This is the softmax workbook for ECE 239AS 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.

In [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
```

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

    # subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

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

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

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

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


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev = get_CIFAR1
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,)
```

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.

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

```
In [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

        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

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

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

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

        2.3277607028048757
```

Question:

You'll notice the loss returned by the softmax is about 2.3 (if implemented correctly). Why does this value make sense?

Answer:

Since we have already normlized the data and the inital softmax classifier favors each class equally, the loss returned should be ~log(num_classes).

Softmax gradient

In [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 the
#   use the appropriate intermediate values to calculate the gradient.

loss, grad = softmax.loss_and_grad(X_dev,y_dev)
# loss, grad = softmax.loss_and_grad(X_train,y_train)
# print(loss)

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

```
numerical: 0.398346 analytic: 0.398346, relative error: 3.517939e-08
numerical: -0.057680 analytic: -0.057680, relative error: 5.681091e-07
numerical: -1.167630 analytic: -1.167630, relative error: 1.616099e-08
numerical: 0.706282 analytic: 0.706281, relative error: 7.127798e-08
numerical: 0.421403 analytic: 0.421402, relative error: 8.744608e-08
numerical: 2.197265 analytic: 2.197265, relative error: 1.091924e-08
numerical: 0.279155 analytic: 0.279155, relative error: 2.508954e-07
numerical: -0.616502 analytic: -0.616502, relative error: 6.510035e-08
numerical: 1.314428 analytic: 1.314428, relative error: 7.188993e-09
numerical: -3.702145 analytic: -3.702145, relative error: 1.568336e-08
```

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.

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

In [9]:
```
## Implement softmax.fast_loss_and_grad which calculates the loss and gra
#    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.

tic = time.time()
loss_vectorized, grad_vectorized = softmax.fast_loss_and_grad(X_dev, y_de
toc = time.time()
print('Vectorized loss / grad: {} / {} computed in {}s'.format(loss_vecto

# The losses should match but your vectorized implementation should be mu
print('difference in loss / grad: {} /{} '.format(loss - loss_vectorized,

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

```
Normal loss / grad_norm: 2.3339288796696986 / 320.57262196944913 compu
ted in 0.16233110427856445s
Vectorized loss / grad: 2.3339288796697 / 320.57262196944913 computed
in 0.0158689022064209s
difference in loss / grad: -1.3322676295501878e-15 /2.8103635486604383
e-13
```

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).

Question:

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

Answer:

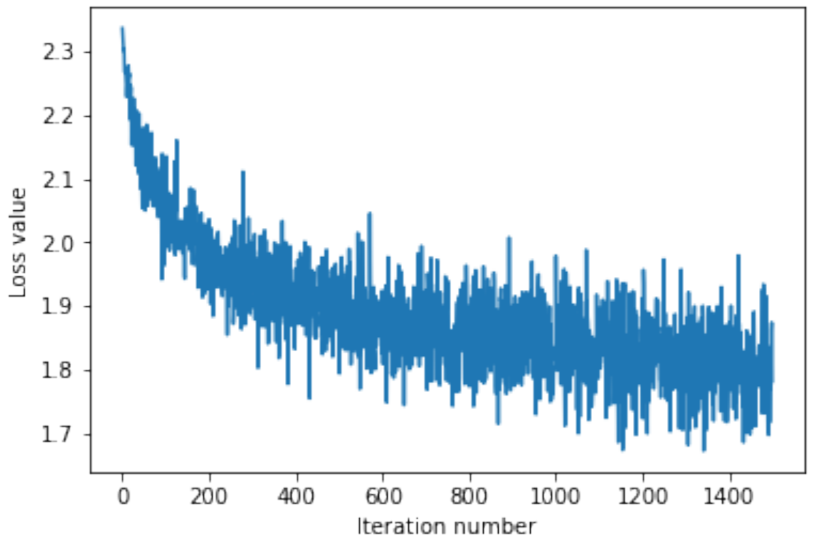
Actually there is no difference between softmax gradient descent training step and the svm training step. Both of them are updating the weights by subtracting the gradients multiplied by the learning rate. However, there exists a difference between their gradient function. For svm, when the label is correct (j = y[i]), the gradient descent of it is just X[i], and when the label is incorrect, the gradient descent is -X[i]. However for softmax, it is like to calculate a condition probability (p) on each X[i], then if the label is correct, the gradient descent is (p-1)X[i], and if it is incorrect, the gradient descent will be p*X[i].

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


tic = time.time()
loss_hist = softmax.train(X_train, y_train, learning_rate=1e-7, num_iters
# loss_hist = softmax.train(X_train, y_train, learning_rate=5e-7, num_ite
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 1.9583142443981614
iteration 500 / 1500: loss 1.8622653073541355
iteration 600 / 1500: loss 1.8532611454359382
iteration 700 / 1500: loss 1.835306222372583
iteration 800 / 1500: loss 1.829389246882764
iteration 900 / 1500: loss 1.8992158530357484
iteration 1000 / 1500: loss 1.9783503540252299
iteration 1100 / 1500: loss 1.8470797913532633
iteration 1200 / 1500: loss 1.8411450268664082
iteration 1300 / 1500: loss 1.7910402495792102
iteration 1400 / 1500: loss 1.8705803029382257
That took 12.67657995223999s
```



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

```
In [11]: ## Implement softmax.predict() and use it to compute the training and tes

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

Optimize the softmax classifier

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

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

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

```
In [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
         #
         #   Select the SVM that achieved the best validation error and report
         #     its error rate on the test set.
         # ================================================================ #
         learning_rates = np.array([7e-7, 1e-6, 4e-6, 7e-6, 1e-5, 4e-5, 7e-5])

         best_softmax = None
         best_val_acc = -1
         best_lr = 0

         for lr in learning_rates:
             softmax = Softmax(dims=[num_classes, num_features])
             losses = softmax.train(X_train, y_train, learning_rate = lr,
                                   num_iters=1500, verbose=True)
             y_train_pred = softmax.predict(X_train)
             print('training accuracy: {}'.format(np.mean(np.equal(y_train,y_train
             y_val_pred = softmax.predict(X_val)
             val_acc = np.mean(np.equal(y_val, y_val_pred))
```

```
    print('validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val_

    if val_acc > best_val_acc:
        best_softmax = softmax
        best_val_acc = val_acc
        best_lr = lr
print('The best learning rate is {}'.format(lr, ))
print('The best validation accuracy is {} and its corresponding validatio

y_test_pred = best_softmax.predict(X_test)
test_acc = np.mean(np.equal(y_test, y_test_pred))
print('testing accuracy: {}'.format(test_acc, ))



