In [1]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'cs231n/assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/assignments/assignment1
```

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 [2]:
```
# 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'
```

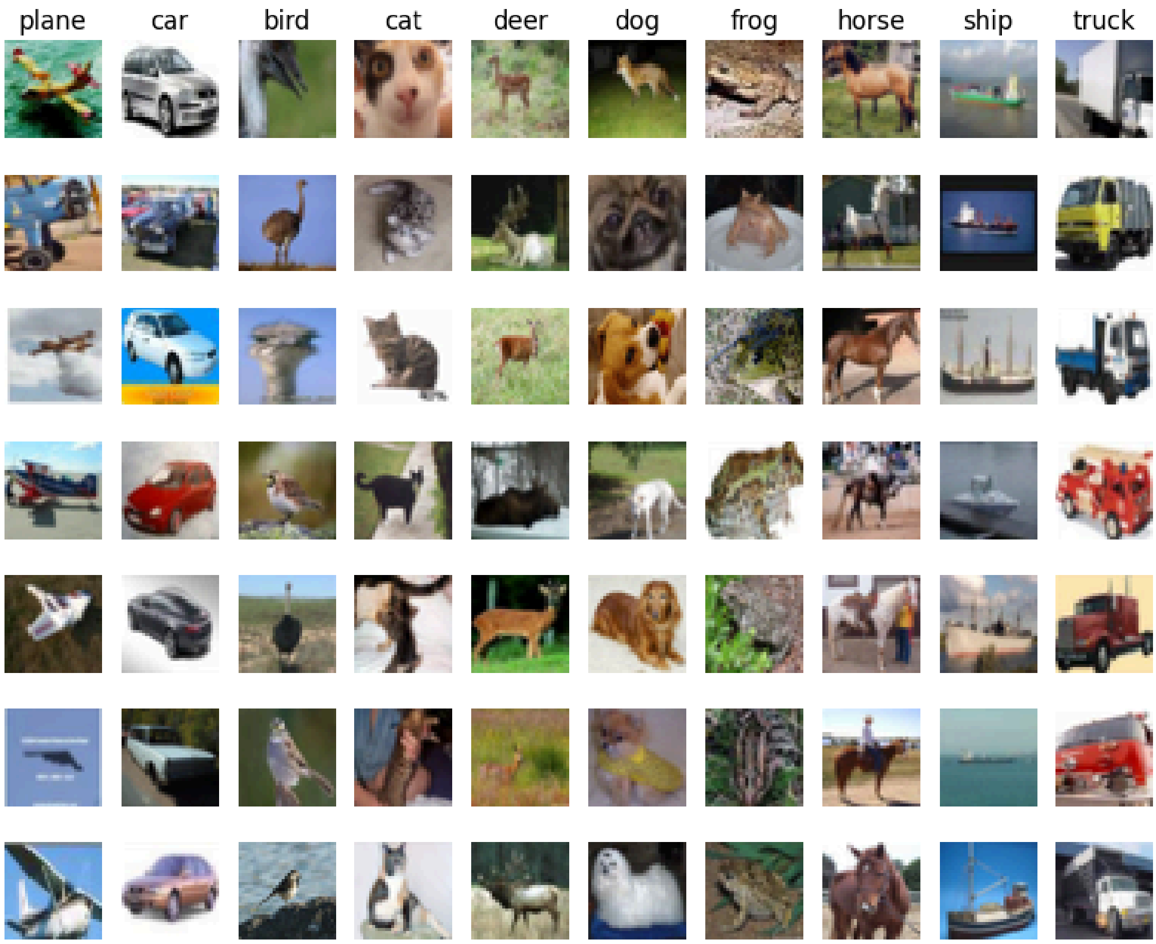
```
# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipytho
%load_ext autoreload
%autoreload 2
```

CIFAR-10 Data Loading and Preprocessing

In [3]:
```
# 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
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 [4]:
```
# 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'
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 [5]:

```
# 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]
```

```python
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 [6]:

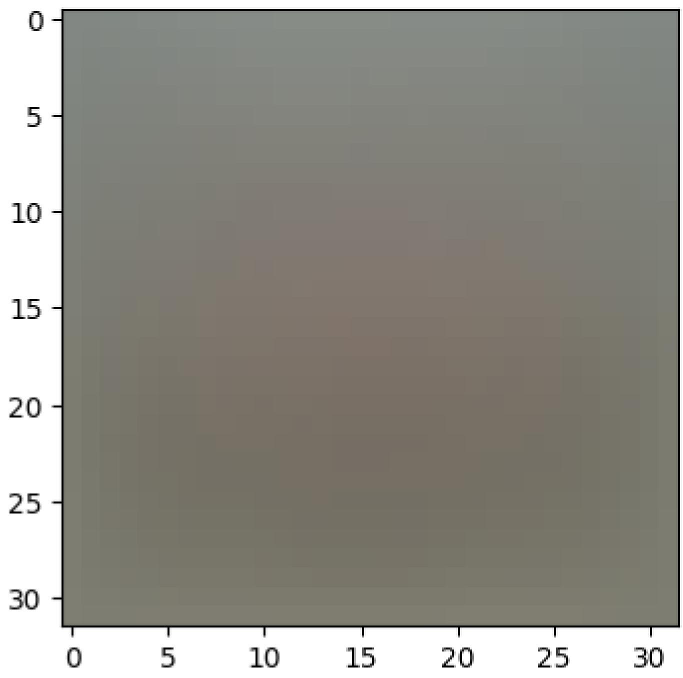
```python
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# 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 [7]:

```python
# 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
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 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)
```

```
[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 [8]:
```
# 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: 9.304046
```

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, 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 [9]:
```
# 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
# 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: 37.969526 analytic: 37.969526, relative error: 9.039369e-12
numerical: 10.142597 analytic: 10.142597, relative error: 1.119379e-11
numerical: 17.599408 analytic: 17.599408, relative error: 8.711204e-12
numerical: 9.187081 analytic: 9.187081, relative error: 5.942052e-12
numerical: -0.789392 analytic: -0.789392, relative error: 2.694702e-11
numerical: -0.534573 analytic: -0.534573, relative error: 3.395370e-10
numerical: 36.749991 analytic: 36.749991, relative error: 7.546256e-12
numerical: 10.971723 analytic: 10.971723, relative error: 4.572032e-11
numerical: -2.919024 analytic: -2.919024, relative error: 7.296765e-11
numerical: 26.353395 analytic: 26.353395, relative error: 1.280868e-11
numerical: 14.093647 analytic: 14.093647, relative error: 5.501651e-11
numerical: -39.193654 analytic: -39.193654, relative error: 2.862482e-12
numerical: -6.177805 analytic: -6.177805, relative error: 7.967469e-11
numerical: 5.189721 analytic: 5.189721, relative error: 9.241608e-11
numerical: 9.100506 analytic: 9.100506, relative error: 5.958370e-12
numerical: 9.807377 analytic: 9.807377, relative error: 9.274302e-12
numerical: -6.275799 analytic: -6.275799, relative error: 1.632384e-11
numerical: 23.938894 analytic: 23.938894, relative error: 1.014462e-11
numerical: -11.113542 analytic: -11.113542, relative error: 3.292885e-12
numerical: -6.654904 analytic: -6.654904, relative error: 5.728994e-11
```

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*

$\color{blue}{Your Answer:}$ The discrepancy is caused by the non-derivibility of the SVM loss function at some points.

When performing gradcheck, [L(W+h) - L(W)]/h is calculated, if the score is close to the hinge threshold, some part of loss function may reduce to 0 because of the max(0, margin) function.

I think that is not a big problem. Since in common cases, the majority part of the loss function will not lies close to the hinge threshold, the discrepancy will not become too big.

For example, let f(x) = max(0,x), for x = 0, f(0) = 0, however [f(h)-f(0)]/h = 1, the discrepancy occurs.

Changing the margin does not significantly increase or decrease the frequency of the check fail, since the data distribution is not clear. But if the data are centered near the hinge threshold, it may have some effect.

