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]:
```
# Run some setup code for this notebook.

import random
import numpy as np
from hmwk5_2.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

from __future__ import print_function

# 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
%load_ext autoreload
%autoreload 2
```

CIFAR-10 Data Loading and Preprocessing

In [2]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'hmwk5_2/datasets/cifar-10-batches-py'

# Cleaning up variables to prevent loading data multiple times (which may caus
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
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)
```

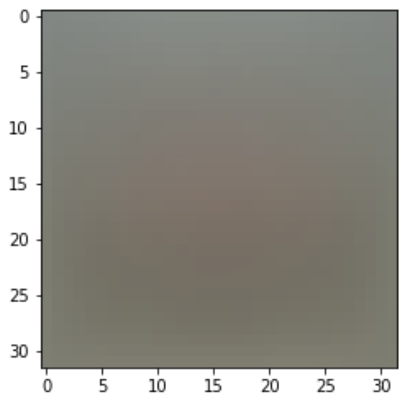
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

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
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 **hmwk5_2/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 hmwk5_2.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
print(X_dev.shape)
loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.000005)
print('loss: %f' % (loss, ))
```

```
(500, 3073)
loss: 8.598005
```

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
# compare them with your analytically computed gradient. The numbers should ma
# almost exactly along all dimensions.
from hmwk5_2.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: -32.737050 analytic: -32.737050, relative error: 8.188690e-12
numerical: 5.440826 analytic: 5.440826, relative error: 1.164088e-12
numerical: 34.547373 analytic: 34.547373, relative error: 2.569872e-13
numerical: -1.308320 analytic: -1.308320, relative error: 1.343601e-10
numerical: 15.178077 analytic: 15.178077, relative error: 1.209626e-11
numerical: -55.327355 analytic: -55.327355, relative error: 1.824280e-13
numerical: -43.831706 analytic: -43.831706, relative error: 5.356259e-12
numerical: -34.821779 analytic: -34.821779, relative error: 6.670741e-12
numerical: 12.243454 analytic: 12.243454, relative error: 8.932743e-12
numerical: 4.117446 analytic: 4.117446, relative error: 9.784219e-12
numerical: -9.035023 analytic: -9.035023, relative error: 2.897096e-11
numerical: 1.694358 analytic: 1.694358, relative error: 2.223191e-10
numerical: 14.880891 analytic: 14.880891, relative error: 1.300280e-11
numerical: -51.455663 analytic: -51.455663, relative error: 5.518977e-12
numerical: 20.810688 analytic: 20.810688, relative error: 5.151691e-12
numerical: -18.777012 analytic: -18.777012, relative error: 1.357474e-11
numerical: -66.779038 analytic: -66.779038, relative error: 6.117587e-12
numerical: 1.274940 analytic: 1.274940, relative error: 1.239979e-10
numerical: -4.733368 analytic: -4.733368, relative error: 1.264388e-11
numerical: -32.831151 analytic: -32.831151, relative error: 1.727058e-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 gradient check may fail for example in the presence of sharp twist in the weight function. An example cause of such a sharp twist in the loss function is the max()-in SVM-loss. Also, if the function is not differentiable then check fails.*

In [11]:
```
# Next implement the function svm_loss_vectorized; for now only compute the lo
# we will implement the gradient in a moment.
tic = time.time()
delta = 1
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 hmwk5_2.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
print('difference: %f' % (loss_naive - loss_vectorized))
```

```
Naive loss: 8.598005e+00 computed in 0.099823s
Vectorized loss: 8.598005e+00 computed in 0.004018s
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 0.088018s
Vectorized loss and gradient: computed in 0.003861s
difference: 0.000000
```

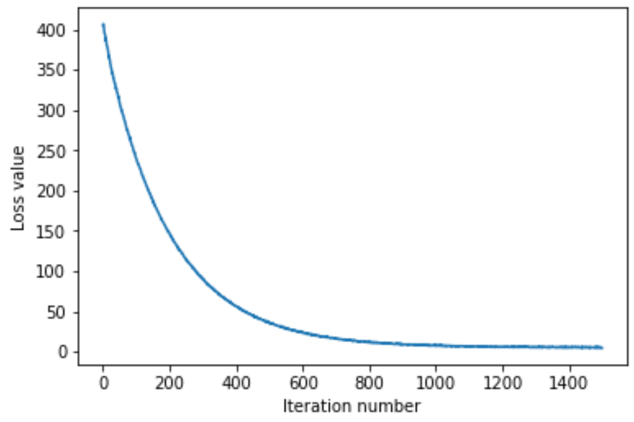
Stochastic Gradient Descent

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 hmwk5_2.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 404.784679
iteration 100 / 1500: loss 239.489670
iteration 200 / 1500: loss 146.610467
iteration 300 / 1500: loss 90.583913
iteration 400 / 1500: loss 56.171996
iteration 500 / 1500: loss 35.135266
iteration 600 / 1500: loss 24.227424
iteration 700 / 1500: loss 16.448077
iteration 800 / 1500: loss 12.000115
iteration 900 / 1500: loss 8.872656
iteration 1000 / 1500: loss 7.223346
iteration 1100 / 1500: loss 6.419222
iteration 1200 / 1500: loss 5.553589
iteration 1300 / 1500: loss 5.450153
iteration 1400 / 1500: loss 5.556843
That took 4.367522s
```

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
# 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), ))
```

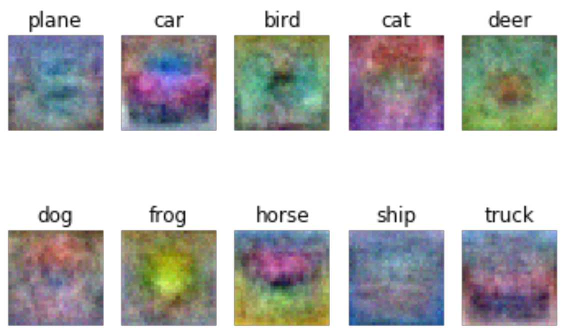
```
training accuracy: 0.385551
validation accuracy: 0.379000
```

In [22]:
```
# 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
# 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

################################################################################
# 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 i in range(128):
    # Calling SVM
    svm = LinearSVM()

    lr = np.random.uniform(learning_rates[0], learning_rates[1])

    reg = np.random.uniform(regularization_strengths[0], regularization_streng

    loss = svm.train(X_train, y_train, learning_rate=lr,reg=reg, num_iters=300

    # training the prediction
    train_pred = svm.predict(X_train)

    # training accuracy
    train_acc = np.mean(train_pred == y_train)

    # validation prediction
    val_pred   = svm.predict(X_val)

    # validation accuracy
    val_acc   = np.mean(val_pred == y_val)

    # best validation vs validation accuracy
    if best_val < val_acc:
        best_svm, best_val = svm, val_acc

    # saving result for learning rate and the regularization
    results[(lr, reg)] = (train_acc, val_acc)
################################################################################
#                              END OF YOUR CODE
################################################################################

# Print out results.
for lr, reg in sorted(results):
```

In [24]:
```
# 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: 0.353000
```

In [25]:
```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, these
# 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', 'shi
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:

- weight of the classes are resembled to class object and pictures are very fuzzy
- There are some weirdness in the picutres like horse is having two head, left and right
- Some pictures are not containing any distinguishable image of an object, but rather just the color that could be associated that class.
- Ship and plane classes, most of the pixels represented by the weight blue, representing ships and sky and plane.

In [ ]: