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 [58]:

```
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 datas
et.
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-ipytho
n
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

In [59]:

```python
# Set the path to the CIFAR-10 data
import os
cur_dir = os.getcwd()
cifar10_dir = cur_dir+'/cs231n/datasets/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 [60]:

```
# 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', '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 [61]:

```
# 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 [62]:

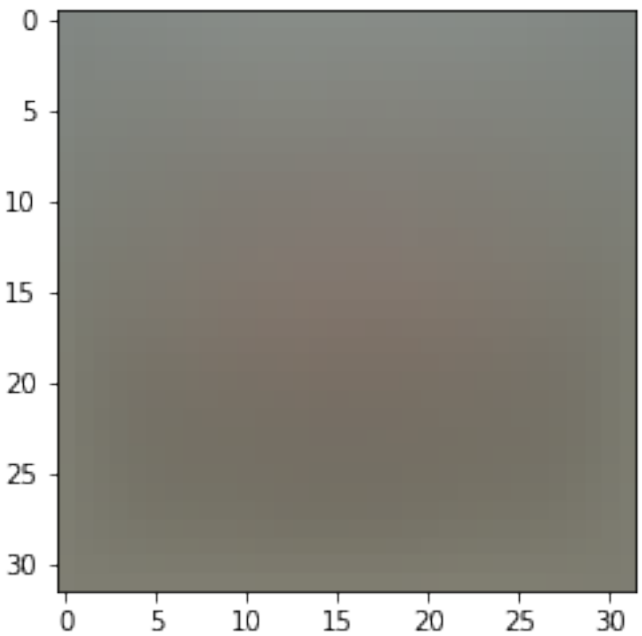
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# 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 [63]:

```
# 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 i
mage
plt.show()
```

```
[ 130.64189796  135.98173469  132.47391837  130.05569388  135.348040
82
  131.75402041  130.96055102  136.14328571  132.47636735  131.484673
47]
```



In [64]:

```
# 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 [65]:

```
# third: append the bias dimension of ones (i.e. bias trick) so that our 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) Mean subtraction is a method of feature scaling that would not have an effect for K-NN. Given some "mean vector", subtracting this from every data point within our training set would correspondingly shift everything by the same magnitude and direction. The nearest neighbors would stay the same, and the outputs of KNN before and after mean subtraction would be the same. Therefore, it would be an unnecessary extra computation step.

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 [66]:

```
from nndl.svm import SVM
```

In [67]:

```
# 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 ran
dom seed.

np.random.seed(1)

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

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

```
3073 10
```

SVM loss

In [68]:

```
## 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 [69]:

```
## 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 you i
mplemented the gradient correctly.
svm.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: -14.486718 analytic: -14.486717, relative error: 3.426800
e-08
numerical: -4.304328 analytic: -4.304329, relative error: 3.317946e-
08
numerical: -1.540845 analytic: -1.540846, relative error: 2.005298e-
07
numerical: 6.694719 analytic: 6.694720, relative error: 4.180430e-08
numerical: -0.745168 analytic: -0.745169, relative error: 7.002595e-
07
numerical: 11.416845 analytic: 11.416845, relative error: 4.692967e-
09
numerical: 7.342247 analytic: 7.342248, relative error: 4.025915e-08
numerical: -6.535749 analytic: -6.535748, relative error: 6.080351e-
08
numerical: -1.621508 analytic: -1.621507, relative error: 3.155933e-
07
numerical: -13.477379 analytic: -13.477380, relative error: 4.728766
e-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 [70]:

```
import time
```

In [71]:

```
## 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)
#print(grad.shape)
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)
#print(grad_vectorized.shape)
toc = time.time()
print('Vectorized loss / grad: {} / {} computed in {}s'.format(loss_vectorized,
np.linalg.norm(grad_vectorized, 'fro'), toc - tic))

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

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

```
Normal loss / grad_norm: 15807.88880779353 / 2229.005182658606 compu
ted in 0.02430582046508789s
Vectorized loss / grad: 15807.88880779355 / 2229.005182658606 comput
ed in 0.008964061737060547s
difference in loss / grad: -2.000888343900442e-11 / 7.17282901881172
2e-12
```

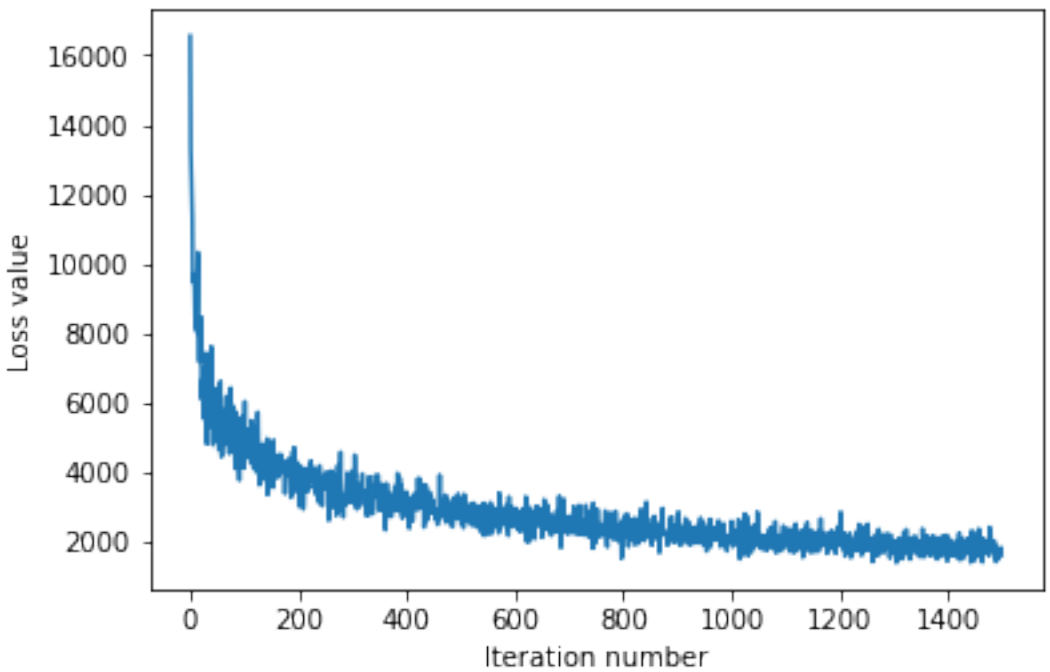
Stochastic gradient descent

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 [72]:

```
# 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 4017.333137942788
iteration 300 / 1500: loss 3681.9226471953616
iteration 400 / 1500: loss 2732.6164373988995
iteration 500 / 1500: loss 2786.6378424645054
iteration 600 / 1500: loss 2837.035784278267
iteration 700 / 1500: loss 2206.234868739932
iteration 800 / 1500: loss 2269.0388241169803
iteration 900 / 1500: loss 2543.23781538592
iteration 1000 / 1500: loss 2566.692135726825
iteration 1100 / 1500: loss 2182.068905905163
iteration 1200 / 1500: loss 1861.1182244250458
iteration 1300 / 1500: loss 1982.901385852825
iteration 1400 / 1500: loss 1927.520415858212
That took 6.337159156799316s
```



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

In [73]:

```
## Implement svm.predict() and use it to compute the training and testing error.

y_train_pred = svm.predict(X_train)
print(y_train_pred)
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)), ))
```

```
[6 1 9 ..., 2 1 9]
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 [74]:

```
# ================================================================ #
# 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 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
# ================================================================ #
def sweep_lr(num_iters):
    learning_rates = [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1e0, 1e1, 1e2, 1e3,
                      1e4, 1e5, 1e6]

    results = {}
    for lr in learning_rates:
        np.random.seed(1)

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

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

        loss_hist = svm.train(X_train, y_train, learning_rate=lr,
                              num_iters=num_iters, verbose=False)

        y_val_pred = svm.predict(X_val)
        val_accuracy = np.mean(np.equal(y_val, y_val_pred))
        results[lr] = val_accuracy

    return results


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

Learning rates were chosen based on order of magnitude (sweeping from 10^-6 to 10-6).

For max_iters=1500, the optimal learning rate is 1.

It's possible that with too small of a learning rate and a num_iter value that is too small, gradient descent will not converge to the global min. This is why it makes sense to sweep values for max_iters until convergence, or to add some epsilon term that determines when to exit gradient descent. The max_iters sweep below was purely experimental.

In [75]:

```
#Find best learning rate and try a couple number max_iters
max_iters = [1500, 2500, 3500]
for max_iter in max_iters:
    print("Testing for max_iter=", max_iter,"...")
    results = sweep_lr(max_iter)
    max_acc = 0
    best_lr = -1
    for key, value in results.items():
        if(value > max_acc):
            max_acc = value
            best_lr = key

    print(results)
    print("Best learning rate is: ", best_lr)
    print("Accuracy is: ", max_acc )
```

```
Testing for max_iter= 1500 ...
{1e-06: 0.13300000000000001, 1e-05: 0.22, 0.0001: 0.2710000000000000
2, 0.001: 0.28499999999999998, 0.01: 0.29299999999999998, 0.1: 0.298
99999999999999, 1.0: 0.315, 10.0: 0.30299999999999999, 100.0: 0.2849
9999999999998, 1000.0: 0.28399999999999997, 10000.0: 0.2959999999999
9999, 100000.0: 0.311, 1000000.0: 0.311}
Best learning rate is:  1.0
Accuracy is:  0.315
Testing for max_iter= 2500 ...
{1e-06: 0.155, 1e-05: 0.23699999999999999, 0.0001: 0.281000000000000
03, 0.001: 0.25800000000000001, 0.01: 0.29499999999999998, 0.1: 0.29
499999999999998, 1.0: 0.29199999999999998, 10.0: 0.29099999999999998
, 100.0: 0.29799999999999999, 1000.0: 0.29399999999999998, 10000.0:
0.29899999999999999, 100000.0: 0.308, 1000000.0: 0.308}
Best learning rate is:  100000.0
Accuracy is:  0.308
Testing for max_iter= 3500 ...
{1e-06: 0.16800000000000001, 1e-05: 0.24099999999999999, 0.0001: 0.2
7500000000000002, 0.001: 0.29999999999999999, 0.01: 0.34300000000000
003, 0.1: 0.35699999999999998, 1.0: 0.30299999999999999, 10.0: 0.320
00000000000001, 100.0: 0.307, 1000.0: 0.34200000000000003, 10000.0:
0.32400000000000001, 100000.0: 0.312, 1000000.0: 0.312}
Best learning rate is:  0.1
Accuracy is:  0.357
```