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 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

# 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 module
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in
%load_ext autoreload
%autoreload 2
```

CIFAR-10 Data Loading and Preprocessing

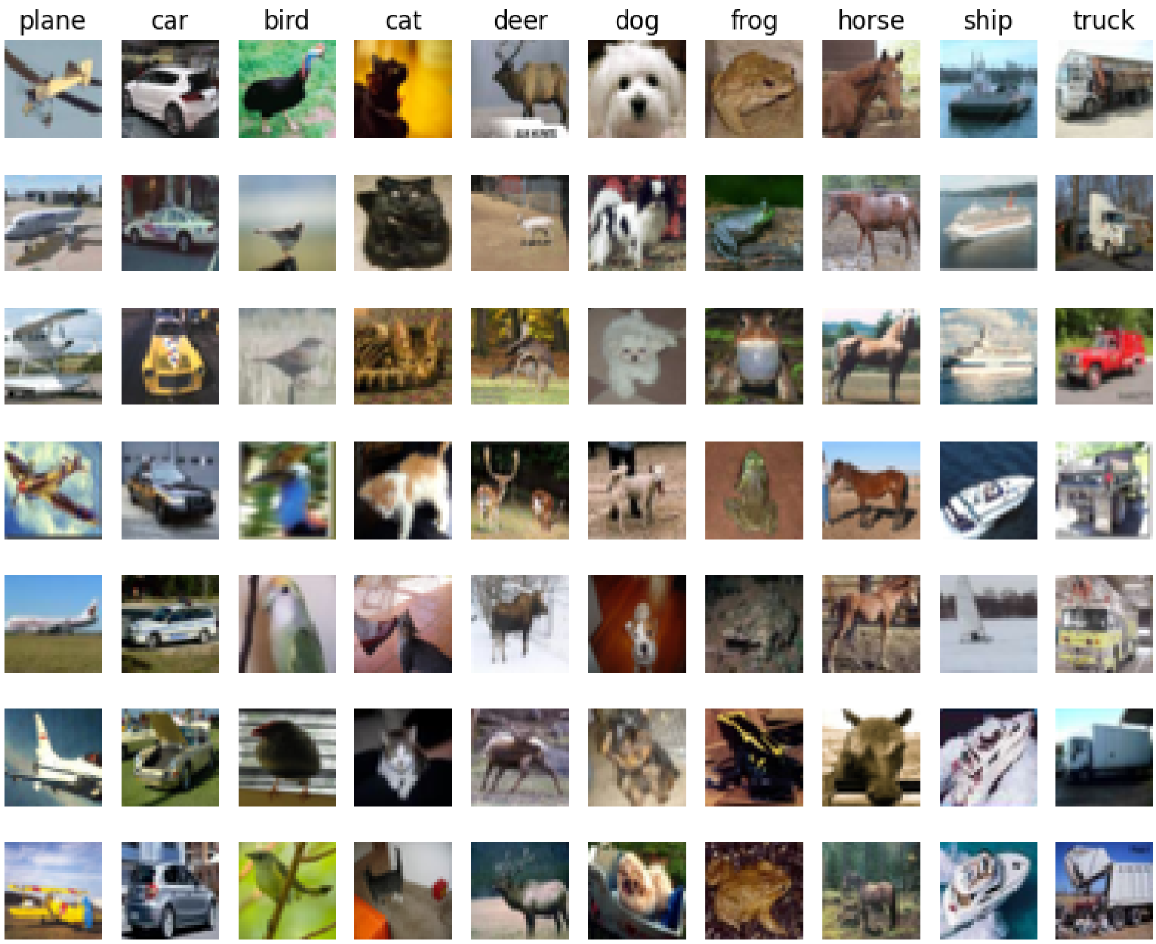
In [2]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

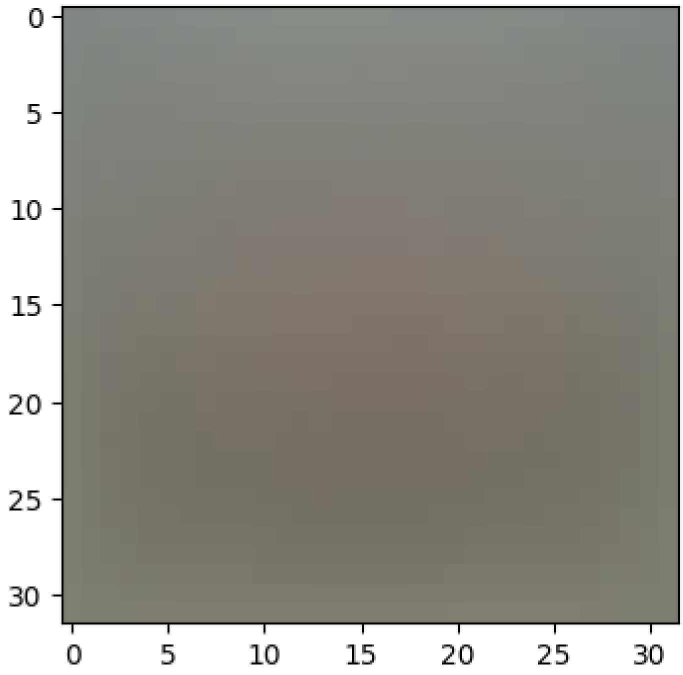
In [5]:
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# 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
plt.show()

# 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

# third: append the bias dimension of ones (i.e. bias trick) so that our
# 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)
```

```
[130.64189796 135.98173469 132.47391837 130.05569388 135.34804082
 131.75402041 130.96055102 136.14328571 132.47636735 131.48467347]
```




```
(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 `svm_loss_naive` which uses for loops to evaluate the multiclass SVM loss function.

In [7]:
```
# 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: 9.355109
```

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 [8]:
```
# 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 dimensio
# compare them with your analytically computed gradient. The numbers shou
# 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)

# 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: 1.103146 analytic: 1.144478, relative error: 1.838896e-02
numerical: -35.257782 analytic: -35.174089, relative error: 1.188285e-03
numerical: -19.827238 analytic: -19.827238, relative error: 1.772689e-11
numerical: 18.439843 analytic: 18.439843, relative error: 2.390781e-12
numerical: -24.397275 analytic: -24.397275, relative error: 1.459207e-11
numerical: -2.711725 analytic: -2.711725, relative error: 2.526710e-11
numerical: 25.213882 analytic: 25.213882, relative error: 1.009655e-11
numerical: 47.308391 analytic: 47.308391, relative error: 1.100846e-12
numerical: -4.137391 analytic: -4.137391, relative error: 7.656890e-12
numerical: 5.813249 analytic: 5.813249, relative error: 2.712296e-11
numerical: 29.799250 analytic: 29.804406, relative error: 8.650441e-05
numerical: 24.039649 analytic: 24.016772, relative error: 4.760310e-04
numerical: 6.973807 analytic: 6.979935, relative error: 4.391397e-04
numerical: 34.835816 analytic: 34.831412, relative error: 6.321604e-05
numerical: -41.182279 analytic: -41.185053, relative error: 3.368376e-05
numerical: 3.800407 analytic: 3.797105, relative error: 4.346265e-04
numerical: -15.944429 analytic: -15.947869, relative error: 1.078634e-04
numerical: -39.559792 analytic: -39.561093, relative error: 1.643625e-05
numerical: -49.100083 analytic: -49.101241, relative error: 1.179376e-05
numerical: -12.975923 analytic: -12.983406, relative error: 2.882372e-04
```

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*

$YourAnswer:$

- The SVM loss function, particularly with the hinge loss component, is not differentiable at every point. The hinge loss $\max(0, 1 - y_i(w^T x_i + b))$ is not differentiable where $y_i(w^T x_i + b) = 1$. At these points, the gradient is not defined, and any method to compute it might yield different results.
- It's not reason for concern. For those point that $y_i(w^T x_i + b)! = 0$, their grad and loss are correct.
- When there are two weights that are equal, the classifier is unable to determine which category has the greater weight, which then leads to a checking fail
- A larger margin increases the edge's tolerance for point classification errors, which can lead to a decrease in the frequency of such occurrences

In [9]:
```
# Next implement the function svm_loss_vectorized; for now only compute t
# 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 mu
print('difference: %f' % (loss_naive - loss_vectorized))
```

