This is the svm workbook for ECE C147/C247 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 CIFA
R-10 dataset.
import pdb

# Load matplotlib images inline
%matplotlib inline

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

In [2]:
```
# Set the path to the 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)

# As a sanity check, we print out the size of the training and test da
ta.
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', 'hors
e', 'ship', 'truck']
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]:

```
# 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 origina
l
# 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,)
```

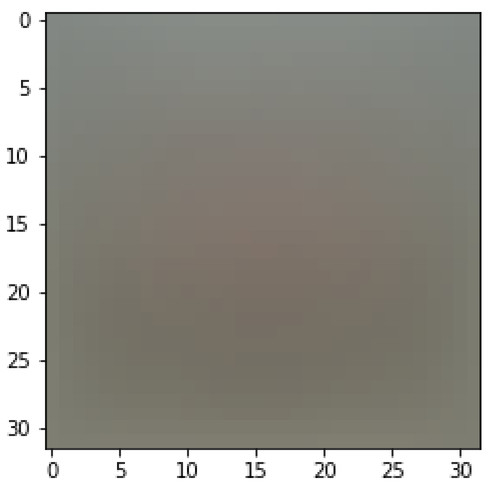
In [5]:
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# 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 mean image
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 o
ur SVM
# 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) The training step of a Support Vector Machine is $W = W - \epsilon \cdot \nabla_W Loss$, where $\nabla_W Loss$ relies linearly on the features of the examples in the training set (except when the loss is 0 occasionally). For that reason, SVM training relies heavily on the magnitude of features, especially because all features are multiplied by the same learning rate. Thus, if the features are of large magnitue, the corresponding weights will also possibly be of high magnitude. Therefore, features of high magnitude will dominate the score of an example, preventing from other low-magnitude features to possibly give significant information about the score/classification of the example. Therefore, subtracting the mean of the dataset can help avoid this possible domination of high-magnitude features.

In KNN there's no significance to the feature scale because Euclidean depends merely on the distance between vectors. Therefore, mean subtraction is not necessary for KNN.

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 a random seed.

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 15569.97791541019.
```

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 les
s if you implemented the gradient correctly.
svm.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: 0.167121 analytic: 0.167121, relative error: 8.579596e-07
numerical: -2.101728 analytic: -2.101729, relative error: 1.953189e-07
numerical: 2.748654 analytic: 2.748655, relative error: 8.216914e-08
numerical: 7.120861 analytic: 7.120860, relative error: 5.563664e-08
numerical: -0.037178 analytic: -0.037178, relative error: 4.921029e-06
numerical: 11.391076 analytic: 11.391076, relative error: 1.297940e-08
numerical: 6.793050 analytic: 6.793050, relative error: 4.358543e-08
numerical: -16.868199 analytic: -16.868199, relative error: 9.581999e-
11
numerical: -12.957866 analytic: -12.957865, relative error: 1.211635e-
08
numerical: -15.639322 analytic: -15.639322, relative error: 1.755329e-
08
```

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 grad
ient
#    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.linalg.norm(grad, 'fro'), toc - tic))

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_ve
ctorized, np.linalg.norm(grad_vectorized, 'fro'), toc - tic))

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

# You should notice a speedup with the same output, i.e., differences
 on the order of 1e-12
```

```
Normal loss / grad_norm: 15057.810738219052 / 2190.3996534895227 compu
ted in 0.07339715957641602s
Vectorized loss / grad: 15057.810738219017 / 2190.399653489523 compute
d in 0.004544496536254883s
difference in loss / grad: 3.456079866737127e-11 / 3.0849259899516946e
-12
```

Stochastic gradient descent

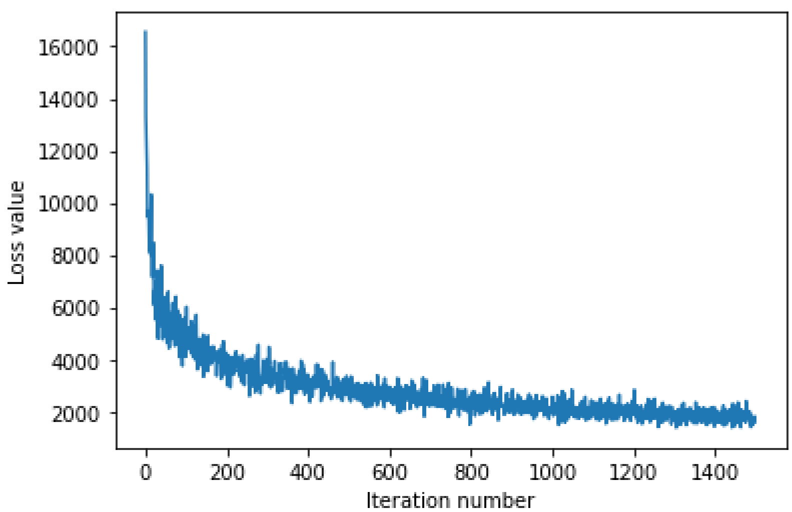
We now implement stochastic gradient descent. This uses the same principles of gradient descent we discussed in class, however, it calculates the gradient by only using examples from a subset of the training set (so each gradient calculation is faster).

In [15]:
```
# Implement svm.train() by filling in the code to extract a batch of d
ata
# 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 4701.089451272713
iteration 200 / 1500: loss 4017.333137942789
iteration 300 / 1500: loss 3681.922647195362
iteration 400 / 1500: loss 2732.6164373988995
iteration 500 / 1500: loss 2786.6378424645054
iteration 600 / 1500: loss 2837.035784278267
iteration 700 / 1500: loss 2206.2348687399326
iteration 800 / 1500: loss 2269.03882411698
iteration 900 / 1500: loss 2543.2378153859195
iteration 1000 / 1500: loss 2566.6921357268266
iteration 1100 / 1500: loss 2182.068905905164
iteration 1200 / 1500: loss 1861.1182244250451
iteration 1300 / 1500: loss 1982.9013858528258
iteration 1400 / 1500: loss 1927.5204158582117
That took 5.560702562332153s
```



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

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

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_p
red)), ))
```

```
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]:
```
# ================================================================ #
# 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 VALIDAT
ION error.
#
#   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
# ================================================================ #
best_learning_rate = 0
best_val_accur = 0
best_test_accur = 0
for i in range(1, 10000, 200):
    learning_rate = i*1e-5
    svm.train(X_train, y_train, learning_rate=learning_rate,
                      num_iters=500, verbose=False)
    y_train_pred = svm.predict(X_train)
    train_accur = np.mean(np.equal(y_train,y_train_pred))
    y_val_pred = svm.predict(X_val)
    val_accur = np.mean(np.equal(y_val, y_val_pred))

    if val_accur > best_val_accur:
        best_val_accur = val_accur
        best_learning_rate = learning_rate

svm.train(X_train, y_train, learning_rate=best_learning_rate,
                      num_iters=500, verbose=False)

y_test_pred = svm.predict(X_test)
test_error = 1 - np.mean(np.equal(y_test, y_test_pred))

print('best validation accuracy: {}'.format(best_val_accur))
print('best learning rate: {}'.format(best_learning_rate))
print('test set error using the best learning rate: {}'.format(test_er
ror))
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
best validation accuracy: 0.341
best learning rate: 0.02001
test set error using the best learning rate: 0.7110000000000001
```