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

```
iteration 0 / 1500: loss 2.3647455566656226
iteration 100 / 1500: loss 1.7895958179038585
iteration 200 / 1500: loss 1.7683277130834936
iteration 300 / 1500: loss 1.838658622343926
iteration 400 / 1500: loss 1.817211364117334
iteration 500 / 1500: loss 1.7015302844759808
iteration 600 / 1500: loss 1.752257123113929
iteration 700 / 1500: loss 1.8358793374061855
iteration 800 / 1500: loss 1.8166590744536828
iteration 900 / 1500: loss 1.8122447397395467
iteration 1000 / 1500: loss 1.5755750355461282
iteration 1100 / 1500: loss 1.6795700274481435
iteration 1200 / 1500: loss 1.772104885974278
iteration 1300 / 1500: loss 1.7883418458233162
iteration 1400 / 1500: loss 1.7304260831823877
training accuracy: 0.41746938775510206
validation accuracy: 0.408
iteration 0 / 1500: loss 2.4021441573551026
iteration 100 / 1500: loss 1.8350557682218942
iteration 200 / 1500: loss 1.8500911414944952
iteration 300 / 1500: loss 1.822562814641484
iteration 400 / 1500: loss 1.7362610000911636
iteration 500 / 1500: loss 1.632724842221387
iteration 600 / 1500: loss 1.741192868375939
iteration 700 / 1500: loss 1.7105161206348083
iteration 800 / 1500: loss 1.7763574005831406
iteration 900 / 1500: loss 1.6656420604264717
iteration 1000 / 1500: loss 1.8009178789683227
iteration 1100 / 1500: loss 1.6875567651068213
iteration 1200 / 1500: loss 1.7327007403888408
iteration 1300 / 1500: loss 1.8576128822480065
iteration 1400 / 1500: loss 1.8452184563054466
training accuracy: 0.4177959183673469
validation accuracy: 0.417
iteration 0 / 1500: loss 2.331228877278063
iteration 100 / 1500: loss 1.788273851695311
iteration 200 / 1500: loss 1.7464719551706238
iteration 300 / 1500: loss 1.805693485140309
iteration 400 / 1500: loss 1.7601206529852218
iteration 500 / 1500: loss 1.909601946275884
iteration 600 / 1500: loss 1.7402858805900119
iteration 700 / 1500: loss 1.782950951380771
iteration 800 / 1500: loss 1.7036318565188777
iteration 900 / 1500: loss 1.5391680329835185
iteration 1000 / 1500: loss 1.798695198692907
iteration 1100 / 1500: loss 1.815714091251636
iteration 1200 / 1500: loss 1.6152719560415247
iteration 1300 / 1500: loss 1.8079648779103832
iteration 1400 / 1500: loss 1.6045247431428533
training accuracy: 0.4084081632653061
validation accuracy: 0.379
iteration 0 / 1500: loss 2.32982808647334
iteration 100 / 1500: loss 2.0084662313206683
iteration 200 / 1500: loss 2.079507224909775
iteration 300 / 1500: loss 2.351430080415239
iteration 400 / 1500: loss 2.3319968607119286
iteration 500 / 1500: loss 1.873352297761005
iteration 600 / 1500: loss 1.844645269173414
iteration 700 / 1500: loss 1.9031083296540061
iteration 800 / 1500: loss 1.9655154084523345
iteration 900 / 1500: loss 2.0568005022309266
iteration 1000 / 1500: loss 1.7650636881295305
iteration 1100 / 1500: loss 1.6819870674176918
iteration 1200 / 1500: loss 1.9596063535403434
iteration 1300 / 1500: loss 2.1668979454398127
iteration 1400 / 1500: loss 2.0094836889541057
training accuracy: 0.3425918367346939
validation accuracy: 0.324
iteration 0 / 1500: loss 2.3513886692503307
iteration 100 / 1500: loss 3.1478513657837515
iteration 200 / 1500: loss 3.267136912291495
iteration 300 / 1500: loss 3.072345433838428
iteration 400 / 1500: loss 2.536594587155592
iteration 500 / 1500: loss 2.7572891534713277
iteration 600 / 1500: loss 2.4319050147283456
iteration 700 / 1500: loss 3.833296547068013
iteration 800 / 1500: loss 3.804956421316697
iteration 900 / 1500: loss 2.855061328521807
iteration 1000 / 1500: loss 2.946975481449491
iteration 1100 / 1500: loss 2.4120322189815138
iteration 1200 / 1500: loss 2.9626942298947974
iteration 1300 / 1500: loss 2.9693062309965184
iteration 1400 / 1500: loss 2.4267529908755106
training accuracy: 0.34916326530612246
validation accuracy: 0.33
iteration 0 / 1500: loss 2.4432746460289296
iteration 100 / 1500: loss 9.61018702035632
iteration 200 / 1500: loss 8.005908726793542
iteration 300 / 1500: loss 12.33982943107916
iteration 400 / 1500: loss 12.978447087251993
iteration 500 / 1500: loss 9.803076737241955
iteration 600 / 1500: loss 10.361817623902203
iteration 700 / 1500: loss 10.569011114255977
iteration 800 / 1500: loss 10.643609885548862
iteration 900 / 1500: loss 9.767047745673263
iteration 1000 / 1500: loss 15.337638751806058
iteration 1100 / 1500: loss 11.611471097628073
iteration 1200 / 1500: loss 13.10893384939679
iteration 1300 / 1500: loss 9.628758516567421
iteration 1400 / 1500: loss 7.557085070114904
training accuracy: 0.3154489795918367
validation accuracy: 0.331
iteration 0 / 1500: loss 2.337055619175014
iteration 100 / 1500: loss 17.209318676554066
iteration 200 / 1500: loss 15.61893876899142
iteration 300 / 1500: loss 14.415547976680195
iteration 400 / 1500: loss 14.615521748806886
iteration 500 / 1500: loss 18.59092410264801
iteration 600 / 1500: loss 23.615464000792368
iteration 700 / 1500: loss 20.578560825637222
iteration 800 / 1500: loss 21.13015109438893
iteration 900 / 1500: loss 19.142806157881633
iteration 1000 / 1500: loss 14.995445305670877
iteration 1100 / 1500: loss 21.990219469963154
iteration 1200 / 1500: loss 18.494082304350176
iteration 1300 / 1500: loss 24.607680667951435
iteration 1400 / 1500: loss 16.18395865037472
training accuracy: 0.30948979591836734
validation accuracy: 0.296
The best learning rate is 7e-05
The best validation accuracy is 0.417 and its corresponding validation
error is 0.583
testing accuracy: 0.389
```

```
In [ ]:
```