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
, num_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepar
e
    it for the linear classifier. These are the same steps as we used for t
he
    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_
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,))
```

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
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

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:

The weights are initialized to a normal distribution with mean 0, which means that the expected value of the samples would most likely also be 0. This would result in the log term in the loss function having an argument of 10 each time, yielding 2.3. Since this should be the same for every sample, the average loss would also come out to 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
you implemented the gradient correctly.
softmax.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: -0.461254 analytic: -0.461254, relative error: 1.147387e-09
numerical: 1.415071 analytic: 1.415071, relative error: 3.611131e-08
numerical: 0.221536 analytic: 0.221536, relative error: 1.913397e-08
numerical: 0.823889 analytic: 0.823889, relative error: 3.548639e-08
numerical: 0.131422 analytic: 0.131421, relative error: 5.121652e-07
numerical: 1.079281 analytic: 1.079281, relative error: 4.340823e-08
numerical: -0.442191 analytic: -0.442191, relative error: 1.171090e-07
numerical: -1.603514 analytic: -1.603514, relative error: 1.784821e-08
numerical: -0.774959 analytic: -0.774959, relative error: 1.523634e-08
numerical: -3.820474 analytic: -3.820474, relative error: 1.752965e-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 gradi
ent
#    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
nalg.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_vectori
zed, 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, n
p.linalg.norm(grad - grad_vectorized)))

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

```
Normal loss / grad_norm: 2.30711514969 / 324.916463415 computed in 0.064689
874649s
Vectorized loss / grad: 2.30711514969 / 324.916463415 computed in 0.0102250
576019s
difference in loss / grad: 4.4408920985e-16 /2.149853952e-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:

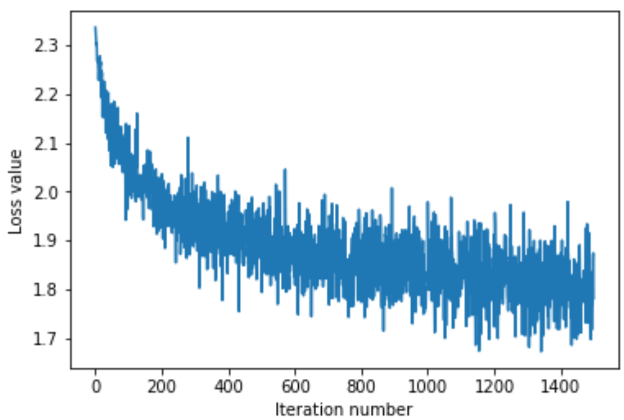
They should be the same, aside from the loss and gradient calculations

In [10]:
```
# Implement softmax.train() by filling in the code to extract a batch of da
ta
# 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 2.33659266066
iteration 100 / 1500: loss 2.05572226139
iteration 200 / 1500: loss 2.03577451207
iteration 300 / 1500: loss 1.98133481656
iteration 400 / 1500: loss 1.9583142444
iteration 500 / 1500: loss 1.86226530735
iteration 600 / 1500: loss 1.85326114544
iteration 700 / 1500: loss 1.83530622237
iteration 800 / 1500: loss 1.82938924688
iteration 900 / 1500: loss 1.89921585304
iteration 1000 / 1500: loss 1.97835035403
iteration 1100 / 1500: loss 1.84707979135
iteration 1200 / 1500: loss 1.84114502687
iteration 1300 / 1500: loss 1.79104024958
iteration 1400 / 1500: loss 1.87058030294
That took 7.12574601173s
```



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
ng 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: 0.381142857143
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 [14]:
```
# ================================================================ #
# 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
rror.
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
# ================================================================ #


learning_rates = [5e-5, 1e-5, 5e-6, 1e-6, 5e-7, 1e-7, 5e-8, 1e-8]
validation_scores = []


for i in np.arange(len(learning_rates)):
    softmax.train(X_train, y_train, learning_rate=learning_rates[i],
                  num_iters=1500, verbose=False)

    y_val_pred = softmax.predict(X_val)
    validation_scores.append(np.mean(np.equal(y_val, y_val_pred)))


m_idx = np.argmax(validation_scores)

print validation_scores
print 'Learning rate of {} achieved the best validation rate of {}'.format(
learning_rates[m_idx], validation_scores[m_idx])

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

```
[0.312, 0.346, 0.389, 0.411, 0.404, 0.384, 0.365, 0.313]
Learning rate of 1e-06 achieved the best validation rate of 0.411
```