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.

In [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
```

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 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 = '/Users/wangyuchen/desktop/COM SCI 247/HW/HW2/hw2_Questions/code/cifar-10-batches-py' # You ne
    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 [216]:

```
from nndl import Softmax
```

In [217]:

```
# 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 [218]:

```
## Implement the loss function of the softmax using a for loop over
#  the number of examples

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

In [219]:

```
print(loss)
```

```
2.327760702804897
```

Question:

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

Answer:

Currently, there is no regularization or trained model, and we are using the random weights. This might cause the large error.

Softmax gradient

In [220]:
```
## 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 correct.
softmax.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: -0.969757 analytic: -0.969757, relative error: 2.962114e-08
numerical: -0.541608 analytic: -0.541608, relative error: 2.532321e-08
numerical: 0.059517 analytic: 0.059517, relative error: 2.484896e-07
numerical: 2.379901 analytic: 2.379901, relative error: 1.451935e-09
numerical: -0.471978 analytic: -0.471978, relative error: 4.772627e-08
numerical: 1.905349 analytic: 1.905349, relative error: 2.032549e-09
numerical: 0.668743 analytic: 0.668743, relative error: 1.007004e-07
numerical: -0.524710 analytic: -0.524710, relative error: 1.791734e-08
numerical: 1.508797 analytic: 1.508797, relative error: 9.542764e-10
numerical: -4.177994 analytic: -4.177995, relative error: 9.838742e-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 [221]:
```
import time
```

In [222]:

```
## 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,

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

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

```
Normal loss / grad_norm: 2.3431081152133433 / 337.4978602358297 computed in 0.09418487548828125s
Vectorized loss / grad: 2.3431081152133424 / 337.4978602358297 computed in 0.016119003295898438s
difference in loss / grad: 8.881784197001252e-16 /2.3101392126522646e-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).

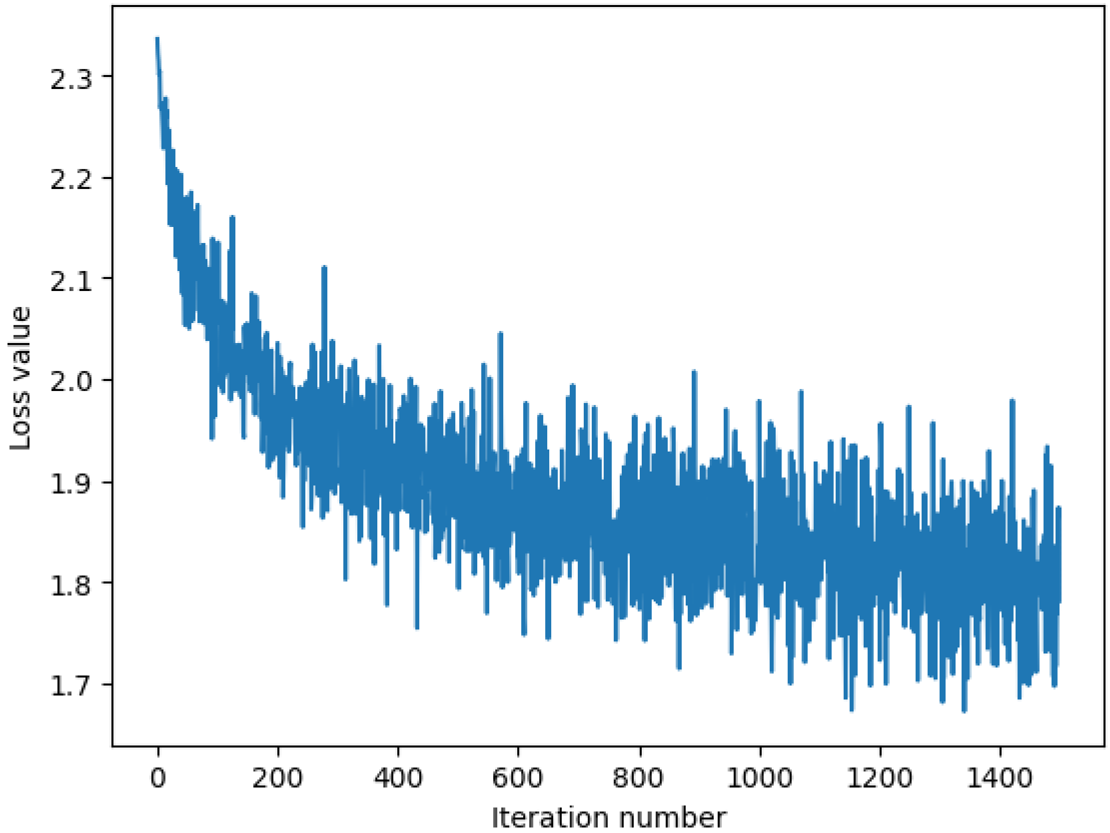
In [223]:

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


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

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

```
iteration 0 / 1500: loss 2.336592660663754
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 1.8293892468827635
iteration 900 / 1500: loss 1.8992158530357484
iteration 1000 / 1500: loss 1.9783503540252299
iteration 1100 / 1500: loss 1.8470797913532635
iteration 1200 / 1500: loss 1.8411450268664082
iteration 1300 / 1500: loss 1.7910402495792102
iteration 1400 / 1500: loss 1.8705803029382257
That took 5.400985240936279s
```



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

In [211]:
```
## 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
```

Optimize the softmax classifier

In [212]:
```
np.finfo(float).eps
```

Out[212]:
```
2.220446049250313e-16
```

In [213]:

```
# ================================================================ #
# YOUR CODE HERE:
#   Train the Softmax classifier with different learning rates and
#     evaluate on the validation data.
#   Report:
#     - The best learning rate of the ones you tested.
#     - The best validation accuracy corresponding to the best validation error.
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
# ================================================================ #

rates = [1e-7, 1e-6, 1e-5, 1e-4]
for learningrate in rates: #for each learning rate, run the train function
    print("learning rate:", learningrate)
    loss_histnew = softmax.train(X_train, y_train, learning_rate=learningrate,
                          num_iters=1500, verbose=True)
    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)), ))

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

```
learning rate: 1e-07
iteration 0 / 1500: loss 2.335383545089155
iteration 100 / 1500: loss 2.0225093946317187
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 1.9240746212608146
iteration 1400 / 1500: loss 1.7846918635831293
training accuracy: 0.37881632653061226
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 1.7068279724663449
iteration 400 / 1500: loss 1.6919412980959523
iteration 500 / 1500: loss 1.7445602534086055
iteration 600 / 1500: loss 1.9070927441191992
iteration 700 / 1500: loss 1.6266282009657487
iteration 800 / 1500: loss 1.7137070023201975
iteration 900 / 1500: loss 1.6755856266246776
iteration 1000 / 1500: loss 1.8076485575084922
iteration 1100 / 1500: loss 1.7256661402410827
iteration 1200 / 1500: loss 1.698554182531272
iteration 1300 / 1500: loss 1.792082948906467
iteration 1400 / 1500: loss 1.6508555418428208
training accuracy: 0.42051020408163264
validation accuracy: 0.415
learning rate: 1e-05
iteration 0 / 1500: loss 2.3790388831757823
iteration 100 / 1500: loss 2.3921785131993487
iteration 200 / 1500: loss 3.271633722357008
iteration 300 / 1500: loss 2.5266749618188404
iteration 400 / 1500: loss 2.51713517924201
iteration 500 / 1500: loss 2.934688054749696
iteration 600 / 1500: loss 1.9232171018196902
iteration 700 / 1500: loss 3.4421089096587862
iteration 800 / 1500: loss 2.2737674945475397
iteration 900 / 1500: loss 2.7452588111633127
iteration 1000 / 1500: loss 2.5895477769472315
iteration 1100 / 1500: loss 2.737909522402885
iteration 1200 / 1500: loss 2.7475945671971704
iteration 1300 / 1500: loss 2.573128824801604
iteration 1400 / 1500: loss 3.1635751023691183
training accuracy: 0.28981632653061223
validation accuracy: 0.27
learning rate: 0.0001
iteration 0 / 1500: loss 2.338799247622096
```

```
iteration 100 / 1500: loss 26.621720114307177
iteration 200 / 1500: loss 36.82287856760831
iteration 300 / 1500: loss 23.05758218762376
iteration 400 / 1500: loss 24.23927687078789
iteration 500 / 1500: loss 40.05316246882273
iteration 600 / 1500: loss 39.80552593669265
iteration 700 / 1500: loss 30.55916210578722
iteration 800 / 1500: loss 15.637270148789185
iteration 900 / 1500: loss 18.13300630986938
iteration 1000 / 1500: loss 17.848554461868563
iteration 1100 / 1500: loss 23.340047437387128
iteration 1200 / 1500: loss 27.352905469183952
iteration 1300 / 1500: loss 38.00938801736349
iteration 1400 / 1500: loss 36.24586649495245
training accuracy: 0.26402040816326533
validation accuracy: 0.234
```

Report:

The best learning rate is 1e-06, where the training accuracy: 0.42051020408163264 and validation accuracy: 0.415