```
Naive loss: 9.355109e+00 computed in 0.241225s
Vectorized loss: 9.355109e+00 computed in 0.002266s
difference: 0.000000
```

In [10]:
```
# Complete the implementation of svm_loss_vectorized, and compute the gra
# of the loss function in a vectorized way.

# The naive implementation and the vectorized implementation should match
# 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 comput
# by the two implementations. The gradient on the other hand is a matrix,
# 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 0.175261s
Vectorized loss and gradient: computed in 0.001737s
difference: 0.000000
```

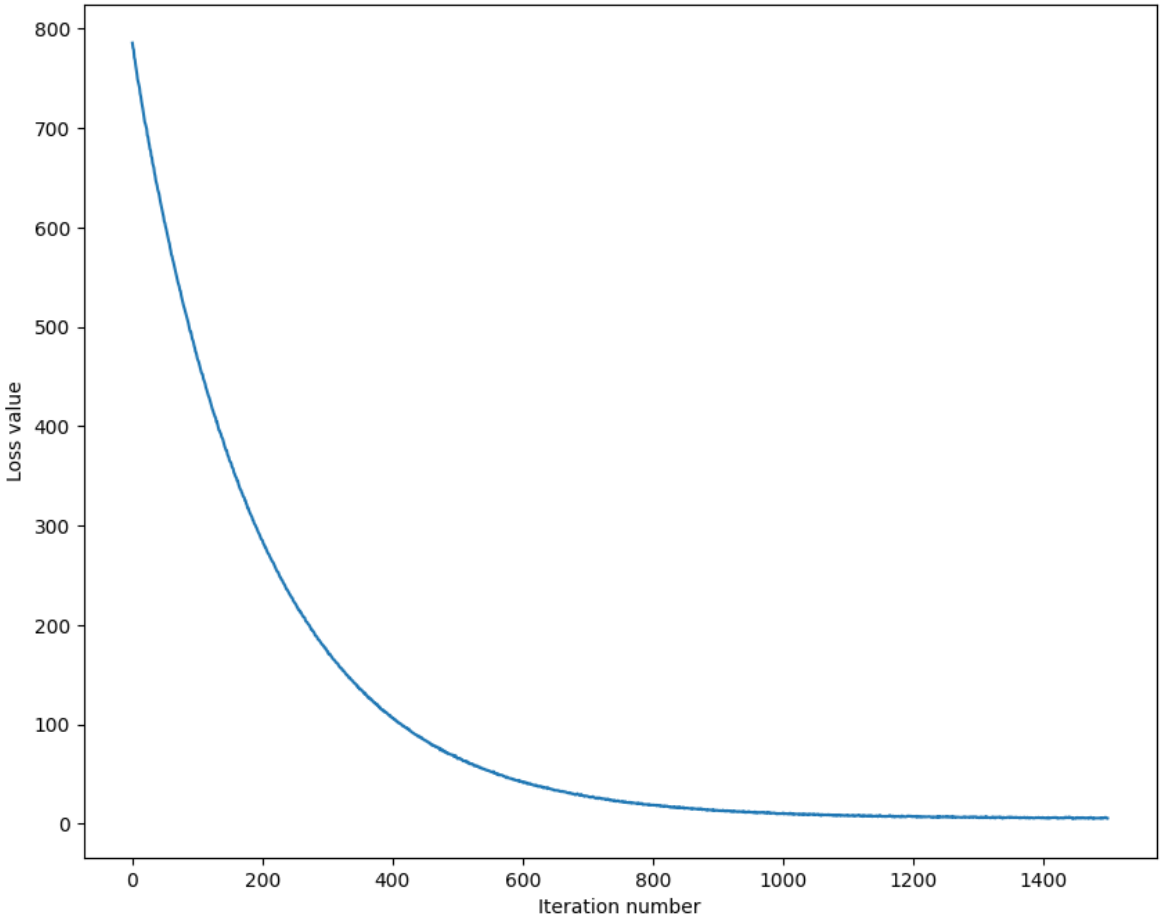
Stochastic Gradient Descent

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. Your code for this part will be written inside `cs231n/classifiers/linear_classifier.py`.

In [11]:
```
# 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 785.480511
iteration 100 / 1500: loss 467.569758
iteration 200 / 1500: loss 285.091416
iteration 300 / 1500: loss 172.797686
iteration 400 / 1500: loss 106.334010
iteration 500 / 1500: loss 65.433763
iteration 600 / 1500: loss 41.659203
iteration 700 / 1500: loss 27.642896
iteration 800 / 1500: loss 19.012065
iteration 900 / 1500: loss 13.559665
iteration 1000 / 1500: loss 10.889506
iteration 1100 / 1500: loss 8.296221
iteration 1200 / 1500: loss 7.508330
iteration 1300 / 1500: loss 6.552024
iteration 1400 / 1500: loss 5.774173
That took 2.521627s
```

In [12]:
```
# 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 [13]:
```
# Write the LinearSVM.predict function and evaluate the performance on bo
# 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: 0.378184
validation accuracy: 0.378000
```

In [14]:
```
# Use the validation set to tune hyperparameters (regularization strength
# learning rate). You should experiment with different ranges for the lea
# rates and regularization strengths; if you are careful you should be ab
# get a classification accuracy of about 0.39 on the validation set.

# Note: you may see runtime/overflow warnings during hyper-parameter sear
# This may be caused by extreme values, and is not a bug.

# 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 fr
# 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 validat

##########################################################################
# TODO:
# Write code that chooses the best hyperparameters by tuning on the valid
# set. For each combination of hyperparameters, train a linear SVM on the
# training set, compute its accuracy on the training and validation sets,
# store these numbers in the results dictionary. In addition, store the b
```

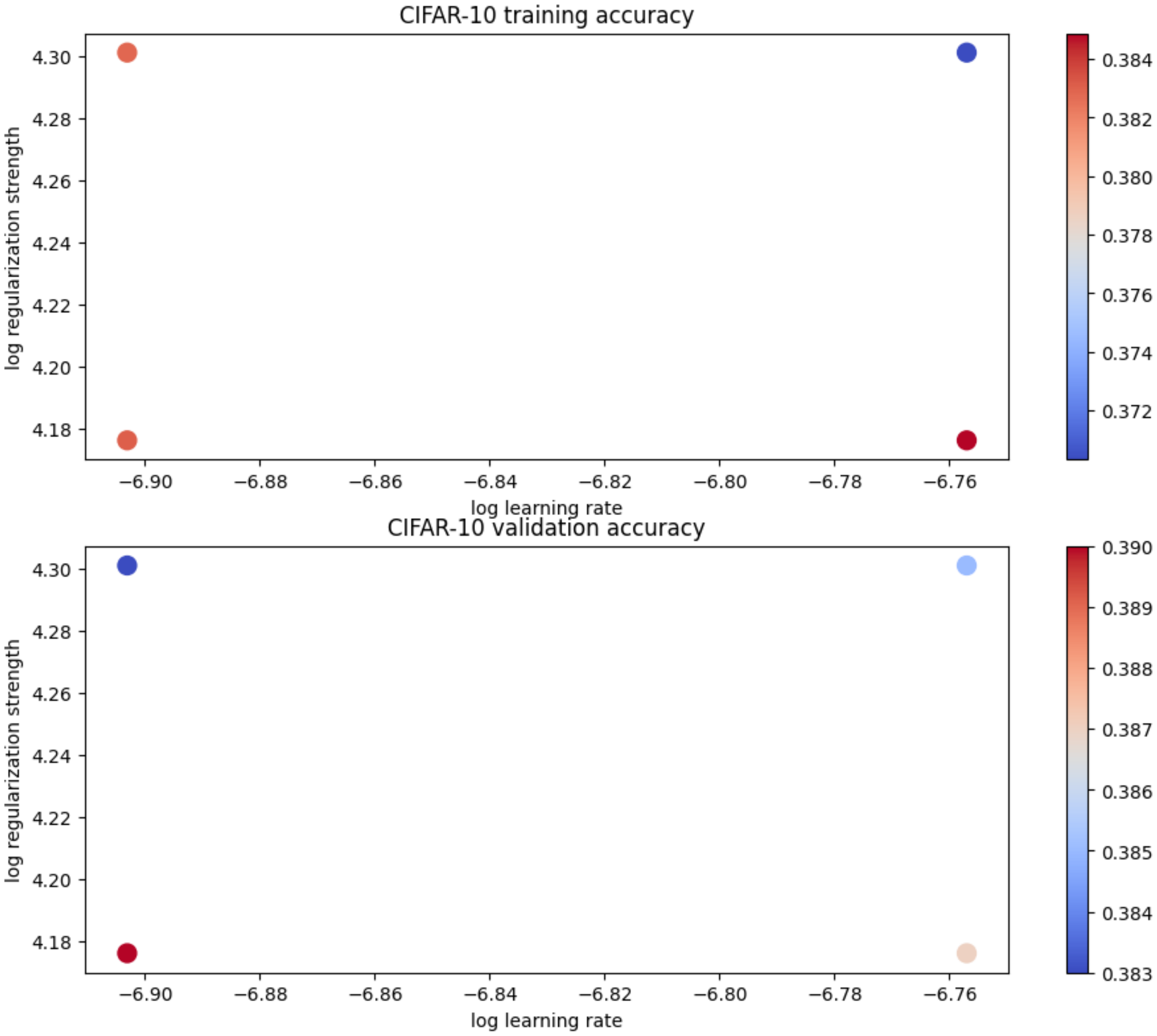
```
# validation accuracy in best_val and the LinearSVM object that achieves
# accuracy in best_svm.
#
# Hint: You should use a small value for num_iters as you develop your
# validation code so that the SVMs don't take much time to train; once yo
# confident that your validation code works, you should rerun the validat
# code with a larger value for num_iters.
########################################################################

# Provided as a reference. You may or may not want to change these hyperp
learning_rates = [1.25e-7, 1.75e-7]
regularization_strengths = [1.5e4, 2e4]

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

for lr in learning_rates:
    for regs in regularization_strengths:
        svm = LinearSVM()
        loss_hist = svm.train(X_train, y_train, lr, regs, num_iters=1500)
        y_train_pred = svm.predict(X_train)
        train_accurary = np.mean(y_train_pred == y_train)
        y_val_pred = svm.predict(X_val)
        val_accurary = np.mean(y_val_pred == y_val)
        if val_accurary > best_val:
            best_val = val_accurary
            best_svm = svm
        results[(lr, regs)] = train_accurary, val_accurary

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# 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' % b
```

```
lr 1.250000e-07 reg 1.500000e+04 train accuracy: 0.383102 val accuracy: 0.
390000
lr 1.250000e-07 reg 2.000000e+04 train accuracy: 0.382878 val accuracy: 0.
383000
lr 1.750000e-07 reg 1.500000e+04 train accuracy: 0.384878 val accuracy: 0.
387000
lr 1.750000e-07 reg 2.000000e+04 train accuracy: 0.370306 val accuracy: 0.
385000
best validation accuracy achieved during cross-validation: 0.390000
```

In [15]:

```
# Visualize the cross-validation results
import math
import pdb

# pdb.set_trace()

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.tight_layout(pad=3)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap=plt.cm.cool
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 2
plt.subplot(2, 1, 2)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap=plt.cm.cool
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.show()
```

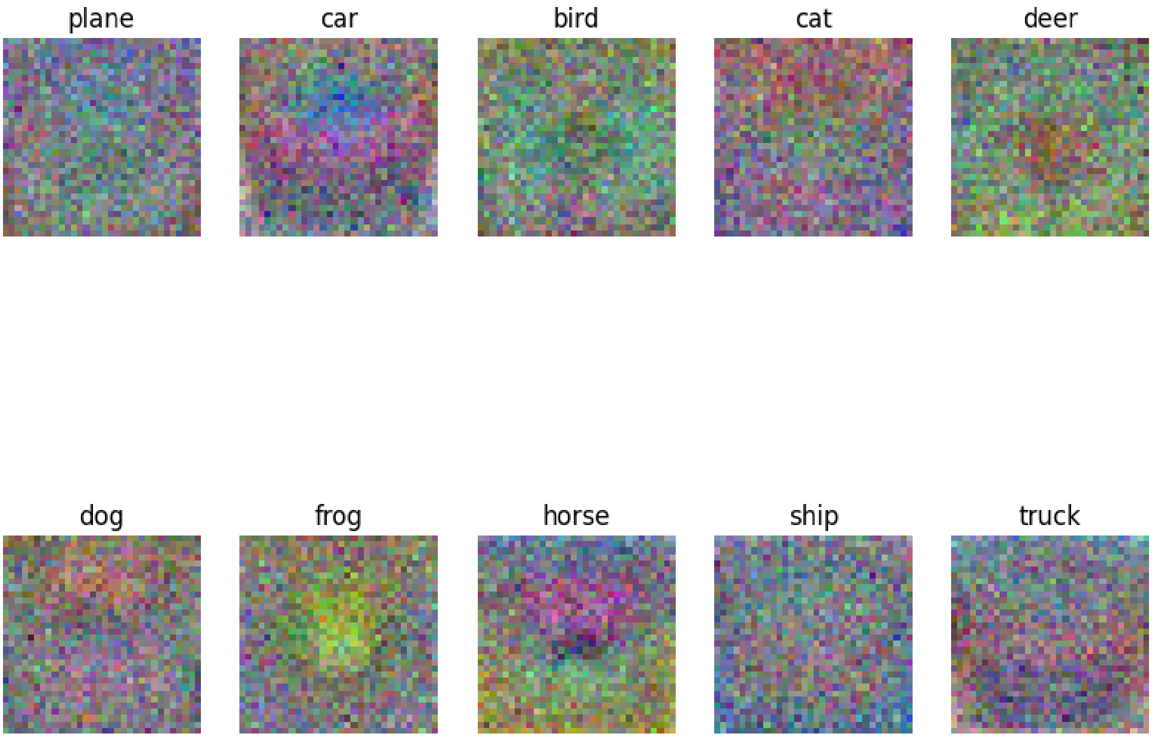


In [16]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accur
```

```
linear SVM on raw pixels final test set accuracy: 0.385000
```

In [17]:
```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength,
# 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',
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.

$\color{blue}{\textit{Your Answer :}}$ *fill this in*

1. discribe The images look blurry, only slightly visible with the outlines of the corresponding objects, and the background color is too muddy
2. Why Each image of weights represents a kind of "template" that the SVM uses to classify new images. The "template" is formed by the aggregate of all the training examples it has seen for each category. If the template matches closely with the features of a new image, the SVM will likely classify that new image as belonging to the corresponding category.