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 [2]:

```
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 [3]:

```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, num_dev=
    """
    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 = '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,)
```

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 [4]:
```
from nndl import Softmax
```

In [5]:
```
# 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 see

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 [6]:
```
## Implement the loss function of the softmax using a for loop over
#  the number of examples

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

In [7]:
```
print(loss)
```

```
[2.3277607]
```

Question:

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

Answer:

This makes sense as we have not trained the classifier yet, we are simply calculating a normalized loss on a random set of classification vectors. As a result, the vectors should not accurately represent the data and thus lead to a large normalized loss. It makes sense that the value is not extremely large as the initial guess has a small norm. Additonally, the loss has been normalzied.

Softmax gradient

In [8]:
```
## 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 implem
softmax.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: -0.094231 analytic: -0.094231, relative error: 3.358148e-07
numerical: -0.273894 analytic: -0.273894, relative error: 5.743756e-08
numerical: 0.571406 analytic: 0.571406, relative error: 2.288228e-09
numerical: 2.481332 analytic: 2.481332, relative error: 6.601671e-09
numerical: 0.521445 analytic: 0.521445, relative error: 5.902726e-08
numerical: 2.882311 analytic: 2.882311, relative error: 1.770825e-08
numerical: -0.126464 analytic: -0.126464, relative error: 4.546086e-07
numerical: -0.551219 analytic: -0.551219, relative error: 3.137158e-10
numerical: 0.598242 analytic: 0.598242, relative error: 1.344027e-09
numerical: -2.998195 analytic: -2.998195, relative error: 1.825305e-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 [9]:
```
import time
```

In [10]:
```
## 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(

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.li

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

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

```
Normal loss / grad_norm: [2.31663569] / 352.6632020772778 computed in 0.2693278789520
2637s
Vectorized loss / grad: [2.31663569] / 352.6632020772778 computed in 0.00650405883789
0625s
difference in loss / grad: [-2.66453526e-15] /2.3709429200456677e-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:

Question to be ignored.

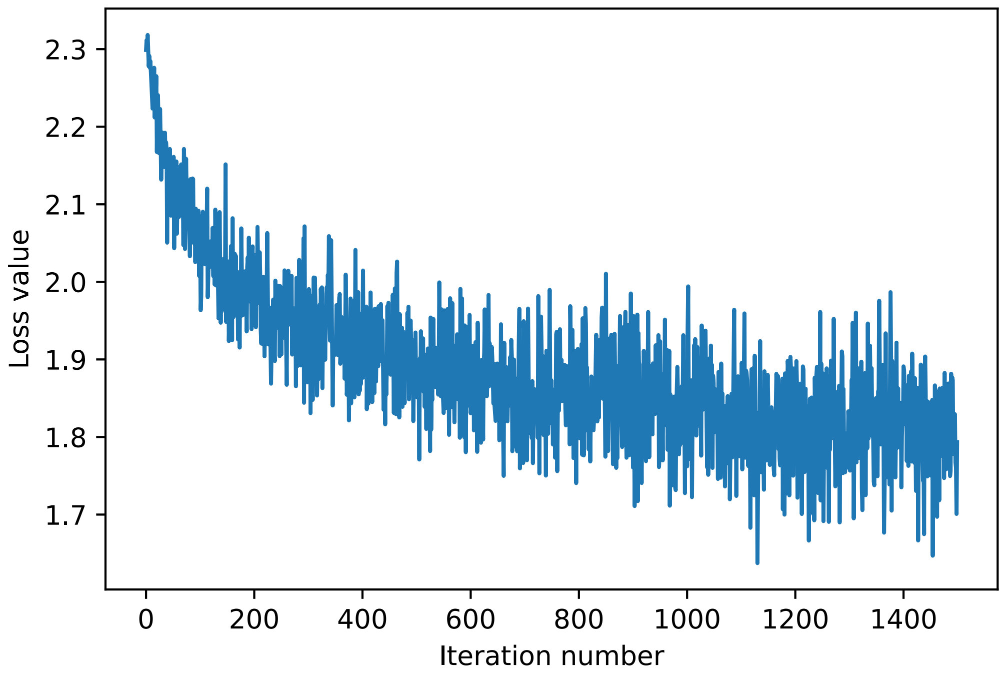
In [11]:

```
# 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 [2.29890543]
iteration 100 / 1500: loss [2.04284746]
iteration 200 / 1500: loss [2.01938856]
iteration 300 / 1500: loss [1.88237294]
iteration 400 / 1500: loss [1.90184648]
iteration 500 / 1500: loss [1.88075198]
iteration 600 / 1500: loss [1.83059957]
iteration 700 / 1500: loss [1.92422133]
iteration 800 / 1500: loss [1.85980518]
iteration 900 / 1500: loss [1.76067328]
iteration 1000 / 1500: loss [1.76210714]
iteration 1100 / 1500: loss [1.87030925]
iteration 1200 / 1500: loss [1.78090716]
iteration 1300 / 1500: loss [1.82938253]
iteration 1400 / 1500: loss [1.89104848]
That took 9.806801795959473s
```



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

In [12]:

```
## 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: 0.3804489795918367
validation accuracy: 0.402
```

Optimize the softmax classifier

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

In [13]:

```
np.finfo(float).eps
```

Out[13]:
```
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 error.
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
# ================================================================ #
learning_rates = np.power(10,np.linspace(-9,-5, 10))
accuracy = np.zeros_like(learning_rates)
for i,rate in enumerate(learning_rates):
    loss_hist = softmax.train(X_train, y_train, learning_rate=rate,
                      num_iters=1500, verbose=False)
    y_val_pred = softmax.predict(X_val)

    accuracy[i] = np.mean(np.equal(y_val, y_val_pred))
print("Max  Accuracy: ", np.max(accuracy))
print("Max  Accuracy Learning Rate: ", learning_rates[np.argmax(accuracy)])
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
Max  Accuracy:  0.418
Max  Accuracy Learning Rate:  1.2915496650148827e-06
```