/s]
100%|
  | 1000/1000 [00:51<00:00, 19.26it/s]
100%|
 | 1000/1000 [00:00<00:00, 5779.37it/s]

Lowest error using  L1-norm
Cross validation error when using L1-norm :  0.6863999999999999
Computation time: 683.14
```



### 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) Lowest error using L1-norm

(2) Cross validation error when using L1-norm and k=10: 0.6863999999999999

## 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.

[40]:
```
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_calc = knn.compute_distances(X=X_test, norm=L2_norm)
test_predict = knn.predict_labels(dists_calc, k=1)
error = np.count_nonzero(y_test-test_predict)/num_test

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

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

```
100%|
   | 500/500 [00:23<00:00, 21.41it/s]
100%|
  | 500/500 [00:00<00:00, 3758.69it/s]

Error rate achieved: 0.726
```

### 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:

It improves by 0.04, which is very trivial.

```
import numpy as np
import pdb
from tqdm import tqdm

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 tqdm(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] = np.linalg.norm(X[i]-self.X_train[j])

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

    else:
        num_test = X.shape[0]
        num_train = self.X_train.shape[0]
        dists = np.zeros((num_test, num_train))
        for i in tqdm(np.arange(num_test)):

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

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

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

    return dists

def compute_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.
        # ================================================================ #
        # vectorize using 2-norm calculation: ||x_i-y_j||^2 = ||x_i||^2+||y_j||^2-2<x_i,y_j
        # property: ||x_i-y_j||^2 = [x_i-y_j]^T[x_i-y_j]
        train_norm = np.linalg.norm(self.X_train, axis=1)
        train_mat = np.broadcast_to(np.reshape(train_norm, [num_train,1]), \
                                    [num_train,num_test])
        test_norm = np.linalg.norm(X, axis=1)
        test_mat = np.broadcast_to(np.reshape(test_norm, [1, num_test]), \
                                   [num_train,num_test])
        dists = pow(train_mat,2) + pow(test_mat,2) - 2*(np.matmul(self.X_train, X.T))
        dists = np.sqrt(dists.T)
        # ================================================================ #
        # 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 tqdm(np.arange(num_test)):
    # A list of length k storing the labels of the k nearest neighbors to
    # the ith test point.
    closest_y = []

    # ================================================================ #
    # YOUR CODE HERE:
    #   Use the distances to calculate and then store the labels of
    #   the k-nearest neighbors to the ith test point.  The function
    #   numpy.argsort may be useful.
    #
    #   After doing this, find the most common label of the k-nearest
    #   neighbors.  Store the predicted label of the ith training example
    #   as y_pred[i].  Break ties by choosing the smaller label.
    # ================================================================ #
    closest_y = np.argsort(dists[i])[0:k] # select the closest k datas
    y_candidate = self.y_train[closest_y]
    y_choice, counts = np.unique(y_candidate, return_counts=True) # return #count fo
    y_pred[i] = y_choice[np.argmax(counts)]

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

return y_pred
```

# softmax_nosol

January 30, 2024

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

Please follow the notebook linearly to implement a softmax classifier.

Please print out the workbook entirely when completed.

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

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

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

[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 = 'C:/Users/User/Desktop/EE247/HW2_Code/hw2_Questions/data/
 ↪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)
```

```
2.3277607028048757
```

## 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:

This makes sense because our weight is randomly intiallized. It is no different from making a random prediction from the 10 classes, which is why our initial scores are evenly distributed across the 10 classes. As a result, if we plug these scores into the softmax loss function:

$$L_\Theta = -\frac{1}{M}\sum_{i=1}^{M}\log\left(\frac{e^{W_{y(i)}^T x(i)+b_{y(i)}}}{\sum_{j=1}^{c} e^{W_j^T x(j)+b_j}}\right) \approx -log\left(\frac{1}{10}\right) \approx 2.30258509$$

It's close to 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)
```

```
100%|
 | 500/500 [00:00<00:00, 1953.33it/s]

numerical: 0.748466 analytic: 0.748466, relative error: 1.147876e-09
numerical: 0.655100 analytic: 0.655100, relative error: 1.911980e-08
numerical: 0.290715 analytic: 0.290715, relative error: 3.430144e-08
numerical: 0.025306 analytic: 0.025306, relative error: 1.004345e-06
numerical: 0.590186 analytic: 0.590186, relative error: 4.540223e-08
numerical: 1.854029 analytic: 1.854029, relative error: 8.706384e-09
numerical: -2.489010 analytic: -2.489010, relative error: 2.169692e-08
numerical: -2.607978 analytic: -2.607978, relative error: 6.218801e-09
numerical: 1.533928 analytic: 1.533928, relative error: 6.156722e-09
numerical: -4.615755 analytic: -4.615755, relative error: 7.355396e-09
```

## 0.5 A vectorized version of Softmax

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

[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.
```

```
100%|
 | 500/500 [00:00<00:00, 1968.76it/s]

Normal loss / grad_norm: 2.331676097115445 / 333.3972385007976 computed in
0.25699782371520996s
Vectorized loss / grad: 2.331676097115445 / 333.3972385007976 computed in
0.0070035457611083984s
difference in loss / grad: 0.0 /2.2278613260596803e-13
```

## 0.6 Stochastic gradient descent

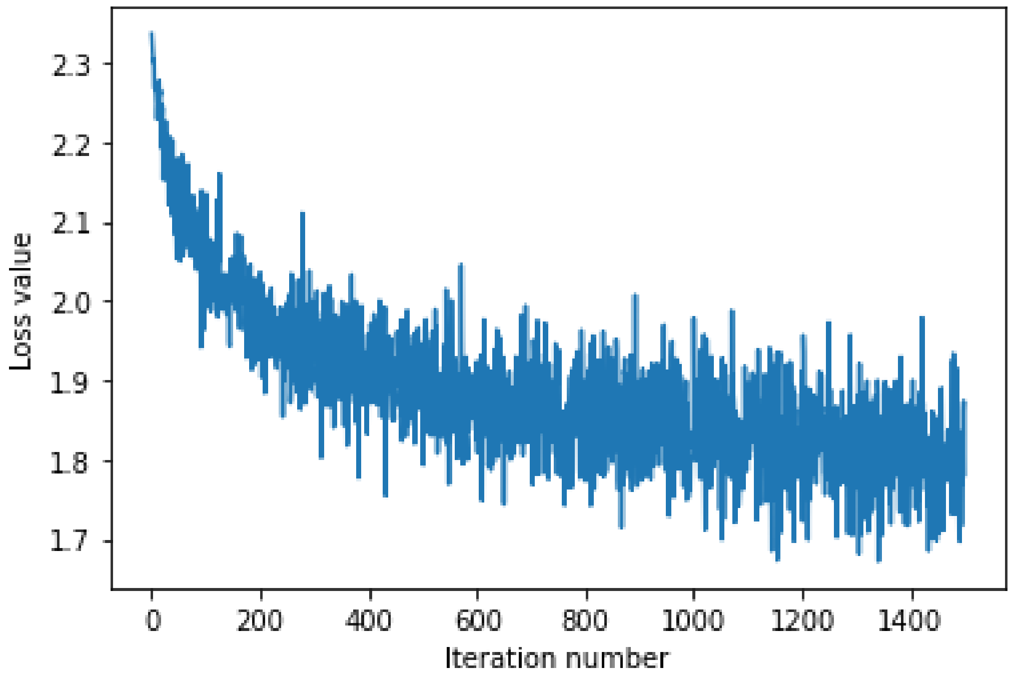
We now implement stochastic gradient descent. This uses the same principles of gradient descent we discussed in class, however, it calculates the gradient by only using examples from a subset of the training set (so each gradient calculation is faster).

[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()
```

```
  1%|
| 22/1500 [00:00<00:06, 220.00it/s]

iteration 0 / 1500: loss 2.3365926606637544

  9%|
| 141/1500 [00:00<00:05, 239.90it/s]

iteration 100 / 1500: loss 2.0557222613850827
```

```
 16%|
| 245/1500 [00:01<00:05, 242.99it/s]

iteration 200 / 1500: loss 2.0357745120662813

 21%|
| 321/1500 [00:01<00:05, 234.49it/s]

iteration 300 / 1500: loss 1.9813348165609888

 29%|
| 439/1500 [00:01<00:04, 223.38it/s]

iteration 400 / 1500: loss 1.9583142443981612

 36%|
| 533/1500 [00:02<00:04, 230.45it/s]

iteration 500 / 1500: loss 1.8622653073541355

 42%|
| 632/1500 [00:02<00:03, 239.72it/s]

iteration 600 / 1500: loss 1.8532611454359382

 49%|
| 739/1500 [00:03<00:02, 256.69it/s]

iteration 700 / 1500: loss 1.8353062223725827

 56%|
| 841/1500 [00:03<00:02, 239.48it/s]

iteration 800 / 1500: loss 1.829389246882764

 63%|
| 944/1500 [00:04<00:02, 248.15it/s]

iteration 900 / 1500: loss 1.8992158530357484

 70%|
| 1048/1500 [00:04<00:01, 241.82it/s]

iteration 1000 / 1500: loss 1.9783503540252303

 77%|
| 1149/1500 [00:04<00:01, 240.39it/s]

iteration 1100 / 1500: loss 1.8470797913532633

 82%|
| 1229/1500 [00:05<00:01, 254.95it/s]

iteration 1200 / 1500: loss 1.8411450268664082

 89%|
| 1331/1500 [00:05<00:00, 237.61it/s]

iteration 1300 / 1500: loss 1.7910402495792102
```

```
 96%|
| 1435/1500 [00:06<00:00, 249.36it/s]

iteration 1400 / 1500: loss 1.8705803029382257

100%|
 | 1500/1500 [00:06<00:00, 239.73it/s]

That took 6.259000301361084s
```



### 0.6.1 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
```

## 0.7 Optimize the softmax classifier

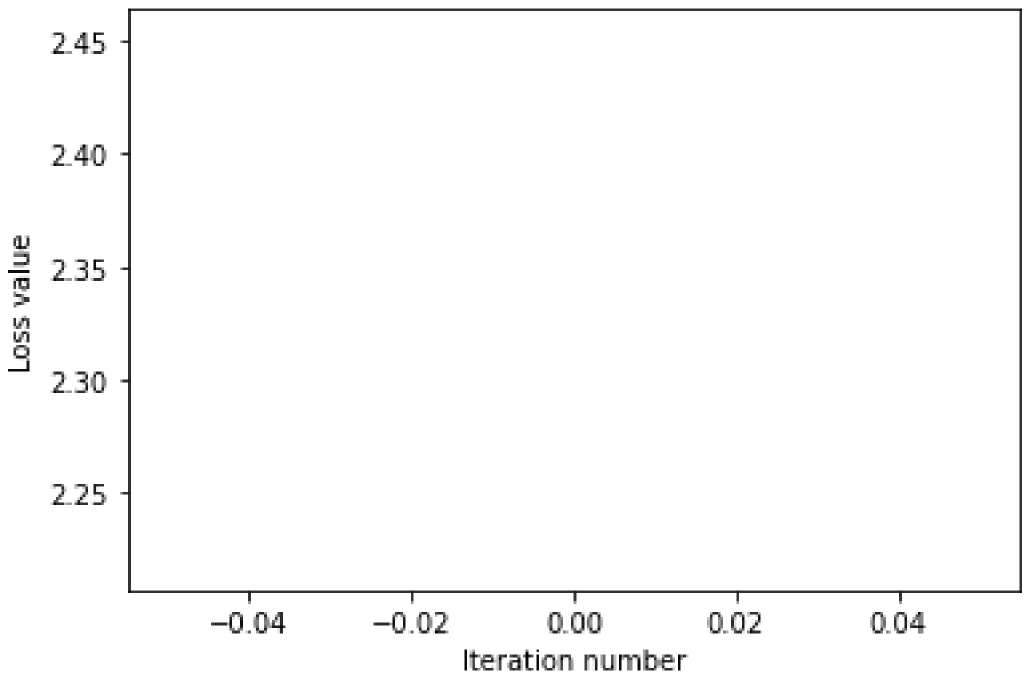
[12]:
```
np.finfo(float).eps
```

[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.
# ================================================================ #

lr_list = [1e-3, 1e-4, 3e-4, 1e-5, 5e-5, 5e-7, 1e-7, 5e-8, 1e-8]
train_pred_ls = []
val_pred_ls = []

for lr in lr_list:
    tic = time.time()
    loss_hist = softmax.train(X_train, y_train, learning_rate=lr,
                              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()

    y_train_pred = softmax.predict(X_train)
    print('training accuracy: {}'.format(np.mean(np.
 ↪equal(y_train,y_train_pred), )))
    train_pred_ls.append(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)), ))
    val_pred_ls.append(np.mean(np.equal(y_val,y_val_pred)))


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

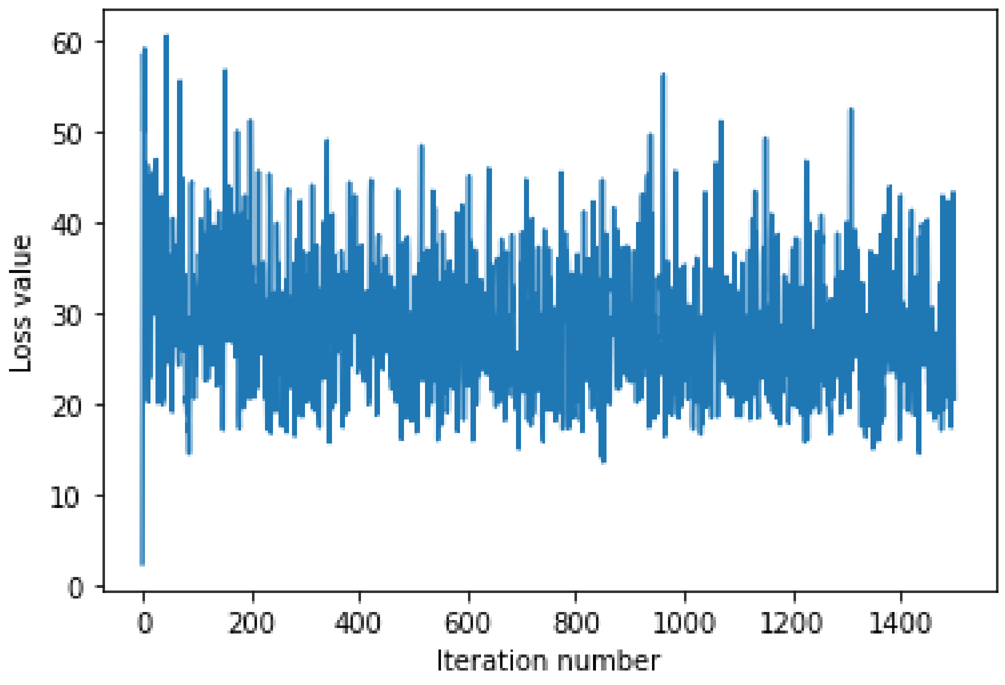
```
  0%|
| 0/1500 [00:00<?, ?it/s]

