Multiclass Support Vector Machine exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission.

In this exercise you will:

- implement a fully-vectorized **loss function** for the SVM
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** using numerical gradient
- use a validation set to **tune the learning rate and regularization** strength
- **optimize** the loss function with **SGD**
- **visualize** the final learned weights

In [1]:
```
from __future__ import print_function

import random
import numpy as np
from cecs551.data_utils import load_CIFAR10
import matplotlib.pyplot as plt


# This is a bit of magic to make matplotlib figures appear inline in the
# notebook rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipyt
hon
%load_ext autoreload
%autoreload 2
```

CIFAR-10 Data Loading and Preprocessing

In [2]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cecs551/datasets/cifar-10-batches-py'

# Cleaning up variables to prevent loading data multiple times (which may caus
e memory issue)
try:
   del X_train, y_train
   del X_test, y_test
   print('Clear previously loaded data.')
except:
   pass

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', 'shi
p', '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 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)
```

```
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,)
```

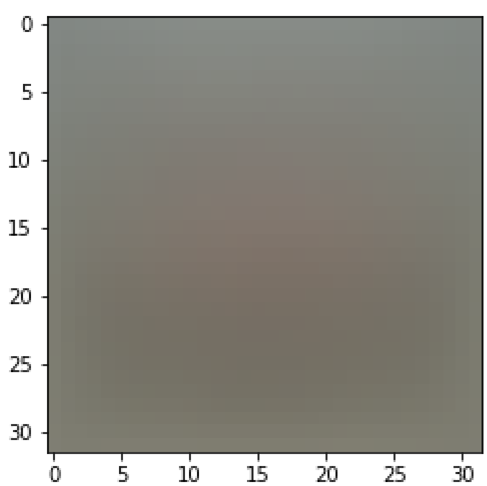
In [5]:
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# 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 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)
```

SVM Classifier

Your code for this section will all be written inside **cecs551/classifiers/linear_svm.py**.

As you can see, we have prefilled the function `compute_loss_naive` which uses for loops to evaluate the multiclass SVM loss function.

In [9]:
```
# Evaluate the naive implementation of the loss we provided for you:
from cecs551.classifiers.linear_svm import svm_loss_naive
import time

# generate a random SVM weight matrix of small numbers
W = np.random.randn(3073, 10) * 0.0001

loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.000005)
print('loss: %f' % (loss, ))
```

```
loss: 9.316358
```

The `grad` returned from the function above is right now all zero. Derive and implement the gradient for the SVM cost function and implement it inline inside the function `svm_loss_naive` . You will find it helpful to interleave your new code inside the existing function.

To check that you have correctly implemented the gradient correctly, you can numerically estimate the gradient of the loss function and compare the numeric estimate to the gradient that you computed. We have provided code that does this for you:

In [10]:
```
# Once you've implemented the gradient, recompute it with the code below
# and gradient check it with the function we provided for you

# Compute the loss and its gradient at W.
loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.0)

# Numerically compute the gradient along several randomly chosen dimensions, a
nd
# compare them with your analytically computed gradient. The numbers should ma
tch
# almost exactly along all dimensions.
from cecs551.gradient_check import grad_check_sparse
f = lambda w: svm_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad)

# do the gradient check once again with regularization turned on
# you didn't forget the regularization gradient did you?
loss, grad = svm_loss_naive(W, X_dev, y_dev, 5e1)
f = lambda w: svm_loss_naive(w, X_dev, y_dev, 5e1)[0]
grad_numerical = grad_check_sparse(f, W, grad)
```

```
numerical: -6.686750 analytic: -6.648966, relative error: 2.833305e-03
numerical: -29.336187 analytic: -29.336187, relative error: 4.008040e-12
numerical: 21.423688 analytic: 21.423688, relative error: 1.440417e-11
numerical: 13.521693 analytic: 13.521693, relative error: 7.488535e-12
numerical: -1.244246 analytic: -1.244246, relative error: 2.128594e-10
numerical: -0.846536 analytic: -0.846536, relative error: 4.356676e-10
numerical: 0.705510 analytic: 0.705510, relative error: 4.952164e-10
numerical: -40.011552 analytic: -40.011552, relative error: 7.523896e-12
numerical: -0.613682 analytic: -0.613682, relative error: 4.993237e-11
numerical: -34.322795 analytic: -34.352748, relative error: 4.361578e-04
numerical: 11.907451 analytic: 11.907451, relative error: 1.651418e-11
numerical: -24.289250 analytic: -24.289250, relative error: 1.625172e-12
numerical: -0.155111 analytic: -0.155111, relative error: 8.399491e-10
numerical: 8.377999 analytic: 8.377999, relative error: 1.567903e-11
numerical: -49.428589 analytic: -49.435664, relative error: 7.156533e-05
numerical: 9.539944 analytic: 9.572989, relative error: 1.728944e-03
numerical: -3.851097 analytic: -3.851097, relative error: 1.617800e-10
numerical: 24.682784 analytic: 24.682784, relative error: 1.677691e-11
numerical: -8.989510 analytic: -8.989510, relative error: 3.738487e-11
numerical: -20.245525 analytic: -20.245525, relative error: 2.952914e-12
```

Inline Question 1:

It is possible that once in a while a dimension in the gradcheck will not match exactly. What could such a discrepancy be caused by? Is it a reason for concern? What is a simple example in one dimension where a gradient check could fail? How would change the margin affect of the frequency of this happening? *Hint: the SVM loss function is not strictly speaking differentiable*

Your Answer: The SVM function is not strictly differentiable, the point is at the very hinge of the loss function. For [;f(x) = max(-x,0);] at x=0, there is no real gradient. For example, if we grad check at x=0.001, our computation will return 0 and at x=-0.001, our computation will return -1. However, the numerical computation does a finite approximation. Therefore They do not have gradients 0 or 1.

In [11]:
```
# Next implement the function svm_loss_vectorized; for now only compute the lo
ss;
# we will implement the gradient in a moment.
tic = time.time()
loss_naive, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from cecs551.classifiers.linear_svm import svm_loss_vectorized
tic = time.time()
loss_vectorized, _ = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

# The losses should match but your vectorized implementation should be much fa
ster.
print('difference: %f' % (loss_naive - loss_vectorized))
```

```
Naive loss: 9.316358e+00 computed in 0.199491s
Vectorized loss: 9.316358e+00 computed in 0.015736s
difference: -0.000000
```

In [12]:
```
# Complete the implementation of svm_loss_vectorized, and compute the gradient
# of the loss function in a vectorized way.

# The naive implementation and the vectorized implementation should match, but
# the vectorized version should still be much faster.
tic = time.time()
_, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss and gradient: computed in %fs' % (toc - tic))

tic = time.time()
_, grad_vectorized = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss and gradient: computed in %fs' % (toc - tic))

# The loss is a single number, so it is easy to compare the values computed
# by the two implementations. The gradient on the other hand is a matrix, so
# we use the Frobenius norm to compare them.
difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('difference: %f' % difference)
```

```
Naive loss and gradient: computed in 0.186156s
Vectorized loss and gradient: computed in 0.016785s
difference: 0.000000
```

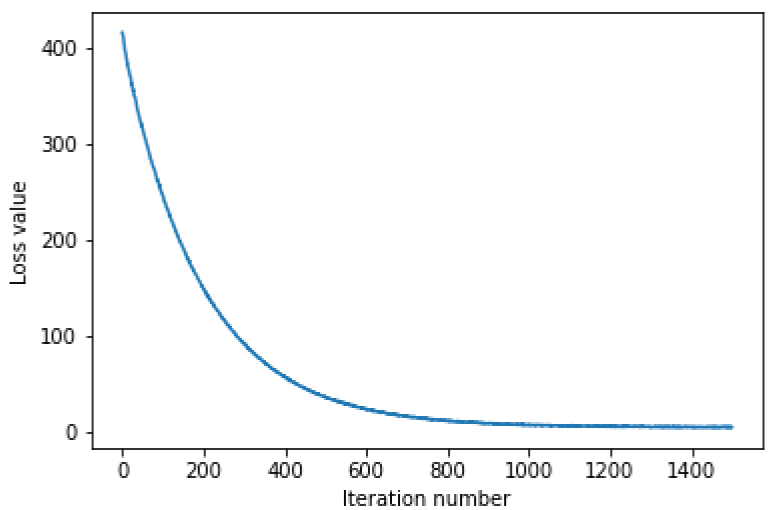
Stochastic Gradient Descent

We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss.

In [13]:
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cecs551.classifiers import LinearSVM
svm = LinearSVM()
tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 415.347342
iteration 100 / 1500: loss 245.124280
iteration 200 / 1500: loss 148.209626
iteration 300 / 1500: loss 91.510294
iteration 400 / 1500: loss 57.082685
iteration 500 / 1500: loss 35.987077
iteration 600 / 1500: loss 23.738575
iteration 700 / 1500: loss 16.347048
iteration 800 / 1500: loss 11.854580
iteration 900 / 1500: loss 9.113707
iteration 1000 / 1500: loss 7.397859
iteration 1100 / 1500: loss 7.034450
iteration 1200 / 1500: loss 6.164279
iteration 1300 / 1500: loss 5.282797
iteration 1400 / 1500: loss 5.329524
That took 12.185249s
```

In [14]:
```
# A useful debugging strategy is to plot the loss as a function of
# iteration number:
plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```



In [15]:
```
# Write the LinearSVM.predict function and evaluate the performance on both th
e
# training and validation set
y_train_pred = svm.predict(X_train)
print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
y_val_pred = svm.predict(X_val)
print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

```
training accuracy: 0.382082
validation accuracy: 0.394000
```

In [16]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.4 on the validation set.
learning_rates = [1e-7, 5e-5]
regularization_strengths = [2.5e4, 5e4]

# results is dictionary mapping tuples of the form
# (learning_rate, regularization_strength) to tuples of the form
# (training_accuracy, validation_accuracy). The accuracy is simply the fractio
n
# of data points that are correctly classified.
results = {}
best_val = -1   # The highest validation accuracy that we have seen so far.
best_svm = None # The LinearSVM object that achieved the highest validation ra
te.

###############################################################################
##
# TODO:
#
# Write code that chooses the best hyperparameters by tuning on the validation
#
# set. For each combination of hyperparameters, train a linear SVM on the
#
# training set, compute its accuracy on the training and validation sets, and
#
# store these numbers in the results dictionary. In addition, store the best
#
# validation accuracy in best_val and the LinearSVM object that achieves this
#
# accuracy in best_svm.
#
#
#
# Hint: You should use a small value for num_iters as you develop your
#
# validation code so that the SVMs don't take much time to train; once you are
#
# confident that your validation code works, you should rerun the validation
#
# code with a larger value for num_iters.
#
###############################################################################
##
for learning in learning_rates:
    for regularization in regularization_strengths:
        svm = LinearSVM()
        svm.train(X_train, y_train, learning_rate=learning, reg=regularization
,
                              num_iters=2000)
        y_train_pred = svm.predict(X_train)
        train_accuracy = np.mean(y_train == y_train_pred)
        y_val_pred = svm.predict(X_val)
        val_accuracy = np.mean(y_val == y_val_pred)
```

```
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_svm = svm

        results[(learning, regularization)] = (train_accuracy, val_accuracy)
##############################################################################
##
#                              END OF YOUR CODE                              
#
##############################################################################
##

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

print('best validation accuracy achieved during cross-validation: %f' % best_v
al)
```

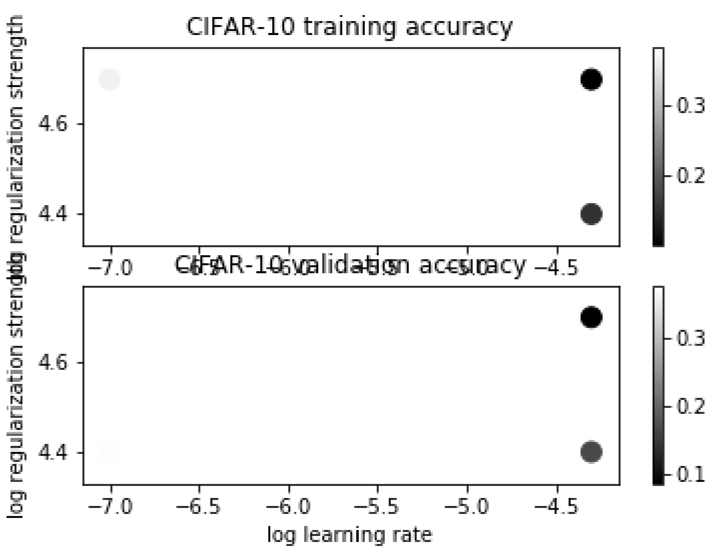
```
E:\CSULB\Spring 19\Adv AI\HW2\HW2\assignment2\cecs551\classifiers\linear_svm.
py:81: RuntimeWarning: overflow encountered in double_scalars
  loss += 0.5 * reg * np.sum(W * W)
C:\Users\abbaz\Anaconda\envs\cecs551\lib\site-packages\numpy\core\fromnumeri
c.py:83: RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
E:\CSULB\Spring 19\Adv AI\HW2\HW2\assignment2\cecs551\classifiers\linear_svm.
py:81: RuntimeWarning: overflow encountered in multiply
  loss += 0.5 * reg * np.sum(W * W)
E:\CSULB\Spring 19\Adv AI\HW2\HW2\assignment2\cecs551\classifiers\linear_svm.
py:78: RuntimeWarning: overflow encountered in subtract
  margins = np.maximum(scores - correct_scores.reshape(N, 1) + 1.0, 0)  # (N,
C)
E:\CSULB\Spring 19\Adv AI\HW2\HW2\assignment2\cecs551\classifiers\linear_svm.
py:78: RuntimeWarning: invalid value encountered in subtract
  margins = np.maximum(scores - correct_scores.reshape(N, 1) + 1.0, 0)  # (N,
C)
E:\CSULB\Spring 19\Adv AI\HW2\HW2\assignment2\cecs551\classifiers\linear_svm.
py:78: RuntimeWarning: invalid value encountered in maximum
  margins = np.maximum(scores - correct_scores.reshape(N, 1) + 1.0, 0)  # (N,
C)
E:\CSULB\Spring 19\Adv AI\HW2\HW2\assignment2\cecs551\classifiers\linear_svm.
py:99: RuntimeWarning: invalid value encountered in greater
  dscores[margins > 0] = 1
E:\CSULB\Spring 19\Adv AI\HW2\HW2\assignment2\cecs551\classifiers\linear_svm.
py:104: RuntimeWarning: overflow encountered in multiply
  dW += reg * W
E:\CSULB\Spring 19\Adv AI\HW2\HW2\assignment2\cecs551\classifiers\linear_clas
sifier.py:71: RuntimeWarning: invalid value encountered in subtract
  self.W -= learning_rate * grad

lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.381306 val accuracy: 0.374
000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.365490 val accuracy: 0.377
000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.154571 val accuracy: 0.169
000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087
000
best validation accuracy achieved during cross-validation: 0.377000
```

In [17]:
```
# Visualize the cross-validation results
import math
x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

# plot training accuracy
marker_size = 100
colors = [results[x][0] for x in results]
plt.subplot(2, 1, 1)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 training accuracy')

# plot validation accuracy
colors = [results[x][1] for x in results] # default size of markers is 20
plt.subplot(2, 1, 2)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.show()
```



In [18]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.373000
```

In [19]:
```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, these may
# or may not be nice to look at.
w = best_svm.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)
w_min, w_max = np.min(w), np.max(w)
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
for i in range(10):
    plt.subplot(2, 5, i + 1)

    # Rescale the weights to be between 0 and 255
    wimg = 255.0 * (w[:, :, :, i].squeeze() - w_min) / (w_max - w_min)
    plt.imshow(wimg.astype('uint8'))
    plt.axis('off')
    plt.title(classes[i])
```

Inline question 2:

Describe what your visualized SVM weights look like, and offer a brief explanation for why they look they way that they do.

Your answer: It seems like each weight represents mean image of each class.

In [ ]:
```
```