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.

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 [3]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1
000, num_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to pre
pare
    it for the linear classifier. These are the same steps as we used fo
r 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_CIFAR
10_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 [15]:
```
# 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 [16]:
```
## Implement the loss function of the softmax using a for loop over
#  the number of examples

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

In [17]:
```
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:

You fill this out.

Softmax gradient

In [33]:
```
## 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 th
en
#   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: 0.562437 analytic: 0.748704, relative error: 1.420647e-01
numerical: 0.305438 analytic: -0.379973, relative error: 1.000000e+00
numerical: -3.502289 analytic: -1.823858, relative error: 3.151304e-01
numerical: -0.365661 analytic: -0.107569, relative error: 5.453823e-01
numerical: 0.390667 analytic: -0.102664, relative error: 1.000000e+00
numerical: -2.804612 analytic: -1.763484, relative error: 2.279130e-01
numerical: -1.706104 analytic: 2.509280, relative error: 1.000000e+00
numerical: 0.480570 analytic: -0.732089, relative error: 1.000000e+00
numerical: -1.176266 analytic: -0.280261, relative error: 6.151659e-01
numerical: -1.647541 analytic: 1.179013, relative error: 1.000000e+00
```

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 [31]:
```
import time
```

In [40]:
```
## Implement softmax.fast_loss_and_grad which calculates the loss and gr
adient
#    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_d
ev)
toc = time.time()
print('Vectorized loss / grad: {} / {} computed in {}s'.format(loss_vect
orized, np.linalg.norm(grad_vectorized, 'fro'), toc - tic))

# The losses should match but your vectorized implementation should be m
uch 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: 1.843227423994661 / 355.66429796093706 compute
d in 0.034364938735961914s
Vectorized loss / grad: 1.8432274239946609 / 321.1914085995837 computed
in 0.003840923309326172s
difference in loss / grad: 2.220446049250313e-16 /427.1678171924219
```

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:

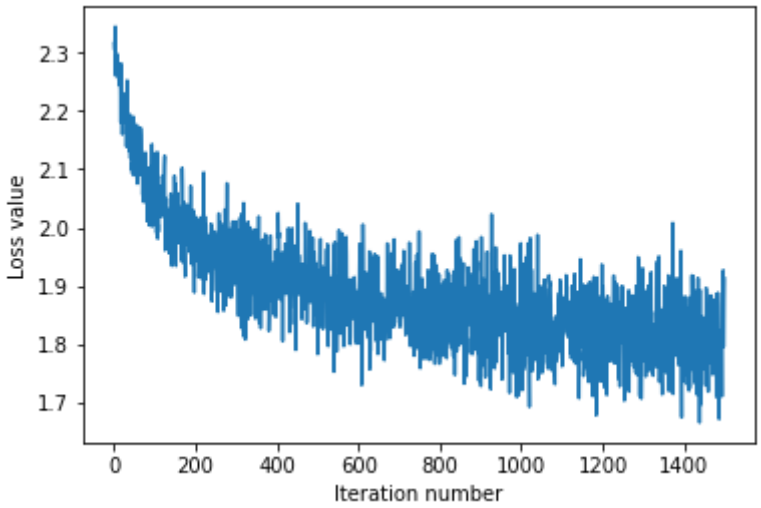
You fill this out.

In [21]:
```
# 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 2.316168370885791
iteration 100 / 1500: loss 2.116057123972999
iteration 200 / 1500: loss 1.9870478571324697
iteration 300 / 1500: loss 1.9791564604652032
iteration 400 / 1500: loss 1.882831676356327
iteration 500 / 1500: loss 1.7824244670893632
iteration 600 / 1500: loss 1.8007030338381145
iteration 700 / 1500: loss 1.8358270212594863
iteration 800 / 1500: loss 1.8114792230132881
iteration 900 / 1500: loss 1.7743496237590566
iteration 1000 / 1500: loss 1.8227673260549329
iteration 1100 / 1500: loss 1.7615025897697387
iteration 1200 / 1500: loss 1.8958686538899827
iteration 1300 / 1500: loss 1.8350510961395958
iteration 1400 / 1500: loss 1.8548663975072008
That took 8.6178719997406s
```



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

In [23]:
```
## Implement softmax.predict() and use it to compute the training and te
sting error.

y_train_pred = softmax.predict(X_train)
print('training accuracy: {}'.format(np.mean(np.equal(y_train,y_train_pr
ed), )))
y_val_pred = softmax.predict(X_val)
print('validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val_pre
d)), ))
```

```
training accuracy: 0.3821428571428571
validation accuracy: 0.39
```

Optimize the softmax classifier

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

In [26]:
```
np.finfo(float).eps
```

Out[26]:
```
2.220446049250313e-16
```

In [41]:
```
# ================================================================ #
# 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 validatio
n error.
#
#   Select the softmax that achieved the best validation error and repor
t
#     its error rate on the test set.
# ================================================================ #

learning_rates = np.array([1e-7, 5e-7, 1e-6, 5e-6, 1e-5, 5e-5, 1e-4, 5e-
4, 1e-3, 5e-3, 1e-2, 5e-2, 1e-1, 5e-1, 1, 5, 10])
val_accs = np.zeros_like(learning_rates)

for idx, learning_rate in enumerate(learning_rates):
    softmax.train(X_train, y_train, learning_rate=learning_rate)
    y_val_pred = softmax.predict(X_val)
    val_accs[idx] = np.mean(np.equal(y_val, y_val_pred))

best_val_acc_idx = np.argmax(val_accs)
best_val_acc = val_accs[best_val_acc_idx]
best_learning_rate = learning_rates[best_val_acc_idx]

print("The best learning rate is {}, with a validation accuracy of {}.".
format(best_learning_rate, best_val_acc))

softmax.train(X_train, y_train, learning_rate=best_learning_rate)
y_test_pred = softmax.predict(X_test)
test_acc = np.mean(np.equal(y_test, y_test_pred))

print("The best learning rate of {} had a test error of {}.".format(best
_learning_rate, 1 - test_acc))

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

```
/Users/jackzhang/GoogleDrive/UCLA/Academics/Junior/Q2/EC ENGR C247/HW2/
nndl/softmax.py:143: RuntimeWarning: divide by zero encountered in log
  loss = np.sum(np.log(np.sum(np.exp(scores).T, axis=0)) - scores[selec
tor]) / num_train
/Users/jackzhang/GoogleDrive/UCLA/Academics/Junior/Q2/EC ENGR C247/HW2/
nndl/softmax.py:147: RuntimeWarning: invalid value encountered in true_
divide
  temp = e_exp_a/sums[:, np.newaxis]
/Users/jackzhang/GoogleDrive/UCLA/Academics/Junior/Q2/EC ENGR C247/HW
2/.env/lib/python3.6/site-packages/numpy/core/_methods.py:26: RuntimeWa
rning: invalid value encountered in reduce
  return umr_maximum(a, axis, None, out, keepdims)

The best learning rate is 5e-07, with a validation accuracy of 0.356.
The best learning rate of 5e-07 had a test error of 0.646.
```