In [10]:

```
# 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
print('difference: %f' % (loss_naive - loss_vectorized))
```

```
Naive loss: 9.304046e+00 computed in 0.068018s
Vectorized loss: 9.304046e+00 computed in 0.012875s
difference: -0.000000
```

In [11]:

```
# 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 0.083003s
Vectorized loss and gradient: computed in 0.008371s
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`.

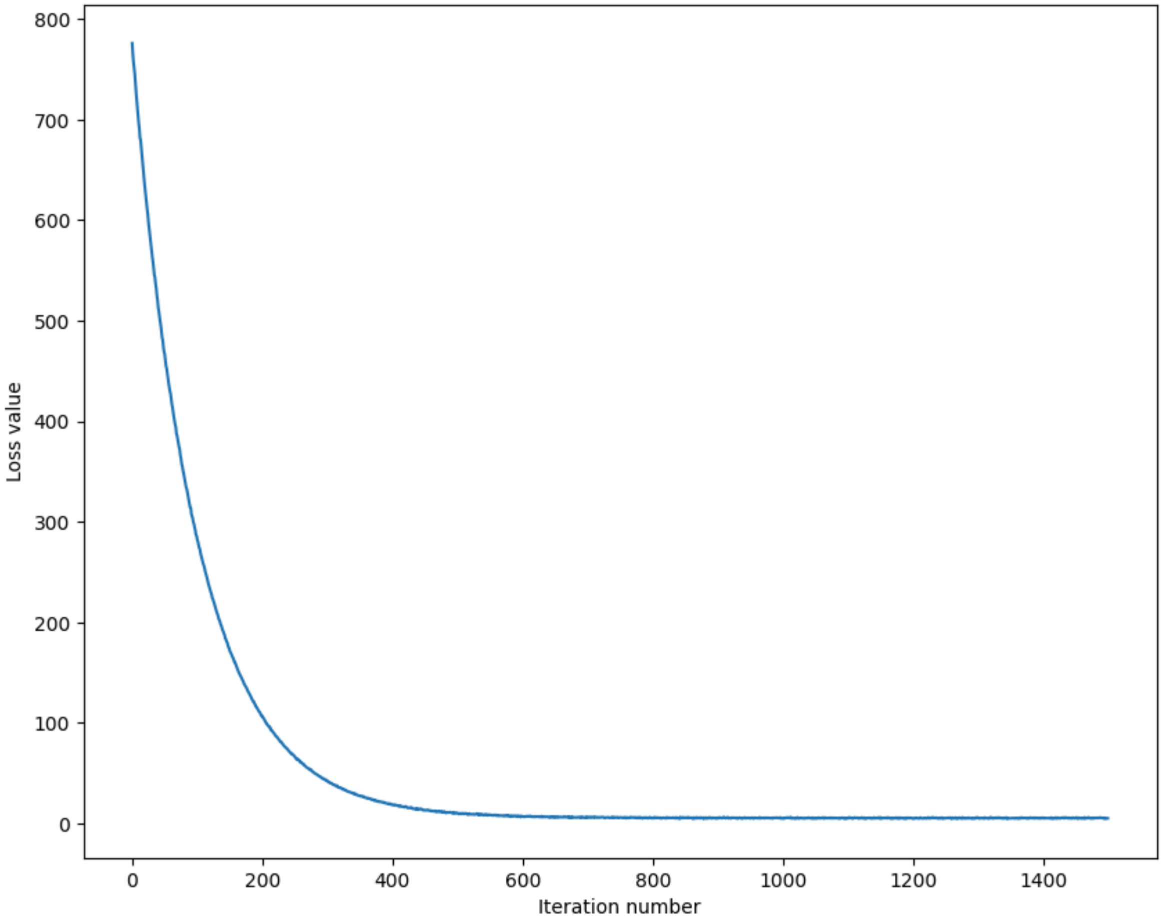
In [13]:

```
# 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 775.799241
iteration 100 / 1500: loss 282.282357
iteration 200 / 1500: loss 106.822358
iteration 300 / 1500: loss 41.844484
iteration 400 / 1500: loss 18.641197
iteration 500 / 1500: loss 10.272672
iteration 600 / 1500: loss 6.987436
iteration 700 / 1500: loss 5.979592
iteration 800 / 1500: loss 5.463722
iteration 900 / 1500: loss 5.264637
iteration 1000 / 1500: loss 5.171628
iteration 1100 / 1500: loss 5.562370
iteration 1200 / 1500: loss 4.691731
iteration 1300 / 1500: loss 5.183524
iteration 1400 / 1500: loss 5.634585
That took 11.779835s
```

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 [16]:

```
# 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: 0.370408
validation accuracy: 0.372000
```

In [24]:

```
# 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 0.39 (> 0.385) on the validation set.

# Note: you may see runtime/overflow warnings during hyper-parameter search.
# 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 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

################################################################################
# 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   #
```

```python
# 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.                                      #
################################################################################

# Provided as a reference. You may or may not want to change these hyperparamete
learning_rates = [1e-7, 5e-5]
regularization_strengths = [2.5e4, 5e4]

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

learning_rates = np.linspace(1e-7, 5e-7, 10)
regularization_strengths = np.linspace(2.5e4, 3.0e4, 10)

for lr in learning_rates:
  for reg in regularization_strengths:
    hyperparameters_tuple = (lr, reg)

    svm = LinearSVM()
    svm.train(X_train, y_train, lr, reg, num_iters=1500)
    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)
    accuracy_tuple = (train_accuracy, val_accuracy)

    results[hyperparameters_tuple] = accuracy_tuple
    if val_accuracy > best_val:
      best_val = val_accuracy
      best_svm = svm

# *****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' % best_val
```

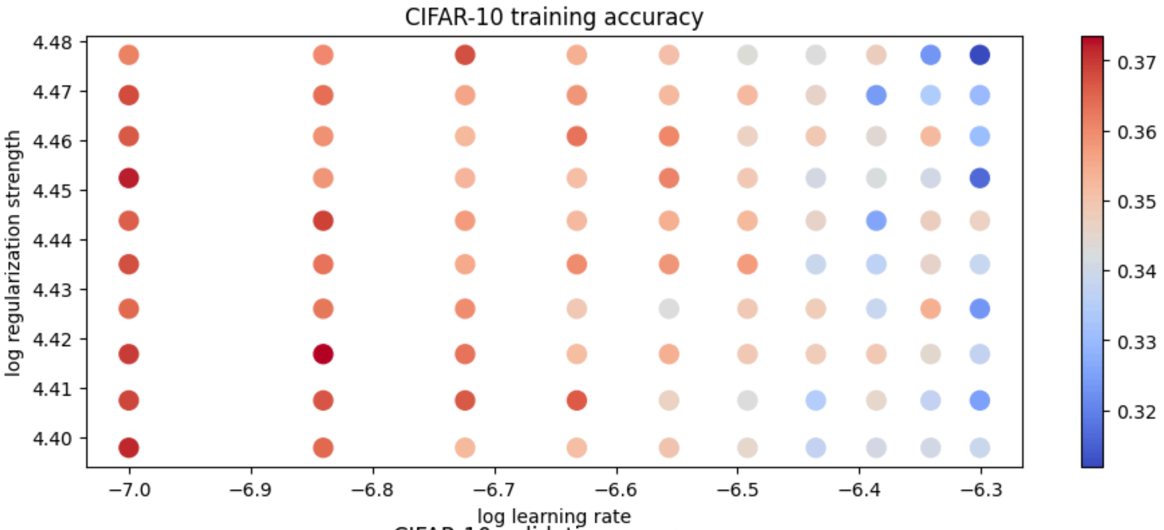
```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.371306 val accuracy: 0.378000
lr 1.000000e-07 reg 2.555556e+04 train accuracy: 0.368388 val accuracy: 0.380000
lr 1.000000e-07 reg 2.611111e+04 train accuracy: 0.369367 val accuracy: 0.385000
lr 1.000000e-07 reg 2.666667e+04 train accuracy: 0.364653 val accuracy: 0.391000
lr 1.000000e-07 reg 2.722222e+04 train accuracy: 0.367694 val accuracy: 0.369000
lr 1.000000e-07 reg 2.777778e+04 train accuracy: 0.365449 val accuracy: 0.378000
lr 1.000000e-07 reg 2.833333e+04 train accuracy: 0.372061 val accuracy: 0.383000
lr 1.000000e-07 reg 2.888889e+04 train accuracy: 0.366531 val accuracy: 0.362000
lr 1.000000e-07 reg 2.944444e+04 train accuracy: 0.368163 val accuracy: 0.377000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.361082 val accuracy: 0.377000
lr 1.444444e-07 reg 2.500000e+04 train accuracy: 0.364837 val accuracy: 0.383000
lr 1.444444e-07 reg 2.555556e+04 train accuracy: 0.366653 val accuracy: 0.369000
lr 1.444444e-07 reg 2.611111e+04 train accuracy: 0.373592 val accuracy: 0.378000
lr 1.444444e-07 reg 2.666667e+04 train accuracy: 0.362776 val accuracy: 0.375000
lr 1.444444e-07 reg 2.722222e+04 train accuracy: 0.363000 val accuracy: 0.380000
lr 1.444444e-07 reg 2.777778e+04 train accuracy: 0.369102 val accuracy: 0.365000
lr 1.444444e-07 reg 2.833333e+04 train accuracy: 0.358531 val accuracy: 0.371000
lr 1.444444e-07 reg 2.888889e+04 train accuracy: 0.359224 val accuracy: 0.375000
lr 1.444444e-07 reg 2.944444e+04 train accuracy: 0.363490 val accuracy: 0.368000
lr 1.444444e-07 reg 3.000000e+04 train accuracy: 0.360224 val accuracy: 0.386000
lr 1.888889e-07 reg 2.500000e+04 train accuracy: 0.352816 val accuracy: 0.375000
lr 1.888889e-07 reg 2.555556e+04 train accuracy: 0.366490 val accuracy: 0.367000
lr 1.888889e-07 reg 2.611111e+04 train accuracy: 0.363245 val accuracy: 0.381000
lr 1.888889e-07 reg 2.666667e+04 train accuracy: 0.359694 val accuracy: 0.354000
lr 1.888889e-07 reg 2.722222e+04 train accuracy: 0.355082 val accuracy: 0.369000
lr 1.888889e-07 reg 2.777778e+04 train accuracy: 0.357551 val accuracy: 0.356000
lr 1.888889e-07 reg 2.833333e+04 train accuracy: 0.353041 val accuracy: 0.366000
lr 1.888889e-07 reg 2.888889e+04 train accuracy: 0.352755 val accuracy: 0.365000
lr 1.888889e-07 reg 2.944444e+04 train accuracy: 0.355918 val accuracy: 0.357000
lr 1.888889e-07 reg 3.000000e+04 train accuracy: 0.367388 val accuracy: 0.362000
lr 2.333333e-07 reg 2.500000e+04 train accuracy: 0.351204 val accuracy: 0.355000
lr 2.333333e-07 reg 2.555556e+04 train accuracy: 0.366286 val accuracy: 0.367000
lr 2.333333e-07 reg 2.611111e+04 train accuracy: 0.351673 val accuracy: 0.362000
lr 2.333333e-07 reg 2.666667e+04 train accuracy: 0.349000 val accuracy: 0.347000
lr 2.333333e-07 reg 2.722222e+04 train accuracy: 0.359653 val accuracy: 0.360000
lr 2.333333e-07 reg 2.777778e+04 train accuracy: 0.352184 val accuracy: 0.357000
lr 2.333333e-07 reg 2.833333e+04 train accuracy: 0.351122 val accuracy: 0.371000
lr 2.333333e-07 reg 2.888889e+04 train accuracy: 0.363122 val accuracy: 0.374000
lr 2.333333e-07 reg 2.944444e+04 train accuracy: 0.358367 val accuracy: 0.365000
lr 2.333333e-07 reg 3.000000e+04 train accuracy: 0.354286 val accuracy: 0.356000
lr 2.777778e-07 reg 2.500000e+04 train accuracy: 0.349857 val accuracy: 0.363000
lr 2.777778e-07 reg 2.555556e+04 train accuracy: 0.346918 val accuracy: 0.357000
lr 2.777778e-07 reg 2.611111e+04 train accuracy: 0.354245 val accuracy: 0.393000
lr 2.777778e-07 reg 2.666667e+04 train accuracy: 0.342551 val accuracy: 0.352000
lr 2.777778e-07 reg 2.722222e+04 train accuracy: 0.358163 val accuracy: 0.370000
lr 2.777778e-07 reg 2.777778e+04 train accuracy: 0.354531 val accuracy: 0.361000
lr 2.777778e-07 reg 2.833333e+04 train accuracy: 0.360878 val accuracy: 0.378000
lr 2.777778e-07 reg 2.888889e+04 train accuracy: 0.360429 val accuracy: 0.372000
lr 2.777778e-07 reg 2.944444e+04 train accuracy: 0.352306 val accuracy: 0.371000
lr 2.777778e-07 reg 3.000000e+04 train accuracy: 0.350694 val accuracy: 0.365000
lr 3.222222e-07 reg 2.500000e+04 train accuracy: 0.345082 val accuracy: 0.337000
lr 3.222222e-07 reg 2.555556e+04 train accuracy: 0.342571 val accuracy: 0.365000
lr 3.222222e-07 reg 2.611111e+04 train accuracy: 0.348755 val accuracy: 0.350000
lr 3.222222e-07 reg 2.666667e+04 train accuracy: 0.348878 val accuracy: 0.350000
lr 3.222222e-07 reg 2.722222e+04 train accuracy: 0.357857 val accuracy: 0.356000
lr 3.222222e-07 reg 2.777778e+04 train accuracy: 0.352571 val accuracy: 0.373000
lr 3.222222e-07 reg 2.833333e+04 train accuracy: 0.349020 val accuracy: 0.367000
lr 3.222222e-07 reg 2.888889e+04 train accuracy: 0.346551 val accuracy: 0.375000
lr 3.222222e-07 reg 2.944444e+04 train accuracy: 0.352245 val accuracy: 0.366000
lr 3.222222e-07 reg 3.000000e+04 train accuracy: 0.342980 val accuracy: 0.346000
```

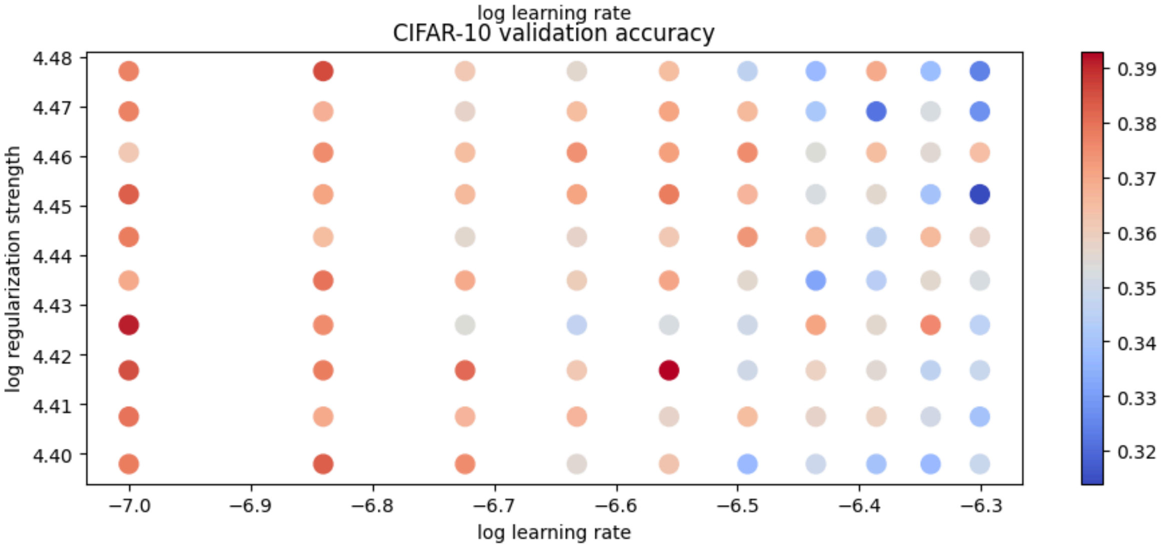
```
lr 3.666667e-07 reg 2.500000e+04 train accuracy: 0.337510 val accuracy: 0.349000
lr 3.666667e-07 reg 2.555556e+04 train accuracy: 0.334408 val accuracy: 0.358000
lr 3.666667e-07 reg 2.611111e+04 train accuracy: 0.348286 val accuracy: 0.359000
lr 3.666667e-07 reg 2.666667e+04 train accuracy: 0.348102 val accuracy: 0.371000
lr 3.666667e-07 reg 2.722222e+04 train accuracy: 0.338857 val accuracy: 0.332000
lr 3.666667e-07 reg 2.777778e+04 train accuracy: 0.346163 val accuracy: 0.366000
lr 3.666667e-07 reg 2.833333e+04 train accuracy: 0.340898 val accuracy: 0.352000
lr 3.666667e-07 reg 2.888889e+04 train accuracy: 0.349408 val accuracy: 0.354000
lr 3.666667e-07 reg 2.944444e+04 train accuracy: 0.345571 val accuracy: 0.341000
lr 3.666667e-07 reg 3.000000e+04 train accuracy: 0.342551 val accuracy: 0.337000
lr 4.111111e-07 reg 2.500000e+04 train accuracy: 0.341020 val accuracy: 0.340000
