In [1]:
```
# enter the foldername in your Drive where you have saved the unzipped
# 'cs231n' folder containing the '.py', 'classifiers' and 'datasets'
# folders.
# e.g. 'cs231n/assignments/assignment1/cs231n/'

# %cd daseCV/datasets
# !bash ./get_datasets.sh
# %cd ../../
```

多分类支撑向量机练习

完成此练习并且上交本ipynb（包含输出及代码）.

在这个练习中，你将会:

- 为SVM构建一个完全向量化的**损失函数**
- 实现**解析梯度**的向量化表达式
- 使用数值梯度检查你的代码是否正确
- 使用验证集**调整学习率和正则化项**
- 用**SGD（随机梯度下降）优化**损失函数
- **可视化** 最后学习到的权重

In [2]:
```
# 导入包
import random
import numpy as np
from daseCV.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

# 下面一行是notebook的magic命令，作用是让matplotlib在notebook内绘图（而不是新建一个窗口）
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # 设置绘图的默认大小
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# 该magic命令可以重载外部的python模块
# 相关资料可以去看 http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipyt
%load_ext autoreload
%autoreload 2
```

准备和预处理CIFAR-10的数据

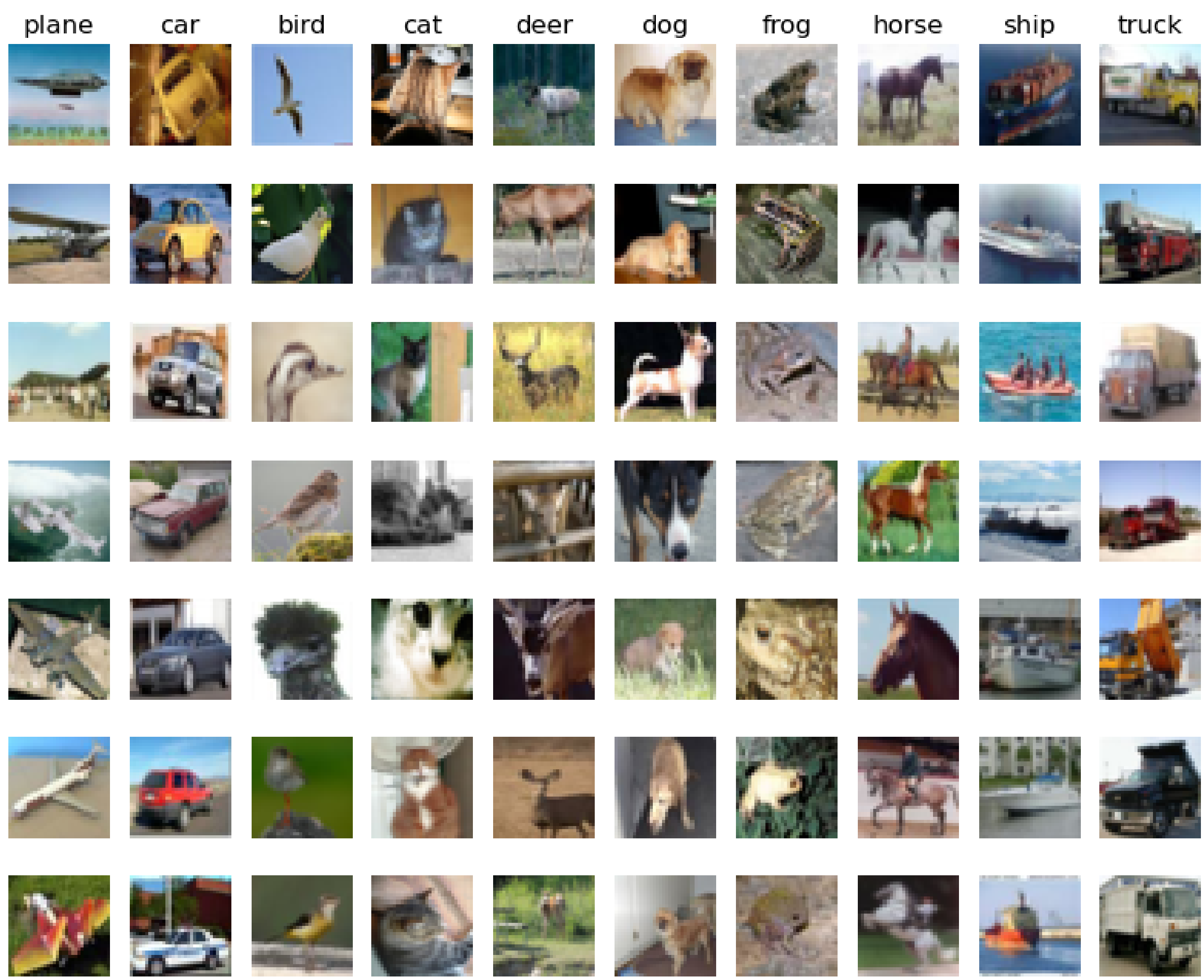
In [3]:
```
# 导入原始CIFAR-10数据
cifar10_dir = 'daseCV/datasets/cifar-10-batches-py'

# 清空变量，防止多次定义变量（可能造成内存问题）
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)

# 完整性检查，打印出训练和测试数据的大小
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]:

```
# 可视化部分数据
# 这里我们每个类别展示了7张图片
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 [5]:

```
# 划分训练集，验证集和测试集，除此之外，
# 我们从训练集中抽取了一小部分作为代码开发的数据，
# 使用小批量的开发数据集能够快速开发代码
num_training = 49000
```

```
num_validation = 1000
num_test = 1000
num_dev = 500

# 从原始训练集中抽取出num_validation个样本作为验证集
mask = range(num_training, num_training + num_validation)
X_val = X_train[mask]
y_val = y_train[mask]

# 从原始训练集中抽取出num_training个样本作为训练集
mask = range(num_training)
X_train = X_train[mask]
y_train = y_train[mask]

# 从训练集中抽取num_dev个样本作为开发数据集
mask = np.random.choice(num_training, num_dev, replace=False)
X_dev = X_train[mask]
y_dev = y_train[mask]

# 从原始测试集中抽取num_test个样本作为测试集
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 [6]:

```
# 预处理: 把图片数据rehspae成行向量
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))

# 完整性检查，打印出数据的shape
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)
```

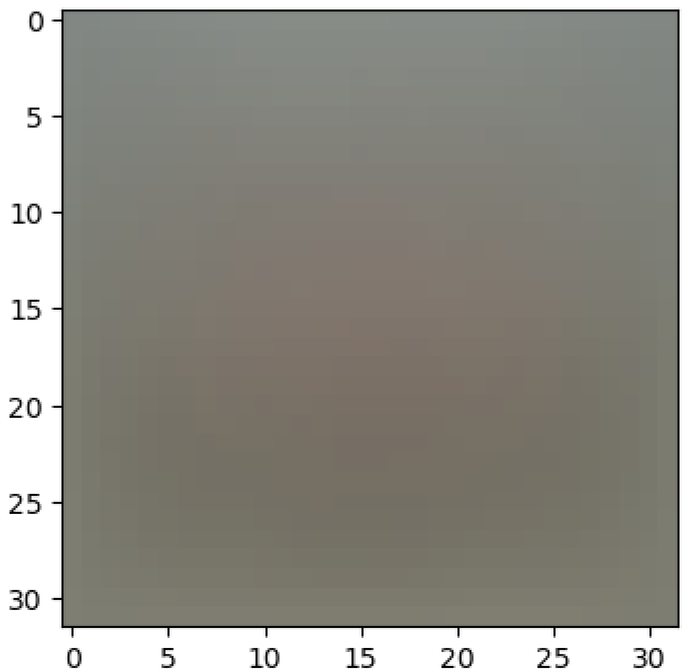
```
Training data shape:  (49000, 3072)
Validation data shape:  (1000, 3072)
Test data shape:  (1000, 3072)
dev data shape:  (500, 3072)
```

In [7]:

```
# 预处理: 减去image的平均值（均值规整化）
# 第一步: 计算训练集中的图像均值
mean_image = np.mean(X_train, axis=0)
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean image
plt.show()

# 第二步: 所有数据集减去均值
X_train -= mean_image
X_val -= mean_image
```

```
X_test -= mean_image
X_dev -= mean_image

print(X_train.shape, X_val.shape, X_test.shape, X_dev.shape)

# 第三步：拼接一个bias维，其中所有值都是1（bias trick），
# SVM可以联合优化数据和bias，即只需要优化一个权值矩阵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, 3072) (1000, 3072) (1000, 3072) (500, 3072)
(49000, 3073) (1000, 3073) (1000, 3073) (500, 3073)
```

SVM分类器

你需要在**daseCV/classifiers/linear_svm.py**里面完成编码

我们已经预先定义了一个函数 `compute_loss_naive` ，该函数使用循环来计算多分类SVM损失函数

In [8]:
```
# 调用朴素版的损失计算函数
from daseCV.classifiers.linear_svm import svm_loss_naive
import time

# 生成一个随机的SVM权值矩阵（矩阵值很小）
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: 8.647988
```

从上面的函数返回的 `grad` 现在是零。请推导支持向量机损失函数的梯度，并在svm_loss_naive中编码实现。

为了检查是否正确地实现了梯度，你可以用数值方法估计损失函数的梯度，并将数值估计与你计算出来的梯度进行比较。我们已经为你提供了检查的代码:

In [9]:

```
# 一旦你实现了梯度计算的功能，重新执行下面的代码检查梯度

# 计算损失和W的梯度
loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.0)

# 数值估计梯度的方法沿着随机几个维度进行计算，并且和解析梯度进行比较，
# 这两个方法算出来的梯度应该在任何维度上完全一致(相对误差足够小)
from daseCV.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)

# 把正则化项打开后继续再检查一遍梯度
# 你没有忘记正则化项吧？（忘了的罚抄100遍(๑•́ ₃•̀๑) ）
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: -3.026186 analytic: -3.026186, relative error: 9.872147e-11
numerical: -5.846811 analytic: -5.846811, relative error: 3.905217e-11
numerical: -11.248084 analytic: -11.248084, relative error: 1.356957e-11
numerical: -18.138456 analytic: -18.138456, relative error: 1.368176e-11
numerical: 13.632920 analytic: 13.649080, relative error: 5.923255e-04
numerical: 8.439588 analytic: 8.431843, relative error: 4.590774e-04
numerical: -38.467240 analytic: -38.467240, relative error: 6.726997e-12
numerical: 15.638514 analytic: 15.638514, relative error: 2.109249e-11
numerical: 0.310198 analytic: 0.310198, relative error: 1.444300e-09
numerical: 14.496186 analytic: 14.496186, relative error: 7.623066e-12
numerical: -0.848597 analytic: -0.848597, relative error: 1.034521e-10
numerical: 3.641614 analytic: 3.641614, relative error: 1.913280e-11
numerical: 28.866654 analytic: 28.866654, relative error: 4.123570e-13
numerical: -19.528132 analytic: -19.528132, relative error: 6.117783e-12
numerical: -22.075335 analytic: -22.075335, relative error: 1.224086e-11
numerical: 16.131988 analytic: 16.131988, relative error: 6.870229e-12
numerical: -4.414805 analytic: -4.414805, relative error: 4.203060e-11
numerical: 7.194422 analytic: 7.196766, relative error: 1.629059e-04
numerical: 15.474753 analytic: 15.452784, relative error: 7.103451e-04
numerical: -13.022915 analytic: -13.022915, relative error: 1.017698e-11
```

问题 1

有可能会出现某一个维度上的gradcheck没有完全匹配。这个问题是怎么引起的？有必要担心这个问题么？请举一个简单例子，能够导致梯度检查失败。如何改进这个问题？*提示：SVM的损失函数不是严格可微的*

你的回答：

令网络在各类型上的输出为 s_j . 这个问题是因为svm_loss函数在存在 j 使得 $s_i - s_j + 1 = 0$ 时是不可导的. (其中 i 是正确类别的标签) 考虑一个简单的二分类svm, 且输入只有一个特征, 它的在标签为'第2类'时损失函数是:

$$\max\{x \cdot w_1 - x \cdot w_2 + 1, 0\}$$

令 x 为 1 而 $w_1 - w_2 = -1.00001$, 此时 w_1 和 w_2 的解析梯度都为 0 . 如果计算 w_1 时, 选取步长为 $+0.00011$, 则得到的数值梯度是 $0.00010/0.00011$, 这个数字非常接近 1 .

通过这个简单的例子我们注意到两件事, 其一, 如果取步长方向相反, 即取步长为 -0.00011, 可以得到正确梯度 0 , 其二, 取步长足够小, 也能得到很接近正确结果的梯度.

因此计算数值梯度时, 选取的步长足够小, 或者选取恰当的方向也可避免这个问题, 使得对每个 j 都有 $s_i - s_j + 1$ 的符号不变, 就能避免这个问题. 或者不用存在 j 使得 $s_i - s_j + 1$ 非常接近零的情况来验证, 也不失为一种合理的解决方案.

In [10]:
```
# 接下来实现svm_loss_vectorized函数，目前只计算损失
# 稍后再计算梯度
tic = time.time()
loss_naive, grad_naive = svm_loss_naive(W, X_dev, y_dev, 5)
toc = time.time()
print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from daseCV.classifiers.linear_svm import svm_loss_vectorized
tic = time.time()
loss_vectorized, _ = svm_loss_vectorized(W, X_dev, y_dev, 5)
toc = time.time()
print('Vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