iteration 0 / 1500: loss 2.3353835450891545

C:\Users\User\Desktop\EE247\HW2_Code\hw2_Questions\code\nndl\softmax.py:145:
RuntimeWarning: overflow encountered in exp
  all_score = np.log(np.sum(np.exp(softmax_scores), axis=1))-true_scores
C:\Users\User\Desktop\EE247\HW2_Code\hw2_Questions\code\nndl\softmax.py:151:
RuntimeWarning: overflow encountered in exp
  grad_ = np.divide(np.exp(np.matmul(self.W, X.T)),
np.sum(np.exp(np.matmul(self.W, X.T)), axis=0))
C:\Users\User\Desktop\EE247\HW2_Code\hw2_Questions\code\nndl\softmax.py:151:
RuntimeWarning: invalid value encountered in true_divide
  grad_ = np.divide(np.exp(np.matmul(self.W, X.T)),
np.sum(np.exp(np.matmul(self.W, X.T)), axis=0))
  9%|
| 142/1500 [00:00<00:05, 229.35it/s]

iteration 100 / 1500: loss nan

 16%|
| 236/1500 [00:01<00:05, 212.74it/s]

iteration 200 / 1500: loss nan

 22%|
| 331/1500 [00:01<00:05, 229.81it/s]

iteration 300 / 1500: loss nan

 29%|
| 431/1500 [00:01<00:04, 239.31it/s]

iteration 400 / 1500: loss nan

 36%|
| 538/1500 [00:02<00:03, 254.80it/s]

iteration 500 / 1500: loss nan

 42%|
| 636/1500 [00:02<00:03, 224.35it/s]

iteration 600 / 1500: loss nan

 49%|
| 738/1500 [00:03<00:03, 243.21it/s]

iteration 700 / 1500: loss nan

 54%|
| 816/1500 [00:03<00:02, 251.32it/s]
```

```
iteration 800 / 1500: loss nan

 62%|
| 927/1500 [00:04<00:03, 188.49it/s]

iteration 900 / 1500: loss nan

 68%|
| 1021/1500 [00:04<00:02, 220.30it/s]

iteration 1000 / 1500: loss nan

 76%|
| 1135/1500 [00:05<00:01, 216.86it/s]

iteration 1100 / 1500: loss nan

 82%|
| 1237/1500 [00:05<00:01, 242.17it/s]

iteration 1200 / 1500: loss nan

 89%|
| 1328/1500 [00:05<00:00, 204.84it/s]

iteration 1300 / 1500: loss nan

 95%|
| 1427/1500 [00:06<00:00, 235.78it/s]

iteration 1400 / 1500: loss nan

100%|
 | 1500/1500 [00:06<00:00, 223.25it/s]

That took 6.72200083732605s
```



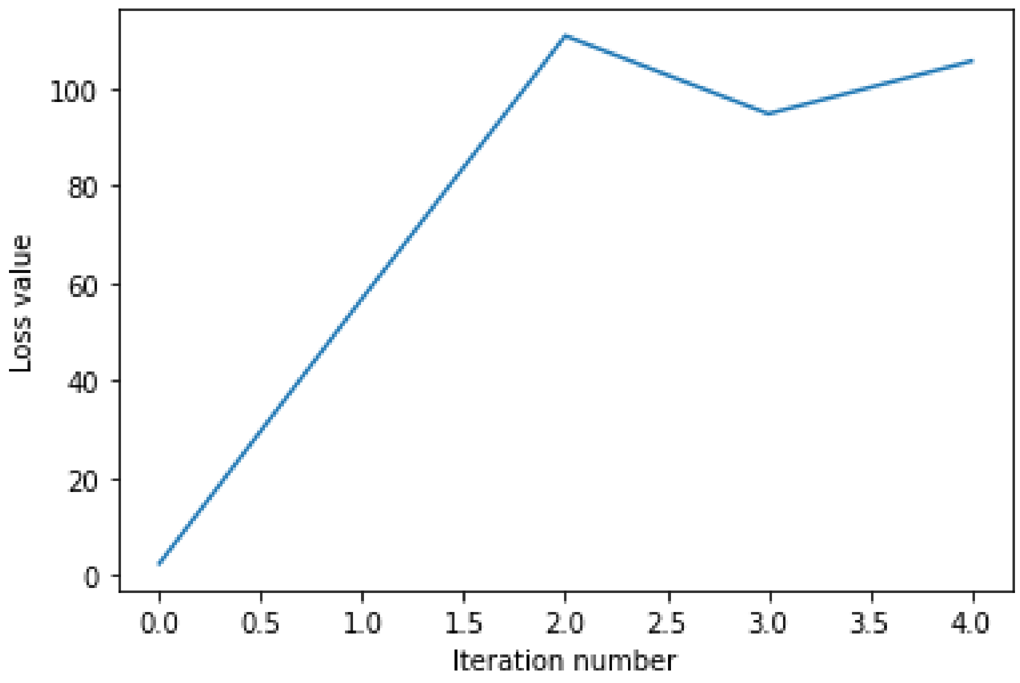
```
training accuracy: 0.10026530612244898
validation accuracy: 0.087

  3%|
| 48/1500 [00:00<00:06, 237.85it/s]

iteration 0 / 1500: loss 2.4615346985497166

  8%|
| 124/1500 [00:00<00:06, 228.41it/s]

iteration 100 / 1500: loss 28.333379080471175

 16%|
| 236/1500 [00:01<00:06, 209.54it/s]

iteration 200 / 1500: loss 44.467116668405716

 23%|
| 338/1500 [00:01<00:04, 242.22it/s]

iteration 300 / 1500: loss 30.121347387504283

 29%|
| 440/1500 [00:01<00:04, 241.02it/s]

iteration 400 / 1500: loss 26.65671169362472

 36%|
| 543/1500 [00:02<00:03, 246.26it/s]
```

```
iteration 500 / 1500: loss 29.871015294984183

 43%|
| 643/1500 [00:02<00:03, 233.93it/s]

iteration 600 / 1500: loss 25.05655491168399

 50%|
| 745/1500 [00:03<00:03, 239.28it/s]

iteration 700 / 1500: loss 28.213828424279914

 56%|
| 845/1500 [00:03<00:02, 235.26it/s]

iteration 800 / 1500: loss 26.455162324427256

 63%|
| 946/1500 [00:04<00:02, 242.96it/s]

iteration 900 / 1500: loss 32.63925656068801

 70%|
| 1044/1500 [00:04<00:01, 229.65it/s]

iteration 1000 / 1500: loss 27.48861483969415

 76%|
| 1143/1500 [00:04<00:01, 237.28it/s]

iteration 1100 / 1500: loss 32.170042270825654

 83%|
| 1241/1500 [00:05<00:01, 239.12it/s]

iteration 1200 / 1500: loss 37.02528923306888

 89%|
| 1338/1500 [00:05<00:00, 218.44it/s]

iteration 1300 / 1500: loss 39.957702753376324

 96%|
| 1437/1500 [00:06<00:00, 236.60it/s]

iteration 1400 / 1500: loss 42.899605058729776

100%|
 | 1500/1500 [00:06<00:00, 232.56it/s]

That took 6.451997756958008s
```



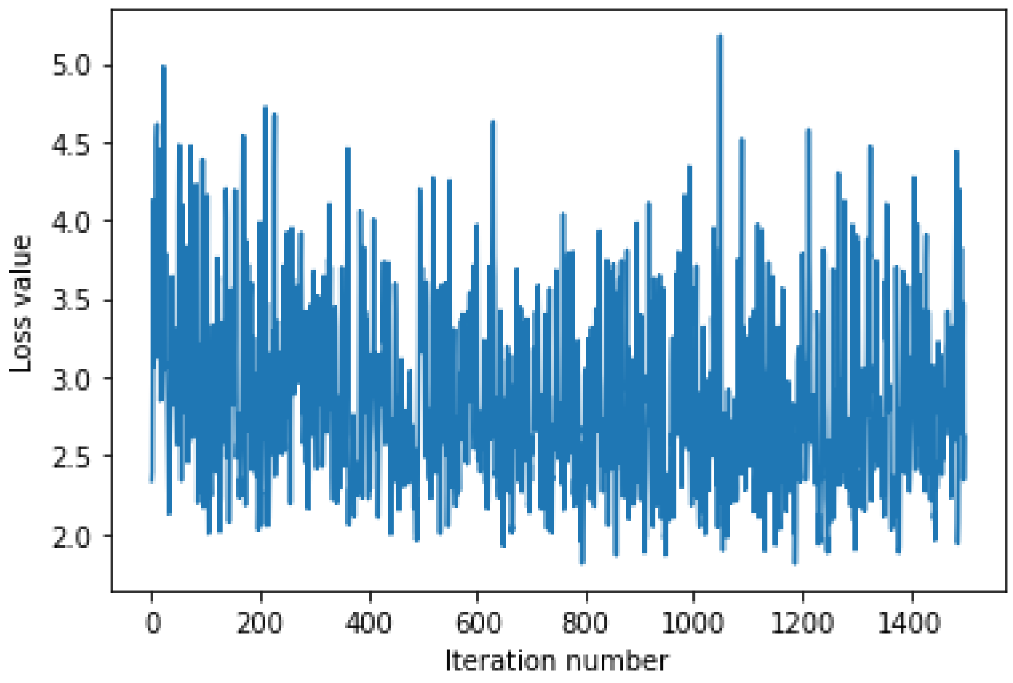
```
training accuracy: 0.2930204081632653
validation accuracy: 0.281

  1%|
| 19/1500 [00:00<00:07, 186.28it/s]

iteration 0 / 1500: loss 2.3790388831757823

 10%|
| 143/1500 [00:00<00:05, 242.51it/s]

iteration 100 / 1500: loss nan

 15%|
| 222/1500 [00:00<00:05, 234.01it/s]

iteration 200 / 1500: loss nan

 23%|
| 343/1500 [00:01<00:04, 234.13it/s]

iteration 300 / 1500: loss nan

 28%|
| 421/1500 [00:01<00:04, 248.57it/s]

iteration 400 / 1500: loss nan

 36%|
| 545/1500 [00:02<00:04, 236.11it/s]
```

```
iteration 500 / 1500: loss nan

 42%|
| 637/1500 [00:02<00:04, 212.64it/s]

iteration 600 / 1500: loss nan

 49%|
| 732/1500 [00:03<00:03, 231.43it/s]

iteration 700 / 1500: loss nan

 55%|
| 831/1500 [00:03<00:02, 235.55it/s]

iteration 800 / 1500: loss nan

 62%|
| 931/1500 [00:04<00:02, 233.59it/s]

iteration 900 / 1500: loss nan

 69%|
| 1028/1500 [00:04<00:02, 227.03it/s]

iteration 1000 / 1500: loss nan

 75%|
| 1127/1500 [00:04<00:01, 239.53it/s]

iteration 1100 / 1500: loss nan

 82%|
| 1229/1500 [00:05<00:01, 244.59it/s]

iteration 1200 / 1500: loss nan

 89%|
| 1329/1500 [00:05<00:00, 221.54it/s]

iteration 1300 / 1500: loss nan

 95%|
| 1427/1500 [00:06<00:00, 233.05it/s]

iteration 1400 / 1500: loss nan

100%|
 | 1500/1500 [00:06<00:00, 232.41it/s]

That took 6.456999063491821s
```



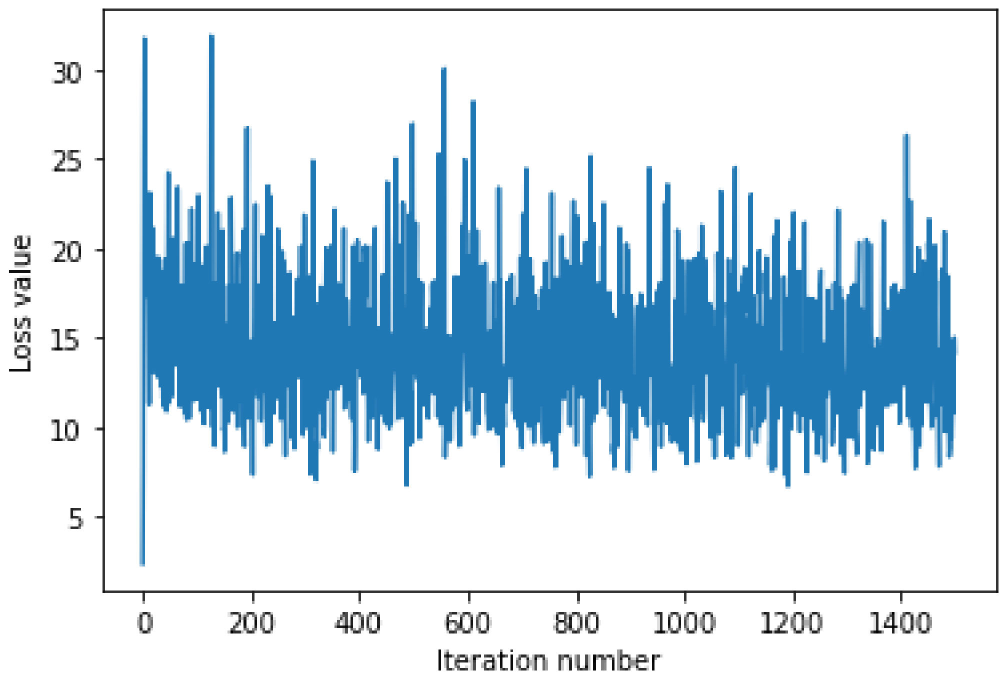
```
training accuracy: 0.10026530612244898
validation accuracy: 0.087

  3%|
| 41/1500 [00:00<00:07, 204.05it/s]

iteration 0 / 1500: loss 2.338799247622096

  9%|
| 139/1500 [00:00<00:05, 239.37it/s]

iteration 100 / 1500: loss 2.3741638193976007

 16%|
| 236/1500 [00:01<00:05, 233.56it/s]

iteration 200 / 1500: loss 3.986164950939759

 22%|
| 332/1500 [00:01<00:05, 229.84it/s]

iteration 300 / 1500: loss 2.9911038436545003

 29%|
| 436/1500 [00:01<00:04, 250.05it/s]

iteration 400 / 1500: loss 2.2226778474682676

 36%|
| 539/1500 [00:02<00:04, 228.03it/s]
```

```
iteration 500 / 1500: loss 3.4665937843255636

 42%|
| 637/1500 [00:02<00:03, 239.94it/s]

iteration 600 / 1500: loss 3.9698696636064796

 49%|
| 737/1500 [00:03<00:03, 241.03it/s]

iteration 700 / 1500: loss 2.3364296345441784

 56%|
| 834/1500 [00:03<00:02, 233.28it/s]

iteration 800 / 1500: loss 3.052528247098768

 62%|
| 937/1500 [00:03<00:02, 247.81it/s]

iteration 900 / 1500: loss 2.3747833366348776

 69%|
| 1037/1500 [00:04<00:02, 228.62it/s]

iteration 1000 / 1500: loss 2.286946186499525

 76%|
| 1135/1500 [00:04<00:01, 234.57it/s]

iteration 1100 / 1500: loss 2.5187840731040905

 82%|
| 1227/1500 [00:05<00:01, 218.08it/s]

iteration 1200 / 1500: loss 2.971672432981369

 88%|
| 1325/1500 [00:05<00:00, 224.04it/s]

iteration 1300 / 1500: loss 2.4157320985896233

 95%|
| 1427/1500 [00:06<00:00, 243.41it/s]

iteration 1400 / 1500: loss 2.6727410576732864

100%|
 | 1500/1500 [00:06<00:00, 230.13it/s]

That took 6.521001100540161s
```



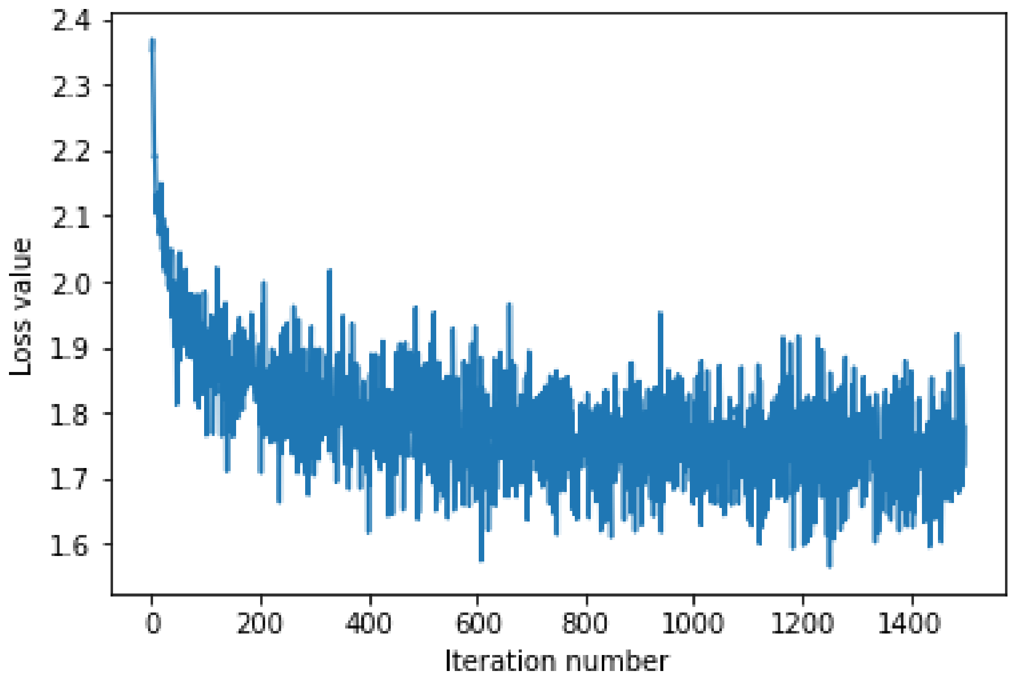
```
training accuracy: 0.2886734693877551
validation accuracy: 0.288

  1%|
| 22/1500 [00:00<00:06, 215.69it/s]

iteration 0 / 1500: loss 2.372474398135772

 10%|
| 149/1500 [00:00<00:05, 238.98it/s]

iteration 100 / 1500: loss 13.897071933368801

 17%|
| 251/1500 [00:01<00:05, 246.92it/s]

iteration 200 / 1500: loss 13.241459651160735

 22%|
| 326/1500 [00:01<00:05, 232.51it/s]

iteration 300 / 1500: loss 17.95197875661463

 30%|
| 451/1500 [00:01<00:04, 244.43it/s]

iteration 400 / 1500: loss 16.513838645503494

 35%|
| 527/1500 [00:02<00:04, 237.34it/s]
```

```
iteration 500 / 1500: loss 13.05491071755

 44%|
| 653/1500 [00:02<00:03, 244.65it/s]

iteration 600 / 1500: loss 11.76007111101514

 49%|
| 730/1500 [00:03<00:03, 250.68it/s]

iteration 700 / 1500: loss 8.85100209452067

 55%|
| 832/1500 [00:03<00:03, 221.20it/s]

iteration 800 / 1500: loss 9.024660329072722

 62%|
| 935/1500 [00:03<00:02, 243.92it/s]

iteration 900 / 1500: loss 14.342883785765649

 69%|
| 1036/1500 [00:04<00:02, 223.63it/s]

iteration 1000 / 1500: loss 16.167245583163112

 76%|
| 1133/1500 [00:04<00:01, 227.63it/s]

iteration 1100 / 1500: loss 11.086608563042825

 82%|
| 1234/1500 [00:05<00:01, 243.47it/s]

iteration 1200 / 1500: loss 16.522346197676306

 89%|
| 1332/1500 [00:05<00:00, 224.76it/s]

iteration 1300 / 1500: loss 12.822429696970728

 95%|
| 1432/1500 [00:06<00:00, 240.56it/s]

iteration 1400 / 1500: loss 12.195772866338547

100%|
 | 1500/1500 [00:06<00:00, 234.41it/s]

That took 6.402997970581055s
```



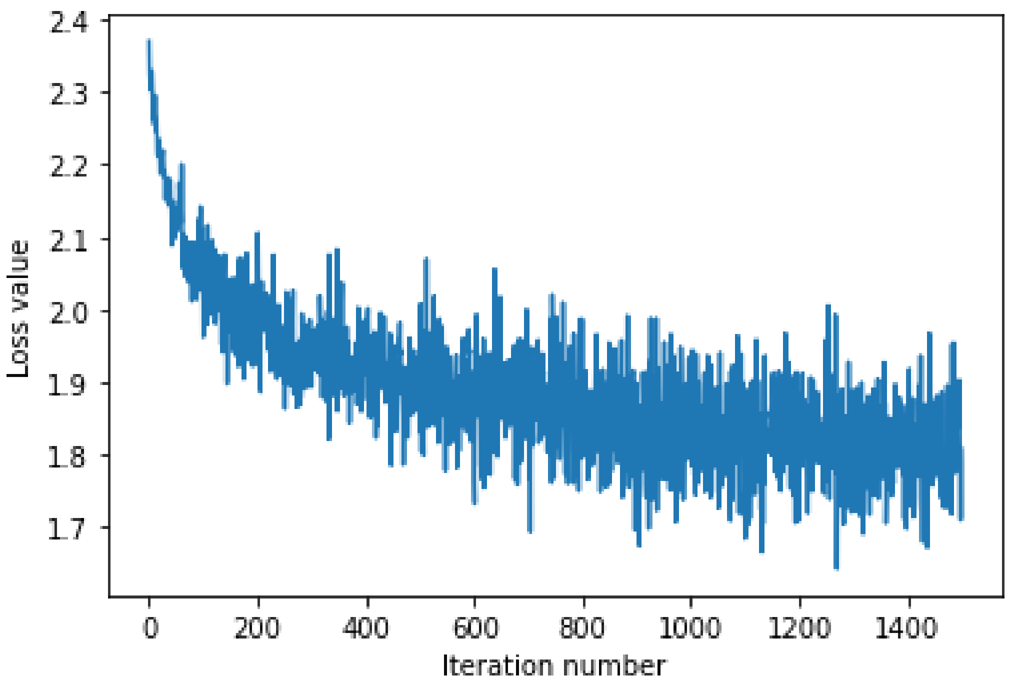
```
training accuracy: 0.30583673469387757
validation accuracy: 0.284

  1%|
| 22/1500 [00:00<00:06, 217.82it/s]

iteration 0 / 1500: loss 2.352610559430411

 10%|
| 148/1500 [00:00<00:05, 241.24it/s]

iteration 100 / 1500: loss 1.8923521357155466

 16%|
| 246/1500 [00:01<00:05, 237.20it/s]

iteration 200 / 1500: loss 1.885593486312816

 23%|
| 345/1500 [00:01<00:04, 243.55it/s]

iteration 300 / 1500: loss 1.7045096526258607

 30%|
| 444/1500 [00:01<00:04, 225.94it/s]

iteration 400 / 1500: loss 1.6173224084287392

 36%|
| 540/1500 [00:02<00:04, 227.98it/s]
```

```
iteration 500 / 1500: loss 1.869431356286612

 42%|
| 628/1500 [00:02<00:04, 201.96it/s]

iteration 600 / 1500: loss 1.7657805731100182

 48%|
| 723/1500 [00:03<00:03, 226.12it/s]

iteration 700 / 1500: loss 1.7357334850225437

 54%|
| 816/1500 [00:03<00:03, 206.00it/s]

iteration 800 / 1500: loss 1.7927984357943871

 63%|
| 946/1500 [00:04<00:02, 213.11it/s]

iteration 900 / 1500: loss 1.7305677909021744

 69%|
| 1039/1500 [00:04<00:02, 220.57it/s]

iteration 1000 / 1500: loss 1.7201312704775944

 75%|
| 1130/1500 [00:05<00:01, 218.59it/s]

iteration 1100 / 1500: loss 1.6376163715831091

 82%|
| 1224/1500 [00:05<00:01, 229.76it/s]

iteration 1200 / 1500: loss 1.792025354690927

 89%|
| 1339/1500 [00:06<00:00, 218.01it/s]

iteration 1300 / 1500: loss 1.7594356481750788

 95%|
| 1428/1500 [00:06<00:00, 214.88it/s]

iteration 1400 / 1500: loss 1.7379950131523725

100%|
 | 1500/1500 [00:06<00:00, 218.37it/s]

That took 6.871999025344849s
```



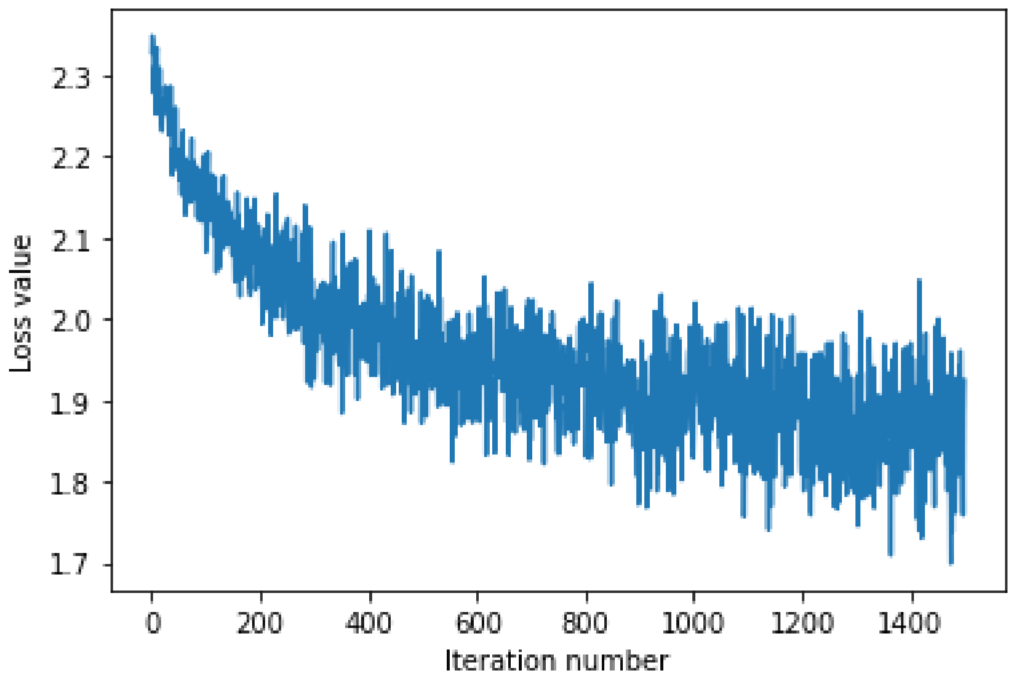
```
training accuracy: 0.4083469387755102
validation accuracy: 0.394

  2%|
| 23/1500 [00:00<00:06, 223.30it/s]

iteration 0 / 1500: loss 2.3704882876562903

  9%|
| 134/1500 [00:00<00:06, 196.90it/s]

iteration 100 / 1500: loss 1.9615720278025612

 15%|
| 222/1500 [00:01<00:06, 210.76it/s]

iteration 200 / 1500: loss 1.966157039029589

 23%|
| 342/1500 [00:01<00:05, 220.88it/s]

iteration 300 / 1500: loss 1.893061028904175

 29%|
| 434/1500 [00:02<00:04, 220.89it/s]

iteration 400 / 1500: loss 1.9193743432735466

 35%|
| 528/1500 [00:02<00:04, 224.85it/s]
```

```
iteration 500 / 1500: loss 1.8489336263191047

 41%|
| 621/1500 [00:02<00:03, 220.41it/s]

iteration 600 / 1500: loss 1.7616429389971195

 49%|
| 741/1500 [00:03<00:03, 231.66it/s]

iteration 700 / 1500: loss 1.9389588615682165

 56%|
| 835/1500 [00:03<00:03, 219.54it/s]

iteration 800 / 1500: loss 1.8212064159430719

 62%|
| 926/1500 [00:04<00:02, 220.22it/s]

iteration 900 / 1500: loss 1.8434593580954894

 69%|
| 1041/1500 [00:04<00:02, 210.51it/s]

iteration 1000 / 1500: loss 1.8234126964890527

 75%|
| 1130/1500 [00:05<00:01, 216.62it/s]

iteration 1100 / 1500: loss 1.6828502030460524

 82%|
| 1224/1500 [00:05<00:01, 220.79it/s]

iteration 1200 / 1500: loss 1.795181692912701

 90%|
| 1343/1500 [00:06<00:00, 231.76it/s]

iteration 1300 / 1500: loss 1.7686206743770954

 96%|
| 1438/1500 [00:06<00:00, 227.13it/s]

iteration 1400 / 1500: loss 1.8648584004789652

100%|
 | 1500/1500 [00:06<00:00, 219.07it/s]

That took 6.849001407623291s
```



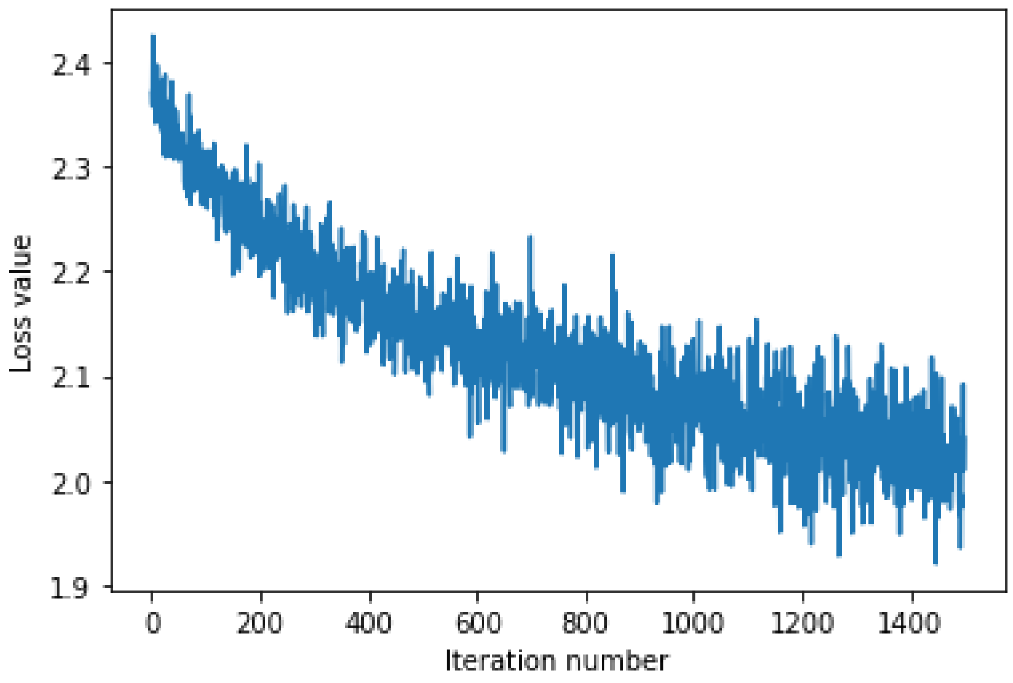
```
training accuracy: 0.38210204081632654
validation accuracy: 0.382

  1%|
| 20/1500 [00:00<00:07, 200.00it/s]

iteration 0 / 1500: loss 2.327719807960521

 10%|
| 150/1500 [00:00<00:06, 217.16it/s]

iteration 100 / 1500: loss 2.0814123988651416

 15%|
| 220/1500 [00:01<00:05, 216.67it/s]

iteration 200 / 1500: loss 2.0583251558453024

 22%|
| 332/1500 [00:01<00:05, 209.78it/s]

iteration 300 / 1500: loss 1.9260348132158982

 30%|
| 451/1500 [00:02<00:04, 231.12it/s]

iteration 400 / 1500: loss 2.0302523445811245

 35%|
| 522/1500 [00:02<00:04, 229.14it/s]
```

```
iteration 500 / 1500: loss 1.9393463560003645

 43%|
| 639/1500 [00:02<00:03, 224.61it/s]

iteration 600 / 1500: loss 1.9694934457422946

 49%|
| 733/1500 [00:03<00:03, 227.62it/s]

iteration 700 / 1500: loss 1.8516708602453769

 56%|
| 843/1500 [00:03<00:03, 204.61it/s]

iteration 800 / 1500: loss 1.9931814441122548

 62%|
| 934/1500 [00:04<00:02, 219.89it/s]

iteration 900 / 1500: loss 1.875279573668774

 68%|
| 1026/1500 [00:04<00:02, 212.25it/s]

iteration 1000 / 1500: loss 1.861260906570181

 76%|
| 1147/1500 [00:05<00:01, 231.29it/s]

iteration 1100 / 1500: loss 1.8211231016268399

 83%|
| 1241/1500 [00:05<00:01, 221.89it/s]

iteration 1200 / 1500: loss 1.9104875667607737

 89%|
| 1336/1500 [00:06<00:00, 224.35it/s]

iteration 1300 / 1500: loss 1.8760559325075774

 95%|
| 1431/1500 [00:06<00:00, 221.20it/s]

iteration 1400 / 1500: loss 1.9703829123833378

100%|
 | 1500/1500 [00:06<00:00, 217.93it/s]

That took 6.886999607086182s
```



```
training accuracy: 0.36081632653061224
validation accuracy: 0.381

  2%|
| 23/1500 [00:00<00:06, 219.05it/s]

iteration 0 / 1500: loss 2.3700514725434676

 10%|
| 143/1500 [00:00<00:05, 234.15it/s]

iteration 100 / 1500: loss 2.2948902797546786

 16%|
| 235/1500 [00:01<00:05, 219.65it/s]

iteration 200 / 1500: loss 2.240713017937215

 23%|
| 346/1500 [00:01<00:05, 210.75it/s]

iteration 300 / 1500: loss 2.2154300766713093

 29%|
| 441/1500 [00:02<00:04, 224.04it/s]

iteration 400 / 1500: loss 2.1397275885668647

 35%|
| 531/1500 [00:02<00:04, 211.57it/s]
```

```
iteration 500 / 1500: loss 2.182694190765592

 42%|
| 627/1500 [00:02<00:03, 227.03it/s]

iteration 600 / 1500: loss 2.123442503042313

 48%|
| 720/1500 [00:03<00:03, 206.43it/s]

iteration 700 / 1500: loss 2.1104598469681353

 55%|
| 823/1500 [00:03<00:03, 194.98it/s]

iteration 800 / 1500: loss 2.1330386022661925

 61%|
| 917/1500 [00:04<00:02, 205.07it/s]

iteration 900 / 1500: loss 2.0592562541503034

 69%|
| 1035/1500 [00:04<00:02, 229.20it/s]

iteration 1000 / 1500: loss 2.1081171973100674

 75%|
| 1132/1500 [00:05<00:01, 231.86it/s]

iteration 1100 / 1500: loss 2.001955147869341

 83%|
| 1244/1500 [00:05<00:01, 215.79it/s]

iteration 1200 / 1500: loss 2.061513357632753

 89%|
| 1337/1500 [00:06<00:00, 223.59it/s]

iteration 1300 / 1500: loss 2.0458751725433

 95%|
| 1421/1500 [00:06<00:00, 182.62it/s]

iteration 1400 / 1500: loss 2.024285435154474

100%|
 | 1500/1500 [00:07<00:00, 212.07it/s]

That took 7.074998140335083s
```



```
training accuracy: 0.2883469387755102
validation accuracy: 0.298
```

[14]:
```
print(train_pred_ls)
print(val_pred_ls)
print("Best training result when using lr=: ", lr_list[np.argmax(val_pred_ls)])
```

```
[0.10026530612244898, 0.2930204081632653, 0.10026530612244898,
0.2886734693877551, 0.30583673469387757, 0.4083469387755102,
0.38210204081632654, 0.36081632653061224, 0.2883469387755102]
[0.087, 0.281, 0.087, 0.288, 0.284, 0.394, 0.382, 0.381, 0.298]
Best training result when using lr=:  5e-07
```

[15]:
```
loss_hist = softmax.train(X_train, y_train, learning_rate=5e-07,
                      num_iters=1500, verbose=True)

y_test_pred = softmax.predict(X_test)
print('test accuracy: {}'.format(np.mean(np.equal(y_test, y_test_pred)), ))
```

```
  0%|
| 0/1500 [00:00<?, ?it/s]

iteration 0 / 1500: loss 2.346281546988113

  9%|
| 129/1500 [00:00<00:06, 218.96it/s]

iteration 100 / 1500: loss 1.8285350488817005
```

```
 16%|
| 243/1500 [00:01<00:05, 216.66it/s]

iteration 200 / 1500: loss 1.8974076699053233

 22%|
| 337/1500 [00:01<00:05, 229.85it/s]

iteration 300 / 1500: loss 1.7764950316699921

 29%|
| 432/1500 [00:02<00:04, 226.15it/s]

iteration 400 / 1500: loss 1.787830988365825

 35%|
| 528/1500 [00:02<00:04, 221.95it/s]

iteration 500 / 1500: loss 1.9631496466715332

 41%|
| 620/1500 [00:02<00:04, 211.99it/s]

iteration 600 / 1500: loss 1.8281237155120293

 49%|
| 733/1500 [00:03<00:03, 217.83it/s]

iteration 700 / 1500: loss 1.8324179773063123

 55%|
| 826/1500 [00:03<00:03, 215.26it/s]

iteration 800 / 1500: loss 1.8485081197252382

 63%|
| 946/1500 [00:04<00:02, 233.40it/s]

iteration 900 / 1500: loss 1.7787622723473455

 69%|
| 1042/1500 [00:04<00:02, 224.26it/s]

iteration 1000 / 1500: loss 1.7673749217879506

 76%|
| 1137/1500 [00:05<00:01, 227.38it/s]

iteration 1100 / 1500: loss 1.7690518250187772

 82%|
| 1225/1500 [00:05<00:01, 204.11it/s]

iteration 1200 / 1500: loss 1.7145892556896916

 89%|
| 1337/1500 [00:06<00:00, 215.03it/s]

iteration 1300 / 1500: loss 1.720861529519568
```

```
 95%|
| 1427/1500 [00:06<00:00, 201.28it/s]

iteration 1400 / 1500: loss 1.7244791752646567

100%|
 | 1500/1500 [00:07<00:00, 212.92it/s]

test accuracy: 0.398
```

[34]:
```
print('CIFAR-10 test accuracy using Softmax (lr=5e-07):', np.round(np.mean(np.
 ↪equal(y_test, y_test_pred)), 4)*100, "%")
```

```
CIFAR-10 test accuracy using Softmax (lr=5e-07): 39.800000000000004 %
```

[ ]:
```

```

```
import numpy as np
from tqdm import tqdm

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.)
        # ================================================================ #
        # X: (N, D); N as batch-size, D as feature dim
        # W: (C, D); C as #classes
    # y: (N, ); training label

    batch_size = X.shape[0]
    softmax_scores = np.matmul(X, self.W.T) # (N, C)
    true_index = np.vstack([np.arange(X.shape[0]), y]).T # index map (N,2)
    row_indices, col_indices = true_index[:, 0], true_index[:, 1]
    true_scores = softmax_scores[row_indices, col_indices] # y_i's for imgs (N,1)
    all_score = np.log(np.sum(np.exp(softmax_scores), axis=1))-true_scores
    loss = np.sum(all_score, axis=0)/batch_size

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

    batch_size = X.shape[0]
    softmax_scores = np.matmul(X, self.W.T) # (N, C)
    true_index = np.vstack([np.arange(X.shape[0]), y]).T # index map (N,2)
    row_indices, col_indices = true_index[:, 0], true_index[:, 1]
    true_scores = softmax_scores[row_indices, col_indices] # y_i's for imgs (N,1)
    all_score = np.log(np.sum(np.exp(softmax_scores), axis=1))-true_scores
    loss = np.sum(all_score, axis=0)/batch_size

    for i in tqdm(range(batch_size)):
        for j in range(10):
            if j==y[i]:
                grad[j,:] += ((np.exp(self.W[y[i]]@X[i])\
                                    /np.sum(np.exp(self.W@X[i]))-1)*X[i]

                else:
                    grad[j,:] += (np.exp(self.W[j]@X[i])\
                                    /np.sum(np.exp(self.W@X[i])))*X[i]

        grad = grad/batch_size
        # ================================================================ #
        # 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.

        Hint: Use a indicator(mask) matrix
        matrix (N, C)
        each row marks correct class 1 with the remaining entries 0
        """
        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.
        # ================================================================ #
        batch_size = X.shape[0]
        softmax_scores = np.matmul(X, self.W.T) # (N, C)
        true_index = np.vstack([np.arange(X.shape[0]), y]).T # index map (N,2)
        row_indices, col_indices = true_index[:, 0], true_index[:, 1]
        true_scores = softmax_scores[row_indices, col_indices] # y_i's for imgs (N,1)
        all_score = np.log(np.sum(np.exp(softmax_scores), axis=1))-true_scores
        loss = np.sum(all_score, axis=0)/batch_size

        # vectorize using matrix multiplication
        # divide matrix by a the sum vector (used to be a scalar in for loop, now with N exa
        # grad (C, D); XN, D)
        grad_ = np.divide(np.exp(np.matmul(self.W, X.T)), \
                          np.sum(np.exp(np.matmul(self.W, X.T)), axis=0))
        grad_ = np.matmul(grad_, X)
        row_indices, col_indices = np.arange(batch_size), y

        # Use a indicator(mask) matrix (N, C); X (N, D)
        # each row marks correct class as 1 and remaining entries 0
        # use this to vectorize the extra term -X[i] if j==y[i]
        mask = np.zeros([batch_size, 10])
        mask[row_indices, col_indices] = 1 # make a mask matrix frrom indice arrays
        grad = (grad_ - np.matmul(mask.T, X))/batch_size

        # ================================================================ #
        # 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 cl

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

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

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

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

    #randomly select 200(#batch_size) training datas for SGD
    batch_choice = np.random.choice(np.arange(X.shape[0]), batch_size)
    X_batch = X[batch_choice]
    y_batch = y[batch_choice]

    # ================================================================ #
    # 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 # W <- W-lr*(dL/dW)

            # ================================================================ #
            # 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[0])
        # ================================================================ #
        # YOUR CODE HERE:
        #   Predict the labels given the training data.
        # ================================================================ #

        scores = np.matmul(self.W, X.T)
        y_pred = np.argmax(scores, axis=0).T

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

        return y_pred
```