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. For more details see the assignments page (http://vision.stanford.edu/teaching/cs231n/assignments.html) on the course website.

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 cs231n.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-ipytho
n
%load_ext autoreload
%autoreload 2
```

CIFAR-10 Data Loading and Preprocessing

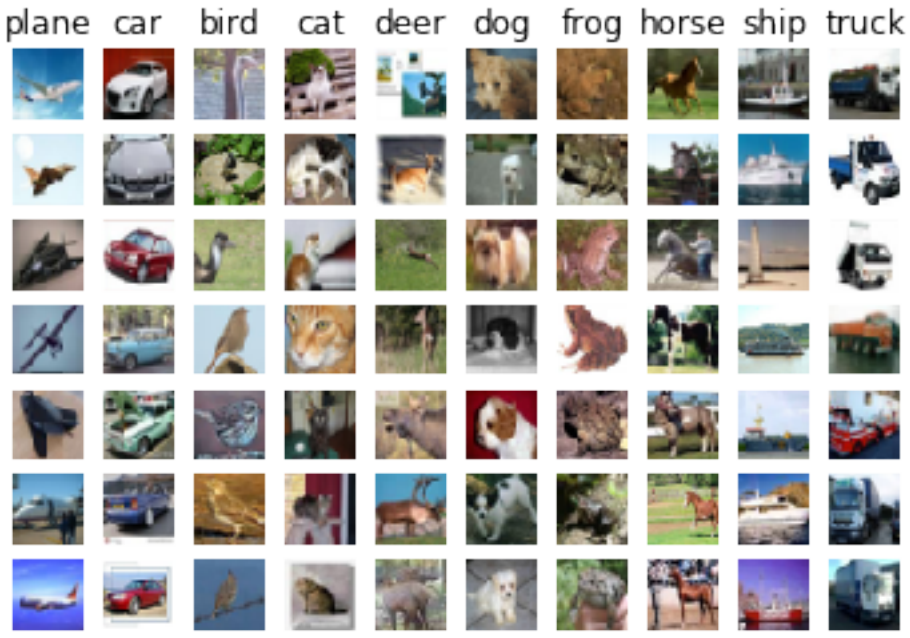
In [2]:

```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

# Cleaning up variables to prevent loading data multiple times (which may cause
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', '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 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]:

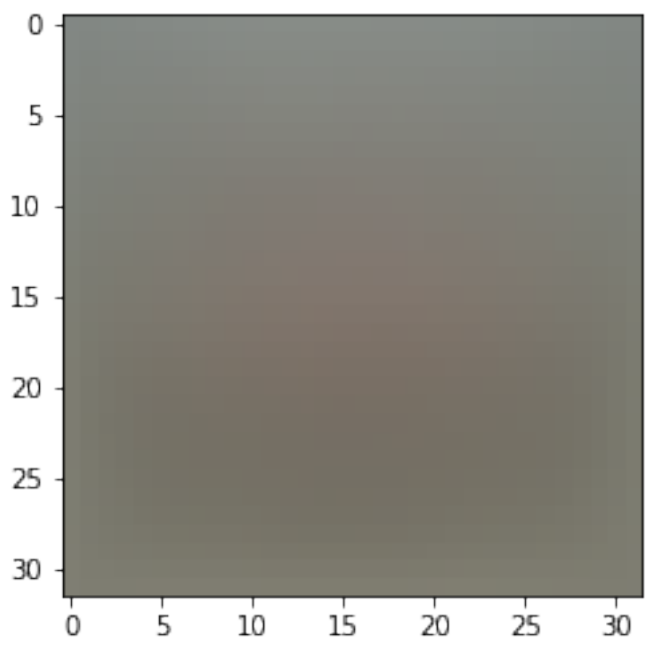
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# 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 i
mage
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 **cs231n/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 cs231n.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: 8.828588
```

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, and
# compare them with your analytically computed gradient. The numbers should matc
h
# almost exactly along all dimensions.
from cs231n.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)
print("****")

# 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: 8.951727 analytic: 8.951727, relative error: 4.156012e-11
numerical: 20.910217 analytic: 20.863606, relative error: 1.115787e-
03
numerical: 30.920956 analytic: 30.970534, relative error: 8.010515e-
04
numerical: 9.897412 analytic: 9.897412, relative error: 6.348008e-12
numerical: 17.382492 analytic: 17.382492, relative error: 2.776074e-
11
numerical: -7.964577 analytic: -8.001215, relative error: 2.294777e-
03
numerical: 10.839305 analytic: 10.839305, relative error: 2.566910e-
11
numerical: -23.446868 analytic: -23.446868, relative error: 7.532532
e-12
numerical: -11.284455 analytic: -11.284455, relative error: 1.187367
e-11
numerical: 19.946680 analytic: 19.946680, relative error: 1.960008e-
11
****
numerical: -4.493948 analytic: -4.524749, relative error: 3.415241e-
03
numerical: 7.383087 analytic: 7.383087, relative error: 2.509571e-11
numerical: 17.474259 analytic: 17.474259, relative error: 2.633592e-
12
numerical: -4.830305 analytic: -4.830305, relative error: 8.046636e-
11
numerical: -7.765563 analytic: -7.765563, relative error: 6.169870e-
11
numerical: 6.656028 analytic: 6.656028, relative error: 3.426538e-11
numerical: 24.989061 analytic: 24.952233, relative error: 7.374411e-
04
numerical: -9.231988 analytic: -9.243166, relative error: 6.050103e-
04
numerical: 21.204751 analytic: 21.224611, relative error: 4.680770e-
04
numerical: 0.307938 analytic: 0.307938, relative error: 9.428135e-10
```

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: Discrepancy can be caused by an incorrect numerical gradient due to the loss function being indifferentiable and hence taking the difference across such a boundary is invalid, such as at 0. Changing the margin would change how often / where we switch from no loss to loss; the effect on frequency is unclear (depends on how often the difference betwen class scores hits this new boundary).

In [11]:

```
# Next implement the function svm_loss_vectorized; for now only compute the loss
;
# 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 cs231n.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 fast
er.
print('difference: %f' % (loss_naive - loss_vectorized))
```

```
Naive loss: 8.828588e+00 computed in 0.085618s
Vectorized loss: 8.828588e+00 computed in 0.027589s
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 0.081041s
Vectorized loss and gradient: computed in 0.011492s
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.

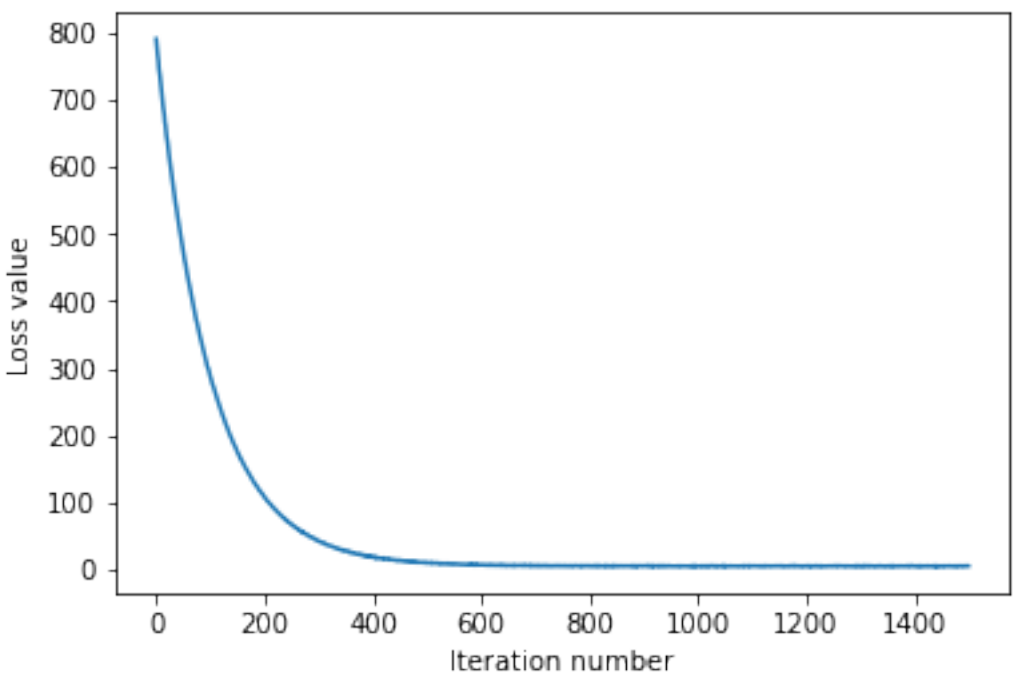
In [13]:

```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cs231n.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 789.930274
iteration 100 / 1500: loss 286.482087
iteration 200 / 1500: loss 108.001109
iteration 300 / 1500: loss 42.434364
iteration 400 / 1500: loss 18.525659
iteration 500 / 1500: loss 10.049786
iteration 600 / 1500: loss 7.151333
iteration 700 / 1500: loss 5.557743
iteration 800 / 1500: loss 5.563516
iteration 900 / 1500: loss 5.457811
iteration 1000 / 1500: loss 5.429021
iteration 1100 / 1500: loss 5.470428
iteration 1200 / 1500: loss 5.677682
iteration 1300 / 1500: loss 5.349342
iteration 1400 / 1500: loss 5.057903
That took 8.804816s
```

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 the
# 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: 0.371061
validation accuracy: 0.380000
```

In [19]:

```
# 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, 2e-7, 4e-7]
regularization_strengths = [1e4, 2e4, 3e4]

# 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 fraction
# 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 rate
```

```
.

best_params = None

################################################################################
# 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 l in learning_rates:
    for r in regularization_strengths:
        svm = LinearSVM()
        curr_loss = svm.train(X_train, y_train, learning_rate=l, reg=r,
                              num_iters=1500, verbose=True)
        y_train_pred = svm.predict(X_train)
        y_train_acc = np.mean(y_train == y_train_pred)
        y_val_pred = svm.predict(X_val)
        y_val_acc = np.mean(y_val == y_val_pred)
        results[(l, r)] = (y_train_acc, y_val_acc)
        if y_val_acc > best_val:
            best_svm = svm
            best_val = y_val_acc
            best_params = (l, r)
################################################################################
#                              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_val
)
print(best_params)
```

```
iteration 0 / 1500: loss 329.812881
iteration 100 / 1500: loss 213.884048
iteration 200 / 1500: loss 143.058440
iteration 300 / 1500: loss 96.680817
iteration 400 / 1500: loss 66.274272
iteration 500 / 1500: loss 45.652302
iteration 600 / 1500: loss 32.018373
iteration 700 / 1500: loss 22.933892
```

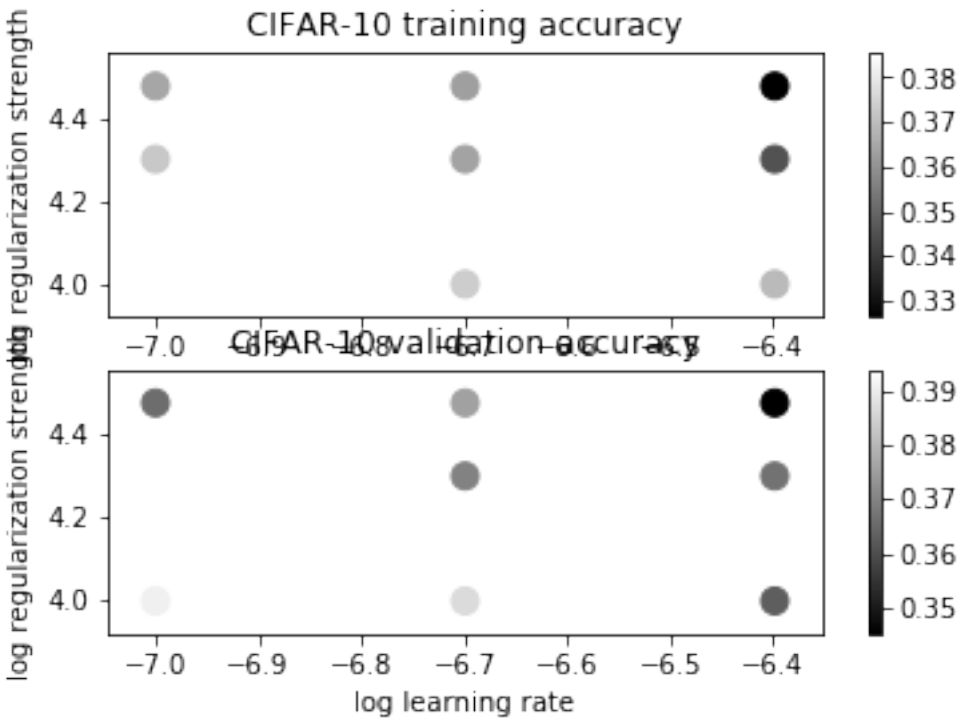
```
iteration 800 / 1500: loss 16.960706
iteration 900 / 1500: loss 13.204388
iteration 1000 / 1500: loss 9.905237
iteration 1100 / 1500: loss 8.624346
iteration 1200 / 1500: loss 7.805575
iteration 1300 / 1500: loss 6.477198
iteration 1400 / 1500: loss 5.921364
iteration 0 / 1500: loss 641.142059
iteration 100 / 1500: loss 284.166304
iteration 200 / 1500: loss 129.317181
iteration 300 / 1500: loss 60.885383
iteration 400 / 1500: loss 29.764385
iteration 500 / 1500: loss 16.033179
iteration 600 / 1500: loss 10.002331
iteration 700 / 1500: loss 7.221273
iteration 800 / 1500: loss 6.113470
iteration 900 / 1500: loss 5.898611
iteration 1000 / 1500: loss 5.647968
iteration 1100 / 1500: loss 4.748945
iteration 1200 / 1500: loss 5.250188
iteration 1300 / 1500: loss 5.457525
iteration 1400 / 1500: loss 5.639028
iteration 0 / 1500: loss 939.636591
iteration 100 / 1500: loss 281.393841
iteration 200 / 1500: loss 87.032507
iteration 300 / 1500: loss 30.018033
iteration 400 / 1500: loss 12.058377
iteration 500 / 1500: loss 7.707717
iteration 600 / 1500: loss 6.032632
iteration 700 / 1500: loss 5.942680
iteration 800 / 1500: loss 6.080024
iteration 900 / 1500: loss 5.455763
iteration 1000 / 1500: loss 5.577070
iteration 1100 / 1500: loss 5.593244
iteration 1200 / 1500: loss 5.278550
iteration 1300 / 1500: loss 5.370906
iteration 1400 / 1500: loss 5.448102
iteration 0 / 1500: loss 334.880441
iteration 100 / 1500: loss 145.749133
iteration 200 / 1500: loss 67.019249
iteration 300 / 1500: loss 31.935991
iteration 400 / 1500: loss 16.874266
iteration 500 / 1500: loss 10.518698
iteration 600 / 1500: loss 7.668947
iteration 700 / 1500: loss 6.304107
iteration 800 / 1500: loss 6.116000
iteration 900 / 1500: loss 5.828414
iteration 1000 / 1500: loss 5.485808
iteration 1100 / 1500: loss 4.864327
iteration 1200 / 1500: loss 5.101638
iteration 1300 / 1500: loss 5.231340
iteration 1400 / 1500: loss 5.233736
iteration 0 / 1500: loss 647.722236
```

```
iteration 100 / 1500: loss 129.030187
iteration 200 / 1500: loss 29.647306
iteration 300 / 1500: loss 10.381281
iteration 400 / 1500: loss 6.229066
iteration 500 / 1500: loss 5.287628
iteration 600 / 1500: loss 5.034977
iteration 700 / 1500: loss 5.086883
iteration 800 / 1500: loss 4.900851
iteration 900 / 1500: loss 5.071654
iteration 1000 / 1500: loss 4.995439
iteration 1100 / 1500: loss 5.349900
iteration 1200 / 1500: loss 4.650375
iteration 1300 / 1500: loss 5.032852
iteration 1400 / 1500: loss 6.045765
iteration 0 / 1500: loss 943.513361
iteration 100 / 1500: loss 87.700810
iteration 200 / 1500: loss 12.691915
iteration 300 / 1500: loss 6.496296
iteration 400 / 1500: loss 5.773099
iteration 500 / 1500: loss 5.794557
iteration 600 / 1500: loss 6.278239
iteration 700 / 1500: loss 5.329029
iteration 800 / 1500: loss 5.808075
iteration 900 / 1500: loss 5.530021
iteration 1000 / 1500: loss 5.997961
iteration 1100 / 1500: loss 5.569494
iteration 1200 / 1500: loss 5.711300
iteration 1300 / 1500: loss 5.374030
iteration 1400 / 1500: loss 5.113521
iteration 0 / 1500: loss 333.361183
iteration 100 / 1500: loss 67.944055
iteration 200 / 1500: loss 17.117683
iteration 300 / 1500: loss 7.442095
iteration 400 / 1500: loss 5.785281
iteration 500 / 1500: loss 5.392824
iteration 600 / 1500: loss 5.244573
iteration 700 / 1500: loss 4.841607
iteration 800 / 1500: loss 5.774334
iteration 900 / 1500: loss 5.524201
iteration 1000 / 1500: loss 5.364770
iteration 1100 / 1500: loss 4.612733
iteration 1200 / 1500: loss 5.402273
iteration 1300 / 1500: loss 5.708605
iteration 1400 / 1500: loss 4.692609
iteration 0 / 1500: loss 637.681800
iteration 100 / 1500: loss 28.715797
iteration 200 / 1500: loss 6.398879
iteration 300 / 1500: loss 5.323034
iteration 400 / 1500: loss 5.697103
iteration 500 / 1500: loss 5.497378
iteration 600 / 1500: loss 4.910578
iteration 700 / 1500: loss 5.173264
iteration 800 / 1500: loss 6.031431
```

```
iteration 900 / 1500: loss 5.393203
iteration 1000 / 1500: loss 5.647651
iteration 1100 / 1500: loss 5.567541
iteration 1200 / 1500: loss 5.107683
iteration 1300 / 1500: loss 5.211513
iteration 1400 / 1500: loss 5.565286
iteration 0 / 1500: loss 951.727452
iteration 100 / 1500: loss 12.247992
iteration 200 / 1500: loss 5.634766
iteration 300 / 1500: loss 6.143236
iteration 400 / 1500: loss 5.552660
iteration 500 / 1500: loss 5.745416
iteration 600 / 1500: loss 5.421951
iteration 700 / 1500: loss 5.998420
iteration 800 / 1500: loss 5.485592
iteration 900 / 1500: loss 5.410751
iteration 1000 / 1500: loss 5.983546
iteration 1100 / 1500: loss 5.506991
iteration 1200 / 1500: loss 5.668893
iteration 1300 / 1500: loss 5.570115
iteration 1400 / 1500: loss 5.794653
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.385469 val accura
cy: 0.391000
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.372694 val accura
cy: 0.394000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.364878 val accura
cy: 0.366000
lr 2.000000e-07 reg 1.000000e+04 train accuracy: 0.373816 val accura
cy: 0.387000
lr 2.000000e-07 reg 2.000000e+04 train accuracy: 0.364000 val accura
cy: 0.370000
lr 2.000000e-07 reg 3.000000e+04 train accuracy: 0.363102 val accura
cy: 0.376000
lr 4.000000e-07 reg 1.000000e+04 train accuracy: 0.369327 val accura
cy: 0.363000
lr 4.000000e-07 reg 2.000000e+04 train accuracy: 0.345408 val accura
cy: 0.367000
lr 4.000000e-07 reg 3.000000e+04 train accuracy: 0.326265 val accura
cy: 0.345000
best validation accuracy achieved during cross-validation: 0.394000
(1e-07, 20000.0)
```

In [20]:

```
# 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 [21]:

```
# 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: 0.372000
```

In [22]:

```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, these m
ay
# 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: The visualized SVM weights look like blurry versions of each of their respective classes. They represent the average pixel values of each class (that maximize dot product with examples of this class).