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 [5]:

```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000,
    """
    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 th
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'dataset\cifar-10-batches-py' # You need to update this li
    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_d
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 [6]:
```python
from nndl import Softmax
```

In [7]:
```python
# 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 r

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]:
```python
## 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]:
```python
print(loss)
```

```
2.3277607028048966
```

Question:

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

Answer:

Because there are a total of 10 classes, an untrained classifier will guess 0.1 probability for each class. Therefore, for each sample we would expect an error of -log(0.1) = 2.3. Therefore, the mean error is also expected to be 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 y
softmax.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: -1.266499 analytic: -1.266499, relative error: 1.289325e-08
numerical: 0.056842 analytic: 0.056842, relative error: 4.306434e-07
numerical: 0.449315 analytic: 0.449315, relative error: 2.518900e-08
numerical: 0.935716 analytic: 0.935716, relative error: 1.540367e-08
numerical: -0.512110 analytic: -0.512110, relative error: 9.848258e-08
numerical: 0.342088 analytic: 0.342088, relative error: 2.513967e-08
numerical: -0.967954 analytic: -0.967954, relative error: 6.087353e-08
numerical: -1.891557 analytic: -1.891557, relative error: 1.427829e-08
numerical: -0.247811 analytic: -0.247811, relative error: 6.318535e-08
numerical: -2.658215 analytic: -2.658215, relative error: 2.500902e-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 gradie
#    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.lin

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_vectoriz

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

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

```
Normal loss / grad_norm: 2.3230138756030048 / 281.3263564766588 computed in 0.0
7181096076965332s
Vectorized loss / grad: 2.323013874528945 / 281.3263564766588 computed in 0.009
247779846191406s
difference in loss / grad: 1.0740599520886462e-09 /1.9427077332200875e-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:

Due to the change in loss function, the value of the gradients that are used to update the coefficient will change

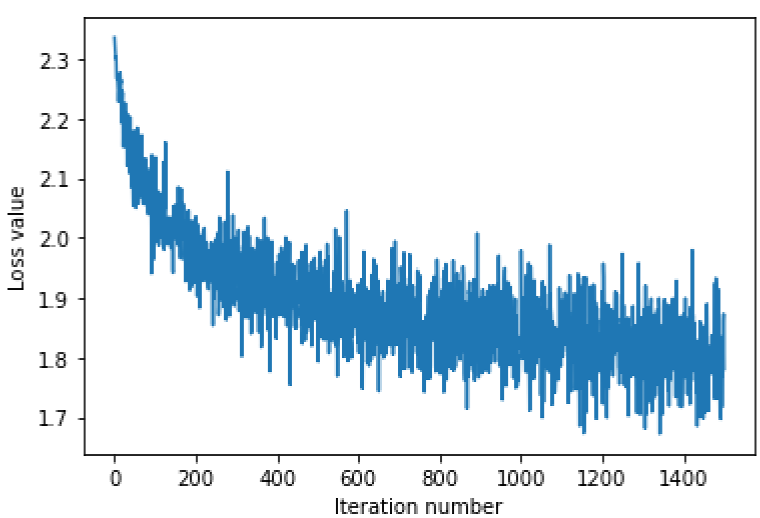
In [46]:

```
# Implement softmax.train() by filling in the code to extract a batch of dat
# 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 1.8532611454359382
iteration 700 / 1500: loss 1.8353062223725827
iteration 800 / 1500: loss 1.829389246882764
iteration 900 / 1500: loss 1.8992158530357484
iteration 1000 / 1500: loss 1.97835035402523
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 6.137814283370972s
```



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

In [47]:
```
## Implement softmax.predict() and use it to compute the training and testin

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 [10]:
```
np.finfo(float).eps
```

Out[10]:
```
2.220446049250313e-16
```

In [11]:

```
# ================================================================ #
# 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 er
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
# ================================================================ #
softmax = Softmax(dims=[num_classes, num_features])
rates = [10**i for i in range(-10,0)]

valAccuracies = []
valLosses = []
for rate in rates:
    softmax.train(X_train, y_train, learning_rate=rate, num_iters=1500, verb
    valLoss,a = softmax.fast_loss_and_grad(X_val, y_val)
    valPreds = softmax.predict(X_val)
    valAcc = np.mean(np.equal(y_val, valPreds))
    valAccuracies.append(valAcc)
    valLosses.append(valLoss)
    print('Current rate:',rate,'validation accuracy: {}'.format(valAcc),'val
    print("Best validation loss so far {}".format(min(valLosses)))
print("-"*100)
bestIndex = np.argmin(valLosses)
bestLR = rates[bestIndex]
bestValLoss = valLosses[bestIndex]
bestValAcc = valAccuracies[bestIndex]
print("The best learning rate is %f.Best validation loss is %f. Best validat


softmax.train(X_train, y_train, learning_rate=bestLR, num_iters=1500, verbos
yTestPred = softmax.predict(X_test)
testAcc = np.mean(np.equal(y_test, yTestPred))
print('Error of the softmax classifier with best learning rate %f on the tes
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

```

```
Current rate: 1e-10 validation accuracy: 0.13 validation Loss: 2.33296360971419
3
Best validation loss so far 2.332963609714193
Current rate: 1e-09 validation accuracy: 0.179 validation Loss: 2.2418688560194
973
Best validation loss so far 2.2418688560194973
Current rate: 1e-08 validation accuracy: 0.302 validation Loss: 2.0201445650194
59
Best validation loss so far 2.020144565019459
Current rate: 1e-07 validation accuracy: 0.379 validation Loss: 1.8287151179489
731
Best validation loss so far 1.8287151179489731
Current rate: 1e-06 validation accuracy: 0.413 validation Loss: 1.7466767713611
```

```
042
Best validation loss so far 1.7466767713611042
Current rate: 1e-05 validation accuracy: 0.316 validation Loss: 2.5812750026016
156
Best validation loss so far 1.7466767713611042
Current rate: 0.0001 validation accuracy: 0.256 validation Loss: 13.73004382669
0776
Best validation loss so far 1.7466767713611042
Current rate: 0.001 validation accuracy: 0.254 validation Loss: 16.890411756297
777
Best validation loss so far 1.7466767713611042
Current rate: 0.01 validation accuracy: 0.145 validation Loss: 16.7166763205028
45
Best validation loss so far 1.7466767713611042
Current rate: 0.1 validation accuracy: 0.181 validation Loss: 17.1312330918501
Best validation loss so far 1.7466767713611042
--------------------------------------------------------------------------------
---------------------
The best learning rate is 0.000001.Best validation loss is 1.746677. Best valid
ation accuracy is 0.413000
Error of the softmax classifier with best learning rate 0.000001 on the test Se
t Error Rate is 0.622000
```