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 = './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 [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 make sense?

Answer:

It is because we have 10 classes and it is randomly distribution. We can expect the probability of each class is 0.1. Therefore, -log(0.1) ~= 2.3 is reasonable to be the loss.

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

```
numerical: -0.597248 analytic: -0.597248, relative error: 1.352238e-09
numerical: 1.231072 analytic: 1.231072, relative error: 5.063644e-09
numerical: -1.063401 analytic: -1.063401, relative error: 3.143855e-09
numerical: 1.924655 analytic: 1.924655, relative error: 1.512317e-08
numerical: 1.005965 analytic: 1.005965, relative error: 5.340822e-08
numerical: 2.265621 analytic: 2.265621, relative error: 1.803286e-08
numerical: -1.235339 analytic: -1.235339, relative error: 3.233409e-08
numerical: -1.706993 analytic: -1.706993, relative error: 8.622539e-09
numerical: -1.196225 analytic: -1.196225, relative error: 8.455186e-09
numerical: -2.421361 analytic: -2.421361, relative error: 2.693823e-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 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,

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.

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

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

```
Normal loss / grad_norm: 2.325139432933735 / 364.46996613435 computed in 0.056212902069091
8s
```

```
Vectorized loss / grad: 2.3251394329337334 / 364.46996613435 computed in 0.0017008781433105469s
difference in loss / grad: 1.7763568394002505e-15 /3.048411952454789e-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 are identical. The processes have no difference.

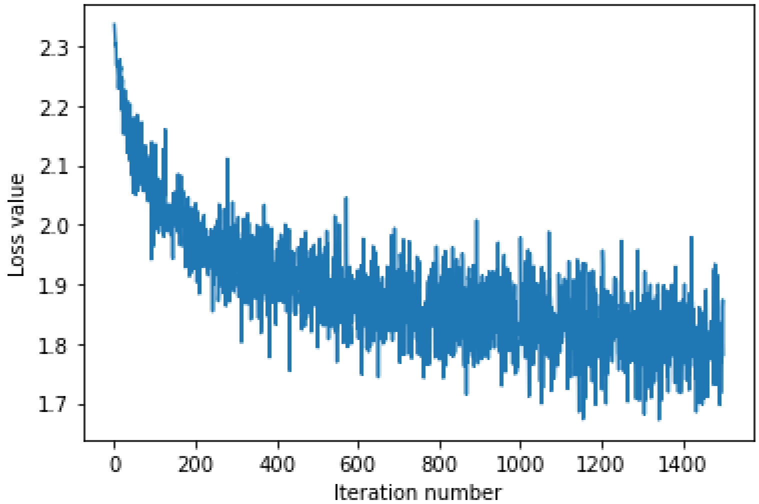
In [10]:

```
# 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 2.3365926606637544
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 1.862265307354135
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 1.97835035402523
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 3.599605083465576s
```



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 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

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 error.
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
# ================================================================ #

learning_rates = [10**i for i in range(-9, 0)]
accuracy = {}

best_learning_rate = 0
best_validation = 0

for learning_rate in learning_rates:
    softmax = Softmax(dims=[num_classes, num_features])
    loss = softmax.train(X_train, y_train, learning_rate=learning_rate, num_iters=1500, ve
```

```
    y_train_pred = softmax.predict(X_train)
    train_accuracy = np.mean(np.equal(y_train,y_train_pred))

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

    accuracy[learning_rate] = (train_accuracy, val_accuracy)

    if best_validation < val_accuracy:
        best_learning_rate = learning_rate
        best_validation = val_accuracy

for learning_rate in accuracy:
    print("Learning Rate: {}, Train Accuracy: {}, Validation: {}".format(learning_rate, ac

print("\nThe Best Learning Rate: {}".format(best_learning_rate))
print("The Best Validation Accuracy: {}".format(best_validation))
print("The Best Validation Error: {}\n".format(1 - best_validation))

# Best Test
softmax.train(X_train, y_train, learning_rate=best_learning_rate, num_iters=1500, verbose=
y_test_pred = softmax.predict(X_test)

test_accuracy = np.mean(np.equal(y_test, y_test_pred))
print("Test Accuracy: {}\nError rate on the test set: {}".format(test_accuracy, 1-test_acc

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

```
/Users/jacky/My_Data/Data/UCLA/2022_Winter/ECE C247_Deep Learning/Homework/HW2/hw2-code/nn
dl/softmax.py:142: RuntimeWarning: divide by zero encountered in log
  probs_log = -np.log(probs_row)
Learning Rate: 1e-09, Train Accuracy: 0.17079591836734695, Validation: 0.16
Learning Rate: 1e-08, Train Accuracy: 0.2886938775510204, Validation: 0.304
Learning Rate: 1e-07, Train Accuracy: 0.38210204081632654, Validation: 0.395
Learning Rate: 1e-06, Train Accuracy: 0.42248979591836733, Validation: 0.407
Learning Rate: 1e-05, Train Accuracy: 0.34916326530612246, Validation: 0.33
Learning Rate: 0.0001, Train Accuracy: 0.2804081632653061, Validation: 0.264
Learning Rate: 0.001, Train Accuracy: 0.2738979591836735, Validation: 0.255
Learning Rate: 0.01, Train Accuracy: 0.2778775510204082, Validation: 0.27
Learning Rate: 0.1, Train Accuracy: 0.2919387755102041, Validation: 0.284

The Best Learning Rate: 1e-06
The Best Validation Accuracy: 0.407
The Best Validation Error: 0.593

Test Accuracy: 0.402
Error rate on the test set: 0.598
```