# 两种方法算出来的损失应该是相同的，但是向量化实现的方法应该更快
print('difference: %f' % (loss_naive - loss_vectorized))
```

```
Naive loss: 8.649550e+00 computed in 0.039999s
Vectorized loss: 8.649550e+00 computed in 0.009999s
difference: 0.000000
```

In [11]:
```
# 完成svm_loss_vectorized函数，并用向量化方法计算梯度

# 朴素方法和向量化实现的梯度应该相同，但是向量化方法也应该更快
tic = time.time()
_, grad_naive = svm_loss_naive(W, X_dev, y_dev, 5)
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, 5)
toc = time.time()
print('Vectorized loss and gradient: computed in %fs' % (toc - tic))

# 损失是一个标量，因此很容易比较两种方法算出的值，
# 而梯度是一个矩阵，所以我们用Frobenius范数来比较梯度的值
difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('difference: %f' % difference)
```

```
Naive loss and gradient: computed in 0.043001s
Vectorized loss and gradient: computed in 0.005008s
difference: 0.000000
```

随机梯度下降（Stochastic Gradient Descent）

我们现在有了向量化的损失函数表达式和梯度表达式，同时我们计算的梯度和数值梯度是匹配的。 接下来我们要做SGD。

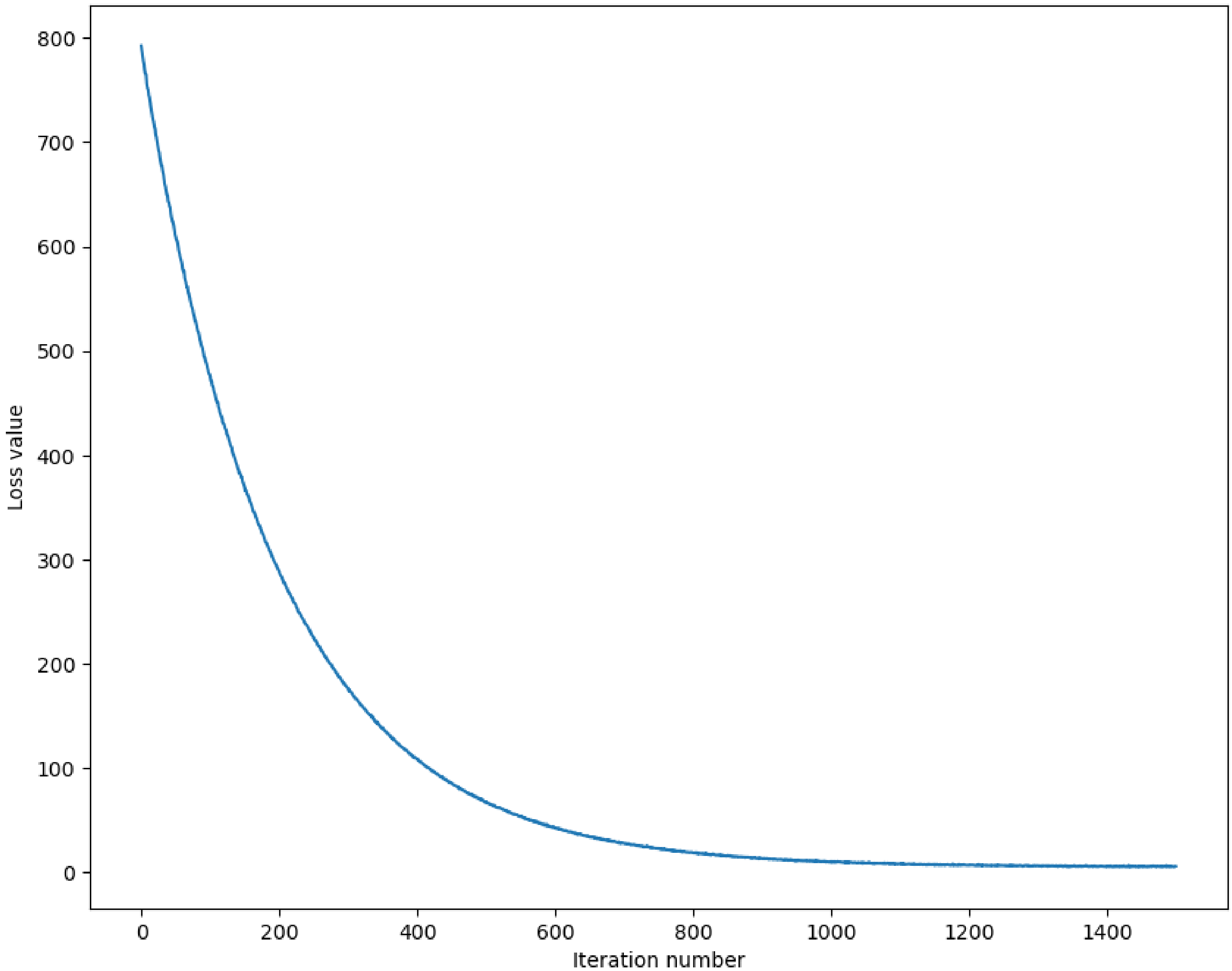
In [12]:
```
# 在linear_classifier.py文件中，编码实现LinearClassifier.train()中的SGD功能，
# 运行下面的代码
from daseCV.classifiers import LinearSVM
svm = LinearSVM()
tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=5e-8, reg=2.5e4, num_iters=1500, v
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 792.255224
iteration 100 / 1500: loss 475.095601
iteration 200 / 1500: loss 288.353593
iteration 300 / 1500: loss 176.284554
iteration 400 / 1500: loss 108.590557
iteration 500 / 1500: loss 68.131879
iteration 600 / 1500: loss 42.595670
iteration 700 / 1500: loss 28.064457
iteration 800 / 1500: loss 18.876689
```

```
iteration 900 / 1500: loss 13.624085
iteration 1000 / 1500: loss 9.757831
iteration 1100 / 1500: loss 8.490425
iteration 1200 / 1500: loss 7.427745
iteration 1300 / 1500: loss 6.340071
iteration 1400 / 1500: loss 6.012650
That took 4.617970s
```

In [13]:

```python
# 一个有用的debugging技巧是把损失函数画出来
plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```



In [14]:

```python
# 完成LinearSVM.predict函数,并且在训练集和验证集上评估其准确性
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: 0.373878
validation accuracy: 0.393000
```

In [15]:

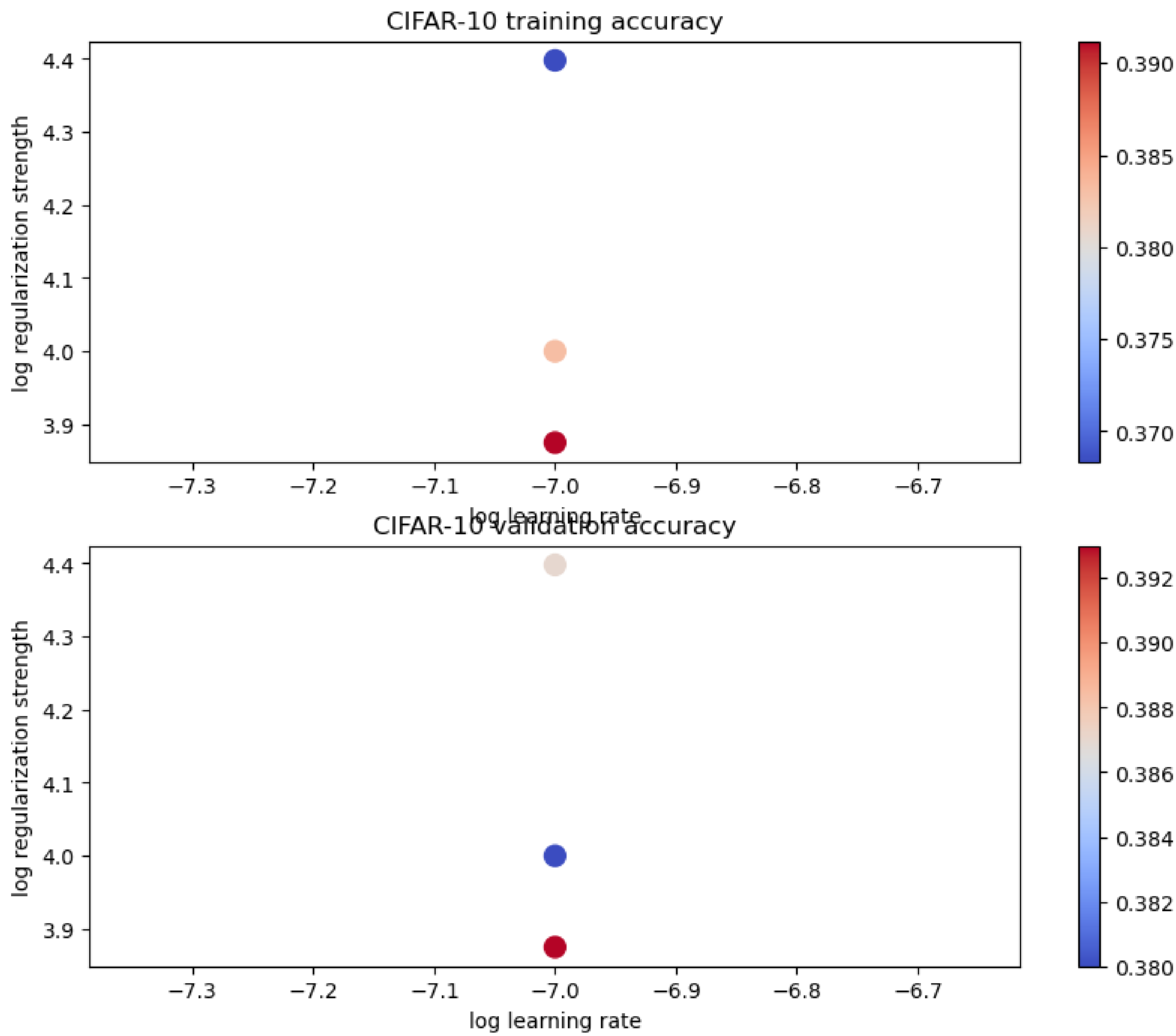
```python
# 使用验证集来调整超参数(正则化强度和学习率)。
# 你可以尝试不同的学习速率和正则化项的值;
# 如果你细心的话,您应该可以在验证集上获得大约0.39的准确率。

# 注意:在搜索超参数时, 您可能会看到runtime/overflow的警告。
# 这是由极端超参值造成的,不是代码的bug。
```

```python
learning_rates = [1e-7]
regularization_strengths = [7.5e3, 1e4, 2.5e4]

# results是一个字典,把元组(learning_rate, regularization_strength)映射到元组(training_accura
# accuracy是样本中正确分类的比例
results = {}
best_val = -1   # 我们迄今为止见过最好的验证集准确率
best_svm = None # 拥有最高验证集准确率的LinearSVM对象

################################################################################
# TODO:
# 编写代码，通过比较验证集的准确度来选择最佳超参数。
# 对于每个超参数组合，在训练集上训练一个线性SVM，在训练集和验证集上计算它的精度，
# 并将精度结果存储在results字典中。此外，在best_val中存储最高验证集准确度，
# 在best_svm中存储拥有此精度的SVM对象。
#
# 提示:
# 在开发代码时，应该使用一个比较小的num_iter值，这样SVM就不会花费太多时间训练;
# 一旦您确信您的代码开发完成，您就应该使用一个较大的num_iter值重新训练并验证。
################################################################################

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
import itertools
for lr, reg in itertools.product(learning_rates, regularization_strengths):
    svm = LinearSVM()
    svm.train(
        X_train, y_train,
        learning_rate=lr, reg=reg,
        num_iters=int(5000),
        verbose=False
    )
    val_acc_rate = np.mean(svm.predict(X_val) == y_val)
    trn_acc_rate = np.mean(svm.predict(X_train) == y_train)
    results[(lr, reg)] = (trn_acc_rate, val_acc_rate)
    if val_acc_rate > best_val:
        best_val = val_acc_rate
        best_svm = svm
        print('new best', best_val)
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# 打印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)
```

```
new best 0.393
lr 1.000000e-07 reg 7.500000e+03 train accuracy: 0.391184 val accuracy: 0.393000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.383102 val accuracy: 0.380000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.368347 val accuracy: 0.387000
best validation accuracy achieved during cross-validation: 0.393000
```