lr 4.111111e-07 reg 2.555556e+04 train accuracy: 0.345163 val accuracy: 0.359000
lr 4.111111e-07 reg 2.611111e+04 train accuracy: 0.348776 val accuracy: 0.355000
lr 4.111111e-07 reg 2.666667e+04 train accuracy: 0.338347 val accuracy: 0.356000
lr 4.111111e-07 reg 2.722222e+04 train accuracy: 0.336102 val accuracy: 0.344000
lr 4.111111e-07 reg 2.777778e+04 train accuracy: 0.326082 val accuracy: 0.346000
lr 4.111111e-07 reg 2.833333e+04 train accuracy: 0.342102 val accuracy: 0.356000
lr 4.111111e-07 reg 2.888889e+04 train accuracy: 0.344571 val accuracy: 0.365000
lr 4.111111e-07 reg 2.944444e+04 train accuracy: 0.323959 val accuracy: 0.322000
lr 4.111111e-07 reg 3.000000e+04 train accuracy: 0.347347 val accuracy: 0.369000
lr 4.555556e-07 reg 2.500000e+04 train accuracy: 0.340551 val accuracy: 0.337000
lr 4.555556e-07 reg 2.555556e+04 train accuracy: 0.337163 val accuracy: 0.351000
lr 4.555556e-07 reg 2.611111e+04 train accuracy: 0.344673 val accuracy: 0.346000
lr 4.555556e-07 reg 2.666667e+04 train accuracy: 0.354184 val accuracy: 0.376000
lr 4.555556e-07 reg 2.722222e+04 train accuracy: 0.345735 val accuracy: 0.356000
lr 4.555556e-07 reg 2.777778e+04 train accuracy: 0.347184 val accuracy: 0.366000
lr 4.555556e-07 reg 2.833333e+04 train accuracy: 0.340592 val accuracy: 0.340000
lr 4.555556e-07 reg 2.888889e+04 train accuracy: 0.351918 val accuracy: 0.355000
lr 4.555556e-07 reg 2.944444e+04 train accuracy: 0.334061 val accuracy: 0.352000
lr 4.555556e-07 reg 3.000000e+04 train accuracy: 0.323041 val accuracy: 0.338000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.338408 val accuracy: 0.348000
lr 5.000000e-07 reg 2.555556e+04 train accuracy: 0.324694 val accuracy: 0.339000
lr 5.000000e-07 reg 2.611111e+04 train accuracy: 0.337286 val accuracy: 0.348000
lr 5.000000e-07 reg 2.666667e+04 train accuracy: 0.323061 val accuracy: 0.345000
lr 5.000000e-07 reg 2.722222e+04 train accuracy: 0.338286 val accuracy: 0.352000
lr 5.000000e-07 reg 2.777778e+04 train accuracy: 0.347041 val accuracy: 0.358000
lr 5.000000e-07 reg 2.833333e+04 train accuracy: 0.316469 val accuracy: 0.314000
lr 5.000000e-07 reg 2.888889e+04 train accuracy: 0.330449 val accuracy: 0.364000
lr 5.000000e-07 reg 2.944444e+04 train accuracy: 0.329918 val accuracy: 0.327000
lr 5.000000e-07 reg 3.000000e+04 train accuracy: 0.311837 val accuracy: 0.324000
best validation accuracy achieved during cross-validation: 0.393000
```

In [25]:

```python
# 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.coolwarm)
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, cmap=plt.cm.coolwarm)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.show()
```



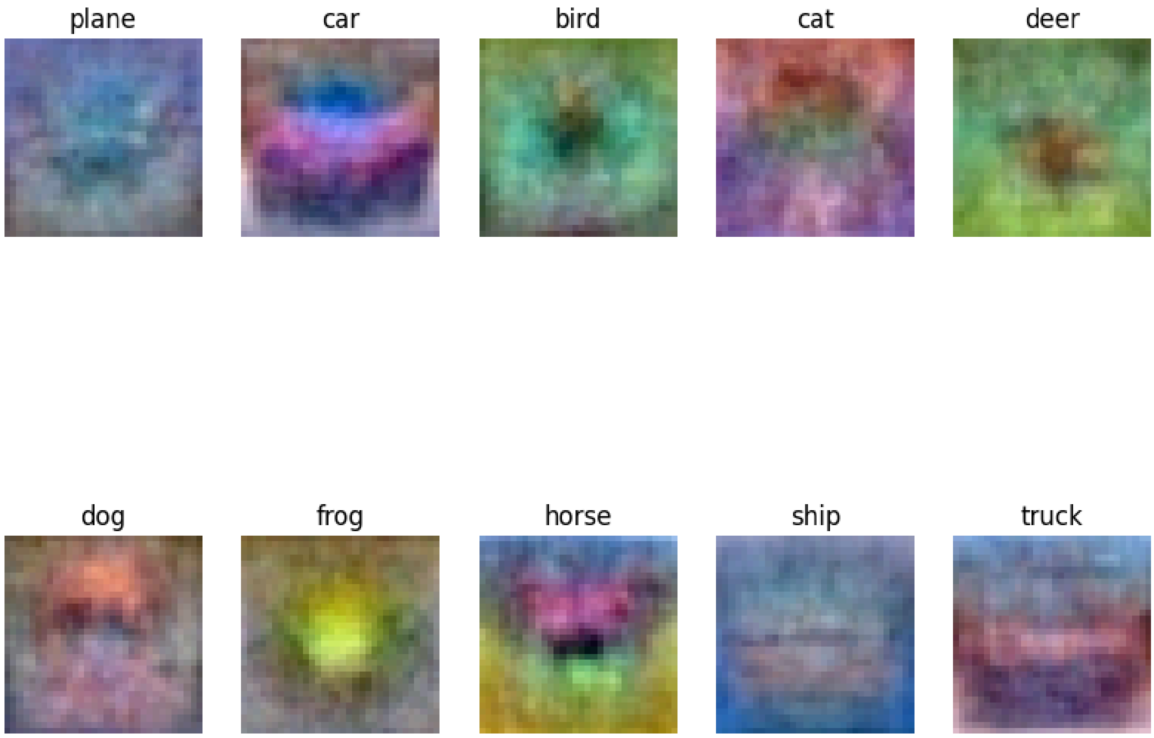


In [26]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.358000
```

In [27]:
```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, these m
# 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'
for i in range(10):
    plt.subplot(2, 5, i + 1)
```

```python
    # 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 the way they do.

$\color{blue}{\textit{Your Answer}}$: The weights for each class resemble a blurry prototype of what that class typically looks like.

The weights highlight the common colors typically associated with each class.

For objects that appear in multiple orientations, the weight visualizations often capture features from the visible directions, reflecting the model's attempt to generalize across different perspectives.