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=1000
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepar
    it for the linear classifier. These are the same steps as we used for t
    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_CIFAR10_
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 a

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)
```

```
2.3277607028048966
```

Question:

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

Answer:

Random chance is 0.1, and so we should expect the loss to be ln(10/1), which it is (ln(10) is approximately 2.3).

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 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
softmax.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: 0.521283 analytic: 0.521283, relative error: 4.486898e-08
numerical: 1.487680 analytic: 1.487680, relative error: 1.262243e-08
numerical: -0.632911 analytic: -0.632911, relative error: 8.090488e-09
numerical: 0.643084 analytic: 0.643084, relative error: 1.800101e-08
numerical: 1.049247 analytic: 1.049247, relative error: 6.407224e-08
numerical: 1.680929 analytic: 1.680929, relative error: 2.309006e-08
numerical: -0.161634 analytic: -0.161634, relative error: 1.753171e-07
numerical: -2.457132 analytic: -2.457132, relative error: 4.540432e-09
numerical: 1.227282 analytic: 1.227282, relative error: 5.211303e-09
numerical: -2.886716 analytic: -2.886716, relative error: 9.843126e-09
```

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 gradi
#    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.li

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_vectori

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

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

```
Normal loss / grad_norm: 2.321805367940105 / 329.3714006915056 computed i
n 0.15343189239501953s
Vectorized loss / grad: 2.321805367940105 / 329.3714006915056 computed in
0.03251814842224121s
difference in loss / grad: 0.0 /2.1406619004282536e-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:

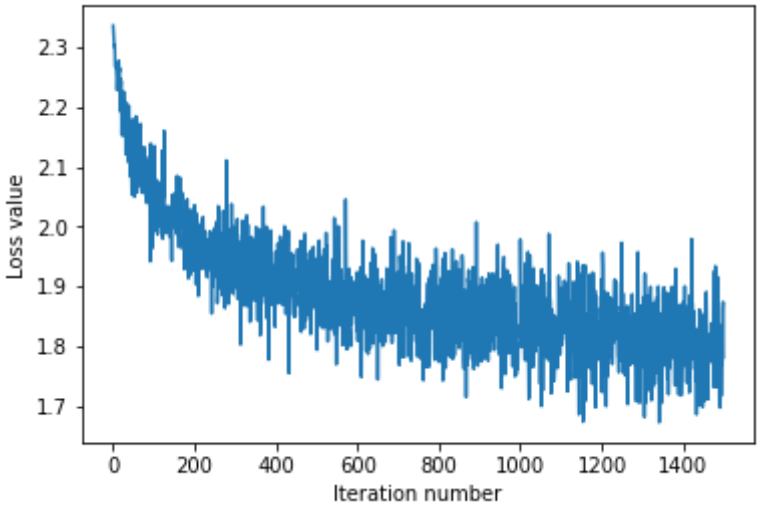
It is the same.

In [10]:
```
# Implement softmax.train() by filling in the code to extract a batch of da
# 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 1.8532611454359387
iteration 700 / 1500: loss 1.8353062223725827
iteration 800 / 1500: loss 1.829389246882764
iteration 900 / 1500: loss 1.899215853035748
iteration 1000 / 1500: loss 1.9783503540252299
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 10.551152229309082s
```



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 testi

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
```

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 e
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
# ================================================================ #
for learning_rate in [1e-7, 1e-6, 1e-5, 1e-4]:
    print('learning rate: {}'.format(learning_rate))
    loss_hist = softmax.train(X_train, y_train, learning_rate=learning_rate
                      num_iters=1500, verbose=True)
    y_val_pred = softmax.predict(X_val)
    print('validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val_pr
    print()

print('learning rate: {}'.format(1e-6))
loss_hist = softmax.train(X_train, y_train, learning_rate=1e-6,
                      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))))
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
learning rate: 1e-07
iteration 0 / 1500: loss 2.3353835450891545
iteration 100 / 1500: loss 2.022509394631719
iteration 200 / 1500: loss 1.982172871654982
iteration 300 / 1500: loss 1.9356442081331486
iteration 400 / 1500: loss 1.882893396815689
iteration 500 / 1500: loss 1.8181869697394497
iteration 600 / 1500: loss 1.874513153185746
iteration 700 / 1500: loss 1.8361832500173585
iteration 800 / 1500: loss 1.8584086819212182
iteration 900 / 1500: loss 1.9275087067564147
iteration 1000 / 1500: loss 1.824667969507725
iteration 1100 / 1500: loss 1.7731817984393607
iteration 1200 / 1500: loss 1.8636308568113116
iteration 1300 / 1500: loss 1.924074621260815
iteration 1400 / 1500: loss 1.7846918635831293
validation accuracy: 0.39

learning rate: 1e-06
iteration 0 / 1500: loss 2.4615346985497166
iteration 100 / 1500: loss 1.7515308294429426
iteration 200 / 1500: loss 1.8653151657870888
iteration 300 / 1500: loss 1.7068279724663447
iteration 400 / 1500: loss 1.6919412980959523
iteration 500 / 1500: loss 1.7445602534086055
iteration 600 / 1500: loss 1.9070927441191992
iteration 700 / 1500: loss 1.6266282009657491
iteration 800 / 1500: loss 1.7137070023201977
```

```
iteration 900 / 1500: loss 1.6755856266246776
iteration 1000 / 1500: loss 1.8076485575084924
iteration 1100 / 1500: loss 1.7256661402410827
iteration 1200 / 1500: loss 1.698554182531272
iteration 1300 / 1500: loss 1.792082948906467
iteration 1400 / 1500: loss 1.6508555418428204
validation accuracy: 0.415

learning rate: 1e-05
iteration 0 / 1500: loss 2.3790388831757823
iteration 100 / 1500: loss 2.3921785131993816
iteration 200 / 1500: loss 3.27163372235802
iteration 300 / 1500: loss 2.5266749616255084
iteration 400 / 1500: loss 2.5171351792770027
iteration 500 / 1500: loss 2.9346880547320247
iteration 600 / 1500: loss 1.9232171018336048
iteration 700 / 1500: loss 3.44210890971557
iteration 800 / 1500: loss 2.27376749441641
iteration 900 / 1500: loss 2.745258811167521
iteration 1000 / 1500: loss 2.589548028949166
iteration 1100 / 1500: loss 2.737905712614821
iteration 1200 / 1500: loss 2.747594734219702
iteration 1300 / 1500: loss 2.5731286656092824
iteration 1400 / 1500: loss 3.1635404402767886
validation accuracy: 0.27

learning rate: 0.0001
iteration 0 / 1500: loss 2.338799247622096
iteration 100 / 1500: loss 34.98601680024577
iteration 200 / 1500: loss 56.97405897104403
iteration 300 / 1500: loss 14.824554602730215
iteration 400 / 1500: loss 25.6996155749384
iteration 500 / 1500: loss 33.93294068543631
iteration 600 / 1500: loss 24.208229112153166
iteration 700 / 1500: loss 20.911355216047518
iteration 800 / 1500: loss 39.34415640628004
iteration 900 / 1500: loss 16.4359260239477
iteration 1000 / 1500: loss 24.71359284063535
iteration 1100 / 1500: loss 18.564259007064685
iteration 1200 / 1500: loss 28.5426977737122
iteration 1300 / 1500: loss 36.66212965037604
iteration 1400 / 1500: loss 36.16062817387996
validation accuracy: 0.271

learning rate: 1e-06
iteration 0 / 1500: loss 2.372474398135772
iteration 100 / 1500: loss 1.892919730065111
iteration 200 / 1500: loss 1.790029417712668
iteration 300 / 1500: loss 1.8396019743788241
iteration 400 / 1500: loss 1.6965287331558643
iteration 500 / 1500: loss 1.8498152607388902
iteration 600 / 1500: loss 1.608555393293517
iteration 700 / 1500: loss 1.7768936439173046
iteration 800 / 1500: loss 1.7288467550175795
iteration 900 / 1500: loss 1.7191678304814781
iteration 1000 / 1500: loss 1.8442746025120789
iteration 1100 / 1500: loss 1.7596152530507356
```

```
iteration 1200 / 1500: loss 1.7196895935660637
iteration 1300 / 1500: loss 1.720036481810709
iteration 1400 / 1500: loss 1.7480014172362022
test accuracy: 0.392
```