In [16]:
```python
# 可是化交叉验证结果
import math
x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

# 画出训练集准确率
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, cmap=plt.cm.coolwarm)
plt.colorbar()
plt.xlabel('log learning rate')
```

```python
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 training accuracy')

# 画出验证集准确率
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()
```

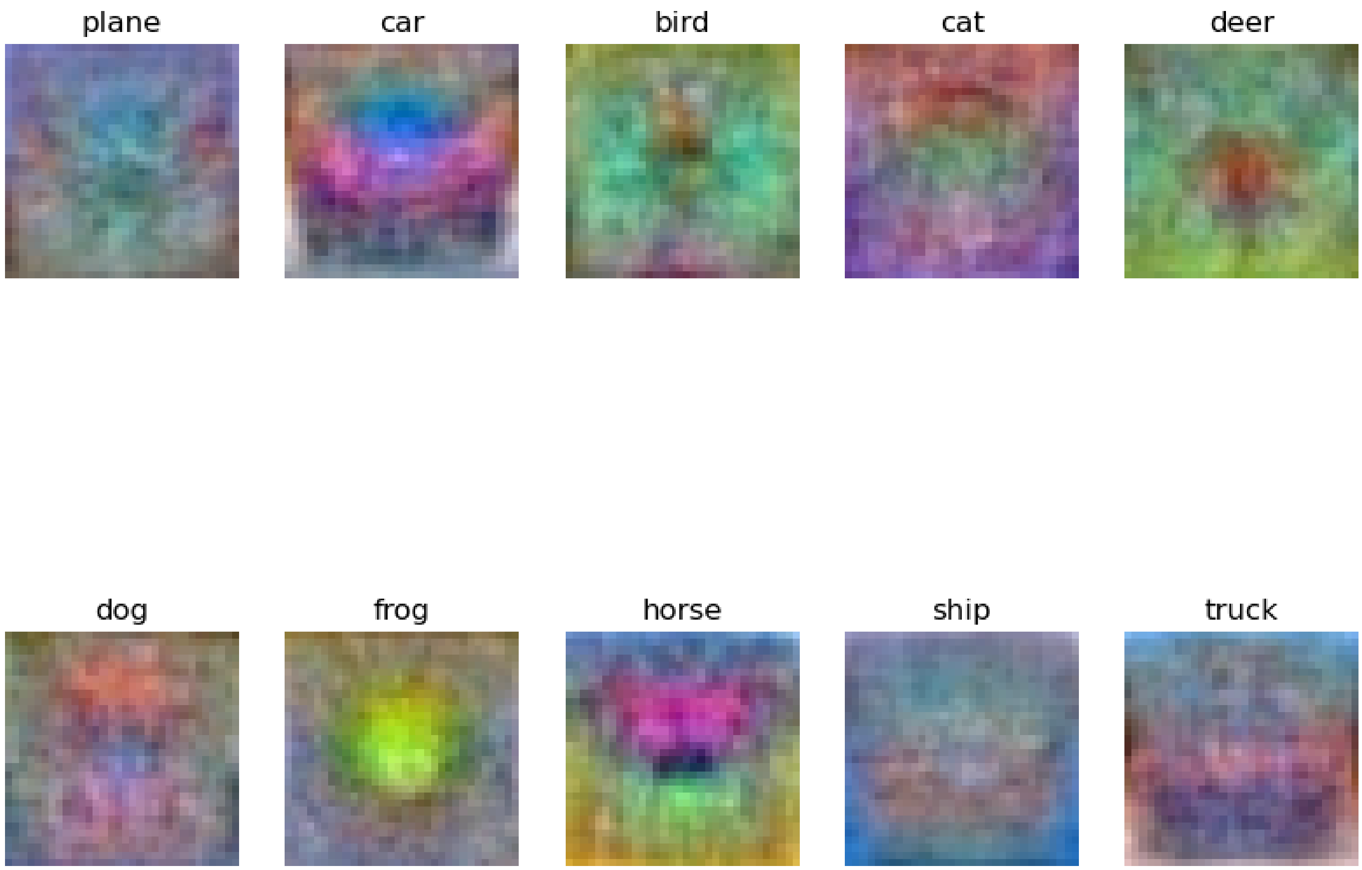


In [17]:
```python
# 在测试集上测试最好的SVM分类器
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.383000
```

In [18]:
```python
# 画出每一类的权重
# 基于您选择的学习速度和正则化强度，画出来的可能不好看
w = best_svm.W[:-1,:] # 去掉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)
    # 将权重调整为0到255之间
    wimg = 255.0 * (w[:, :, :, i].squeeze() - w_min) / (w_max - w_min)
    plt.imshow(wimg.astype('uint8'))
    plt.axis('off')
    plt.title(classes[i])
```



问题2

描述你的可视化权值是什么样子的，并提供一个简短的解释为什么它们看起来是这样的。

你的回答： 可视化权值和该分类的图片非常相似, 这是因为对每个类型的SVM需要关注这些类型更容易在哪几个(像素x,像素y,颜色)上具有更大的数值来得出这个类型的评分. 因此, 如果某种类型的图片上经常有某个像素显特定颜色, 那么SVM会为这个像素赋更大的权值, 反之, 则赋更小的权值, 甚至负值.

Data for leaderboard

这里额外提供了一组未给标签的测试集X，用于leaderborad上的竞赛。

提示：该题的目的是鼓励同学们探索能够提升模型性能的方法。

In [19]:
```
# leaderboard的测试数据
X = np.load("./input/X_3073.npy")
################################################################################
# 需要完成的事情:
# 找到更合适的svm
# 提示：如果你不想花时间，你也可以直接使用上面已经训练好的best_svm。
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

svm_leaderboard = best_svm
preds = svm_leaderboard.predict(X)
```

提醒：运行完下面代码之后，点击下面的submit，然后去leaderboard上查看你的成绩。本模型对应的成绩在phase2的leaderboard中。

In [20]:

```
import os
#输出格式
def output_file(preds, phase_id=2):
    path=os.getcwd()
    if not os.path.exists(path + '/output/phase_{}'.format(phase_id)):
        os.mkdir(path + '/output/phase_{}'.format(phase_id))
    path=path + '/output/phase_{}/prediction.npy'.format(phase_id)
    np.save(path,preds)
def zip_fun(phase_id=2):
    path=os.getcwd()
    output_path = path + '/output'
    files = os.listdir(output_path)
    for _file in files:
        if _file.find('zip') != -1:
            os.remove(output_path + '/' + _file)
    newpath=path+'/output/phase_{}'.format(phase_id)
    os.chdir(newpath)
    cmd = 'zip ../prediction_phase_{}.zip prediction.npy'.format(phase_id)
    os.system(cmd)
    os.chdir(path)
output_file(preds)
zip_fun()
```

In [ ]: