This is the svm workbook for ECE 239AS Assignment #2

Please follow the notebook linearly to implement a linear support vector machine.

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 includes code 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 an SVM classifier via gradient descent.

Importing libraries and data setup

In [1]:
```
import numpy as np # for doing most of our calculations
import matplotlib.pyplot as plt# for plotting
from cs231n.data_utils import load_CIFAR10 # function to load the CIFAR-10
import pdb

# Load matplotlib images inline
%matplotlib inline

# These are important for reloading any code you write in external .py file
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-i
%load_ext autoreload
%autoreload 2
```

In [2]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = 'cifar-10-batches-py'
X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

# As a sanity check, we print out the size of the training and test data.
print('Training data shape: ', X_train.shape)
print('Training labels shape: ', y_train.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

In [3]:

```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', '
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```

In [4]:

```python
# Split the data into train, val, and test sets. In addition we will
# create a small development set as a subset of the training data;
# we can use this for development so our code runs faster.
num_training = 49000
num_validation = 1000
num_test = 1000
num_dev = 500

# Our validation set will be num_validation points from the original
# training set.
mask = range(num_training, num_training + num_validation)
X_val = X_train[mask]
y_val = y_train[mask]

# Our training set will be the first num_train points from the original
# training set.
mask = range(num_training)
X_train = X_train[mask]
y_train = y_train[mask]

# We will also make a development set, which is a small subset of
# the training set.
mask = np.random.choice(num_training, num_dev, replace=False)
X_dev = X_train[mask]
y_dev = y_train[mask]

# We use the first num_test points of the original test set as our
# test set.
mask = range(num_test)
X_test = X_test[mask]
y_test = y_test[mask]

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, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
Dev data shape:  (500, 32, 32, 3)
Dev labels shape:  (500,)
```

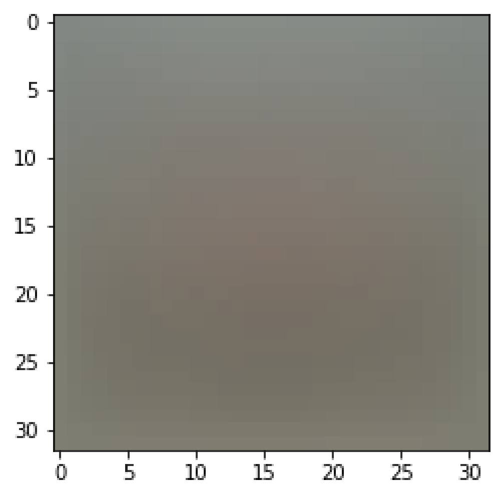
In [5]:
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# As a sanity check, print out the shapes of the data
print('Training data shape: ', X_train.shape)
print('Validation data shape: ', X_val.shape)
print('Test data shape: ', X_test.shape)
print('dev data shape: ', X_dev.shape)
```

```
Training data shape:  (49000, 3072)
Validation data shape:  (1000, 3072)
Test data shape:  (1000, 3072)
dev data shape:  (500, 3072)
```

In [6]:
```
# Preprocessing: subtract the mean image
# first: compute the image mean based on the training data
mean_image = np.mean(X_train, axis=0)
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the m
plt.show()
```

```
[130.64189796 135.98173469 132.47391837 130.05569388 135.34804082
 131.75402041 130.96055102 136.14328571 132.47636735 131.48467347]
```



In [7]:
```
# second: subtract the mean image from train and test data
X_train -= mean_image
X_val -= mean_image
X_test -= mean_image
X_dev -= mean_image
```

In [8]:
```
# third: append the bias dimension of ones (i.e. bias trick) so that our SV
# only has to worry about optimizing a single weight matrix W.
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))])

print(X_train.shape, X_val.shape, X_test.shape, X_dev.shape)
```

```
(49000, 3073) (1000, 3073) (1000, 3073) (500, 3073)
```

Question:

(1) For the SVM, we perform mean-subtraction on the data. However, for the KNN notebook, we did not. Why?

Answer:

(1) In SVM, we do mean-subtraction to "center" the data, which helps because in our SVM, we learn a linear classifier, where the slope of the line is what classifies. When the data is centered, the slope has a greater range of possible useful values, whereas if the data is not centered, the very small changes in the slope would have a large impact on the accuracy. KNN doesn't require this because KNN predicts by comparing against all training examples and not by using a linear classifier.

Training an SVM

The following cells will take you through building an SVM. 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 [9]:
```
from nndl.svm import SVM
```

In [10]:
```
# Declare an instance of the SVM class.
# Weights are initialized to a random value.
# Note, to keep people's initial solutions consistent, we are going to use

np.random.seed(1)

num_classes = len(np.unique(y_train))
num_features = X_train.shape[1]

svm = SVM(dims=[num_classes, num_features])
```

SVM loss

In [11]:
```
## Implement the loss function for in the SVM class(nndl/svm.py), svm.loss(

loss = svm.loss(X_train, y_train)
print('The training set loss is {}.'.format(loss))

# If you implemented the loss correctly, it should be 15569.98
```

```
The training set loss is 15569.977915410243.
```

SVM gradient

In [12]:
```
## Calculate the gradient of the SVM class.
# For convenience, we'll write one function that computes the loss
#   and gradient together. Please modify svm.loss_and_grad(X, y).
# You may copy and paste your loss code from svm.loss() here, and then
#   use the appropriate intermediate values to calculate the gradient.

loss, grad = svm.loss_and_grad(X_dev,y_dev)

# Compare your gradient to a numerical gradient check.
# You should see relative gradient errors on the order of 1e-07 or less if
svm.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: -2.237781 analytic: -2.237782, relative error: 1.370280e-07
numerical: -6.215028 analytic: -6.215029, relative error: 2.057846e-08
numerical: 3.514615 analytic: 3.514616, relative error: 2.669430e-09
numerical: 5.457790 analytic: 5.457790, relative error: 2.598255e-08
numerical: 2.472639 analytic: 2.472639, relative error: 9.816170e-08
numerical: -4.412693 analytic: -4.412693, relative error: 1.745507e-08
numerical: 15.471785 analytic: 15.471785, relative error: 9.692493e-10
numerical: -12.574649 analytic: -12.574649, relative error: 1.399030e-08
numerical: -3.287478 analytic: -3.287478, relative error: 7.007659e-08
numerical: -11.708351 analytic: -11.708351, relative error: 7.247005e-09
```

A vectorized version of SVM

To speed things up, we will vectorize the loss and gradient calculations. This will be helpful for stochastic gradient descent.

In [13]:
```
import time
```

In [14]:

```
## Implement svm.fast_loss_and_grad which calculates the loss and gradient
#    WITHOUT using any for loops.

# Standard loss and gradient
tic = time.time()
loss, grad = svm.loss_and_grad(X_dev, y_dev)
toc = time.time()
print('Normal loss / grad_norm: {} / {} computed in {}s'.format(loss, np.li

tic = time.time()
loss_vectorized, grad_vectorized = svm.fast_loss_and_grad(X_dev, y_dev)
toc = time.time()
print('Vectorized loss / grad: {} / {} computed in {}s'.format(loss_vectori

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

# You should notice a speedup with the same output, i.e., differences on th
```

```
Normal loss / grad_norm: 15795.36706359361 / 2205.996997125613 computed i
n 0.13952994346618652s
Vectorized loss / grad: 15795.367063593592 / 2205.996997125613 computed i
n 0.04236173629760742s
difference in loss / grad: 1.8189894035458565e-11 / 7.647796263659292e-12
```

Stochastic gradient descent

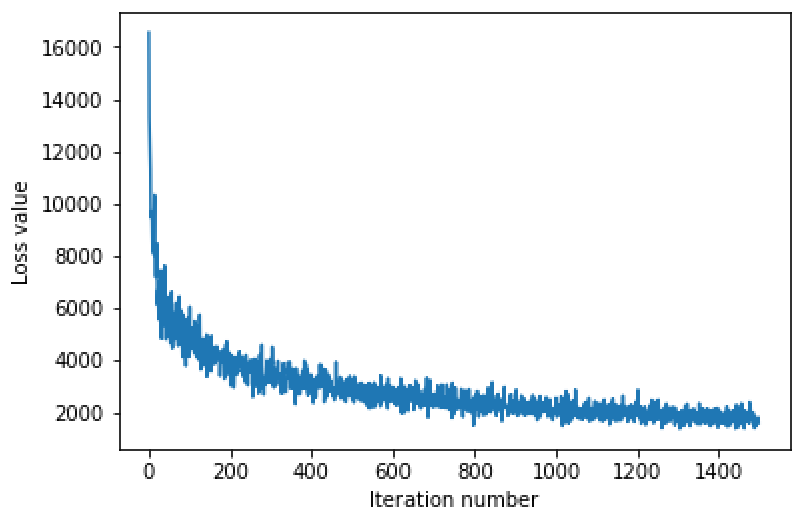
We now implement stochastic gradient descent. This uses the same principles of gradient descent we discussed in class, however, it calculates the gradient by only using examples from a subset of the training set (so each gradient calculation is faster).

In [15]:
```python
# Implement svm.train() by filling in the code to extract a batch of data
# and perform the gradient step.

tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=5e-4,
                      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 16557.38000190916
iteration 100 / 1500: loss 4701.089451272714
iteration 200 / 1500: loss 4017.333137942789
iteration 300 / 1500: loss 3681.9226471953616
iteration 400 / 1500: loss 2732.6164373988995
iteration 500 / 1500: loss 2786.637842464506
iteration 600 / 1500: loss 2837.0357842782664
iteration 700 / 1500: loss 2206.2348687399317
iteration 800 / 1500: loss 2269.0388241169803
iteration 900 / 1500: loss 2543.2378153859204
iteration 1000 / 1500: loss 2566.6921357268266
iteration 1100 / 1500: loss 2182.068905905164
iteration 1200 / 1500: loss 1861.1182244250451
iteration 1300 / 1500: loss 1982.9013858528256
iteration 1400 / 1500: loss 1927.5204158582108
That took 17.0645489692688s
```



Evaluate the performance of the trained SVM on the validation data.

In [16]:
```
## Implement svm.predict() and use it to compute the training and testing e

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

```
training accuracy: 0.28530612244897957
validation accuracy: 0.3
```

Optimize the SVM

Note, to make things faster and simpler, we won't do k-fold cross-validation, but will only optimize the hyperparameters on the validation dataset (X_val, y_val).

In [17]:
```python
# ================================================================ #
# YOUR CODE HERE:
#   Train the SVM 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 e
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
#   Note: You do not need to modify SVM class for this section
# ================================================================ #
for learning_rate in [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1]:
    print('learning rate: {}'.format(learning_rate))
    loss_hist = svm.train(X_train, y_train, learning_rate=learning_rate,
                      num_iters=1500, verbose=True)
    y_val_pred = svm.predict(X_val)
    print('validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val_pr
    print()

print('learning rate: {}'.format(0.01))
loss_hist = svm.train(X_train, y_train, learning_rate=0.01,
                      num_iters=1500, verbose=True)
y_test_pred = svm.predict(X_test)
print('test accuracy: {}'.format(np.mean(np.equal(y_test, y_test_pred)), ))
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
learning rate: 1e-06
iteration 0 / 1500: loss 17772.834341836522
iteration 100 / 1500: loss 14780.267680784524
iteration 200 / 1500: loss 15075.635289979482
iteration 300 / 1500: loss 14839.594710902102
iteration 400 / 1500: loss 13792.279947184972
iteration 500 / 1500: loss 13657.987712622155
iteration 600 / 1500: loss 12401.422542976215
iteration 700 / 1500: loss 14551.880696294287
iteration 800 / 1500: loss 11866.563137872732
iteration 900 / 1500: loss 11604.224785862243
iteration 1000 / 1500: loss 12401.100658679432
iteration 1100 / 1500: loss 12527.734572806692
iteration 1200 / 1500: loss 10772.40405009249
iteration 1300 / 1500: loss 11598.917808630456
iteration 1400 / 1500: loss 10195.066636726382
validation accuracy: 0.158

learning rate: 1e-05
iteration 0 / 1500: loss 23298.531266498594
iteration 100 / 1500: loss 14358.246986907223
iteration 200 / 1500: loss 13318.214004290094
iteration 300 / 1500: loss 9511.451188326495
iteration 400 / 1500: loss 9656.339130875695
iteration 500 / 1500: loss 8570.866583531062
iteration 600 / 1500: loss 10609.262599603697
iteration 700 / 1500: loss 7810.51591088207
```

```
iteration 800 / 1500: loss 7110.797153744181
iteration 900 / 1500: loss 7246.924312358823
iteration 1000 / 1500: loss 7631.926511942877
iteration 1100 / 1500: loss 7250.779561846437
iteration 1200 / 1500: loss 7091.836948900854
iteration 1300 / 1500: loss 7079.380840941409
iteration 1400 / 1500: loss 6363.476743143491
validation accuracy: 0.172

learning rate: 0.0001
iteration 0 / 1500: loss 17434.679795253487
iteration 100 / 1500: loss 7182.478280261159
iteration 200 / 1500: loss 6649.587592162183
iteration 300 / 1500: loss 5125.807516631966
iteration 400 / 1500: loss 5443.689542470026
iteration 500 / 1500: loss 4134.158509323795
iteration 600 / 1500: loss 3609.0871693979925
iteration 700 / 1500: loss 4065.9117916523855
iteration 800 / 1500: loss 3823.1231163265884
iteration 900 / 1500: loss 4367.424151822208
iteration 1000 / 1500: loss 4123.190459235119
iteration 1100 / 1500: loss 3499.7795586868697
iteration 1200 / 1500: loss 3624.974854069729
iteration 1300 / 1500: loss 3880.960353934499
iteration 1400 / 1500: loss 3877.5708952772356
validation accuracy: 0.249

learning rate: 0.001
iteration 0 / 1500: loss 17726.97708162432
iteration 100 / 1500: loss 3993.5313843936437
iteration 200 / 1500: loss 3302.868798593412
iteration 300 / 1500: loss 2918.841268364429
iteration 400 / 1500: loss 2501.9415467794665
iteration 500 / 1500: loss 2544.3633522725886
iteration 600 / 1500: loss 3014.148281082504
iteration 700 / 1500: loss 2813.957096554755
iteration 800 / 1500: loss 2557.0873015992775
iteration 900 / 1500: loss 1989.3584996672423
iteration 1000 / 1500: loss 2260.6714692896467
iteration 1100 / 1500: loss 1632.215258818069
iteration 1200 / 1500: loss 1780.0244682220027
iteration 1300 / 1500: loss 1849.7555073251692
iteration 1400 / 1500: loss 2120.8059634599686
validation accuracy: 0.266

learning rate: 0.01
iteration 0 / 1500: loss 17439.114360917345
iteration 100 / 1500: loss 20026.28540088886
iteration 200 / 1500: loss 17084.722833943473
iteration 300 / 1500: loss 28856.128334494573
iteration 400 / 1500: loss 11897.13339210161
iteration 500 / 1500: loss 13998.344330680375
iteration 600 / 1500: loss 15577.41818175864
iteration 700 / 1500: loss 13406.674685569573
iteration 800 / 1500: loss 13353.254296805462
iteration 900 / 1500: loss 11540.603600221883
iteration 1000 / 1500: loss 20529.126891335487
```

```
iteration 1100 / 1500: loss 18963.55037857191
iteration 1200 / 1500: loss 25299.747843521014
iteration 1300 / 1500: loss 17509.667212414766
iteration 1400 / 1500: loss 14270.853309556525
validation accuracy: 0.306

learning rate: 0.1
iteration 0 / 1500: loss 15370.641063153469
iteration 100 / 1500: loss 150542.27179736446
iteration 200 / 1500: loss 274117.2538056461
iteration 300 / 1500: loss 123071.95138949061
iteration 400 / 1500: loss 155051.4597033335
iteration 500 / 1500: loss 134865.15341798545
iteration 600 / 1500: loss 120845.00936800771
iteration 700 / 1500: loss 168879.85332251273
iteration 800 / 1500: loss 196772.13993255346
iteration 900 / 1500: loss 129220.75171202901
iteration 1000 / 1500: loss 155807.71852033742
iteration 1100 / 1500: loss 131514.71749096332
iteration 1200 / 1500: loss 141571.6569927082
iteration 1300 / 1500: loss 161303.4897508109
iteration 1400 / 1500: loss 122190.6488423454
validation accuracy: 0.262

learning rate: 1
iteration 0 / 1500: loss 17834.89814950957
iteration 100 / 1500: loss 1158266.333167801
iteration 200 / 1500: loss 1786261.3171935224
iteration 300 / 1500: loss 1724946.6334517535
iteration 400 / 1500: loss 1140259.1475829678
iteration 500 / 1500: loss 1090646.4067051795
iteration 600 / 1500: loss 1843775.6248034923
iteration 700 / 1500: loss 1389840.3792271346
iteration 800 / 1500: loss 1684029.6508396333
iteration 900 / 1500: loss 2311462.6616987674
iteration 1000 / 1500: loss 1759591.1423630107
iteration 1100 / 1500: loss 936372.791510725
iteration 1200 / 1500: loss 1209672.535284187
iteration 1300 / 1500: loss 1598376.8037788253
iteration 1400 / 1500: loss 1501289.462830457
validation accuracy: 0.289

learning rate: 0.01
iteration 0 / 1500: loss 15754.992051210502
iteration 100 / 1500: loss 18381.617872154166
iteration 200 / 1500: loss 11726.910059578619
iteration 300 / 1500: loss 10283.978413134835
iteration 400 / 1500: loss 15012.203770867709
iteration 500 / 1500: loss 13323.311549204533
iteration 600 / 1500: loss 14413.097266453857
iteration 700 / 1500: loss 14467.422483423972
iteration 800 / 1500: loss 17488.75701978225
iteration 900 / 1500: loss 10194.2939965192
iteration 1000 / 1500: loss 9706.228906685465
iteration 1100 / 1500: loss 10518.140099153923
iteration 1200 / 1500: loss 23726.206699223956
iteration 1300 / 1500: loss 7353.385283170138
```

```
iteration 1400 / 1500: loss 15801.919263539872
test accuracy: 0.259
```