In [2]:
```
# 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
```

Softmax 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.

This exercise is analogous to the SVM exercise. You will:

- implement a fully-vectorized **loss function** for the Softmax classifier
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** with 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 [3]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

%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'

# for auto-reloading extenrnal modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipytho
```

```
%load_ext autoreload
%autoreload 2
```

In [4]:

```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, num
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the linear classifier. These are the same steps as we used for the
    SVM, but condensed to a single function.
    """
    # 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 ca
    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)

    # subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

    # 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))

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis = 0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image
    X_dev -= mean_image

    # add bias dimension and transform into columns
    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))])

    return X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev = get_CIFAR10_data(
```

```
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)
print('dev data shape: ', X_dev.shape)
print('dev labels shape: ', y_dev.shape)
```

```
Train data shape:  (49000, 3073)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3073)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3073)
Test labels shape:  (1000,)
dev data shape:  (500, 3073)
dev labels shape:  (500,)
```

Softmax Classifier

Your code for this section will all be written inside `cs231n/classifiers/softmax.py`.

In [11]:
```
# First implement the naive softmax loss function with nested loops.
# Open the file cs231n/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cs231n.classifiers.softmax import softmax_loss_naive
import time

# Generate a random softmax weight matrix and use it to compute the loss.
W = np.random.randn(3073, 10) * 0.0001
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# As a rough sanity check, our loss should be something close to -log(0.1).
print('loss: %f' % loss)
print('sanity check: %f' % (-np.log(0.1)))
```

```
loss: 2.325148
sanity check: 2.302585
```

Inline Question 1

Why do we expect our loss to be close to -log(0.1)? Explain briefly.**

YourAnswer : Since we have not trained the model, the W is just a random matrix, the scores of each class are expected to be roughly the same value.

Now we have 10 class in total, P(y_i|X_i) should tend to approach 1/10 = 0.1, then the loss is finally close to -log(0.1) for the reason of average.

In [13]:
```
# Complete the implementation of softmax_loss_naive and implement a (naive)
# version of the gradient that uses nested loops.
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# As we did for the SVM, use numeric gradient checking as a debugging tool.
# The numeric gradient should be close to the analytic gradient.
```

```
from cs231n.gradient_check import grad_check_sparse
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)

# similar to SVM case, do another gradient check with regularization
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 5e1)
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 5e1)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)
```

```
numerical: -0.450496 analytic: -0.450496, relative error: 3.019670e-08
numerical: 0.201822 analytic: 0.201822, relative error: 1.635290e-07
numerical: -2.368610 analytic: -2.368610, relative error: 3.911360e-08
numerical: -0.395033 analytic: -0.395033, relative error: 2.722459e-08
numerical: 0.643837 analytic: 0.643837, relative error: 8.720120e-08
numerical: 0.110287 analytic: 0.110287, relative error: 1.649100e-07
numerical: 1.425220 analytic: 1.425219, relative error: 3.869883e-08
numerical: -0.443472 analytic: -0.443472, relative error: 1.496069e-07
numerical: -2.414213 analytic: -2.414213, relative error: 2.605557e-08
numerical: -1.429808 analytic: -1.429808, relative error: 3.654863e-08
numerical: 3.099699 analytic: 3.099699, relative error: 2.118616e-08
numerical: -2.666262 analytic: -2.666262, relative error: 1.667081e-08
numerical: -0.429317 analytic: -0.429317, relative error: 7.231219e-09
numerical: 1.034238 analytic: 1.034238, relative error: 3.169477e-08
numerical: 0.638868 analytic: 0.638868, relative error: 4.610084e-08
numerical: 0.196967 analytic: 0.196967, relative error: 1.223871e-07
numerical: 0.659079 analytic: 0.659079, relative error: 1.032130e-07
numerical: -1.229625 analytic: -1.229625, relative error: 5.016976e-08
numerical: 0.313248 analytic: 0.313248, relative error: 6.095240e-08
numerical: -3.144537 analytic: -3.144537, relative error: 1.163897e-08
```

In [19]:
```
# Now that we have a naive implementation of the softmax loss function and its g
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version s
# much faster.
tic = time.time()
loss_naive, grad_naive = softmax_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.softmax import softmax_loss_vectorized
tic = time.time()
loss_vectorized, grad_vectorized = softmax_loss_vectorized(W, X_dev, y_dev, 0.00
toc = time.time()
print('vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

# As we did for the SVM, we use the Frobenius norm to compare the two versions
# of the gradient.
grad_difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('Loss difference: %f' % np.abs(loss_naive - loss_vectorized))
print('Gradient difference: %f' % grad_difference)
```

```
naive loss: 2.325148e+00 computed in 0.099868s
vectorized loss: 2.325148e+00 computed in 0.020783s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [26]:
```
# 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 over 0.35 on the validation set.
```

```
from cs231n.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained softmax classifer in best_softmax.                          #
################################################################################

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

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

learning_rates = np.linspace(3e-7, 5e-7, 10)
regularization_strengths = np.linspace(2.5e4, 5e4, 10)

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

    softmax = Softmax()
    softmax.train(X_train, y_train, lr, reg, num_iters=1500)
    y_train_pred = softmax.predict(X_train)
    train_accuracy = np.mean(y_train == y_train_pred)
    y_val_pred = softmax.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_softmax = softmax

# *****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 3.000000e-07 reg 2.500000e+04 train accuracy: 0.331918 val accuracy: 0.353000
lr 3.000000e-07 reg 2.777778e+04 train accuracy: 0.314265 val accuracy: 0.327000
lr 3.000000e-07 reg 3.055556e+04 train accuracy: 0.324082 val accuracy: 0.334000
lr 3.000000e-07 reg 3.333333e+04 train accuracy: 0.322245 val accuracy: 0.336000
lr 3.000000e-07 reg 3.611111e+04 train accuracy: 0.315959 val accuracy: 0.332000
lr 3.000000e-07 reg 3.888889e+04 train accuracy: 0.304918 val accuracy: 0.327000
lr 3.000000e-07 reg 4.166667e+04 train accuracy: 0.303531 val accuracy: 0.321000
lr 3.000000e-07 reg 4.444444e+04 train accuracy: 0.310571 val accuracy: 0.315000
lr 3.000000e-07 reg 4.722222e+04 train accuracy: 0.305939 val accuracy: 0.318000
lr 3.000000e-07 reg 5.000000e+04 train accuracy: 0.308245 val accuracy: 0.321000
lr 3.222222e-07 reg 2.500000e+04 train accuracy: 0.321918 val accuracy: 0.339000
lr 3.222222e-07 reg 2.777778e+04 train accuracy: 0.322694 val accuracy: 0.333000
lr 3.222222e-07 reg 3.055556e+04 train accuracy: 0.318000 val accuracy: 0.336000
lr 3.222222e-07 reg 3.333333e+04 train accuracy: 0.314571 val accuracy: 0.327000
lr 3.222222e-07 reg 3.611111e+04 train accuracy: 0.308959 val accuracy: 0.327000
lr 3.222222e-07 reg 3.888889e+04 train accuracy: 0.321714 val accuracy: 0.335000
lr 3.222222e-07 reg 4.166667e+04 train accuracy: 0.317286 val accuracy: 0.326000
lr 3.222222e-07 reg 4.444444e+04 train accuracy: 0.308939 val accuracy: 0.322000
lr 3.222222e-07 reg 4.722222e+04 train accuracy: 0.302571 val accuracy: 0.321000
lr 3.222222e-07 reg 5.000000e+04 train accuracy: 0.302816 val accuracy: 0.317000
lr 3.444444e-07 reg 2.500000e+04 train accuracy: 0.330020 val accuracy: 0.342000
lr 3.444444e-07 reg 2.777778e+04 train accuracy: 0.327347 val accuracy: 0.337000
lr 3.444444e-07 reg 3.055556e+04 train accuracy: 0.322388 val accuracy: 0.332000
lr 3.444444e-07 reg 3.333333e+04 train accuracy: 0.309755 val accuracy: 0.333000
lr 3.444444e-07 reg 3.611111e+04 train accuracy: 0.322592 val accuracy: 0.337000
lr 3.444444e-07 reg 3.888889e+04 train accuracy: 0.312592 val accuracy: 0.337000
lr 3.444444e-07 reg 4.166667e+04 train accuracy: 0.315347 val accuracy: 0.334000
lr 3.444444e-07 reg 4.444444e+04 train accuracy: 0.310408 val accuracy: 0.325000
lr 3.444444e-07 reg 4.722222e+04 train accuracy: 0.296449 val accuracy: 0.309000
lr 3.444444e-07 reg 5.000000e+04 train accuracy: 0.307449 val accuracy: 0.316000
lr 3.666667e-07 reg 2.500000e+04 train accuracy: 0.328265 val accuracy: 0.336000
lr 3.666667e-07 reg 2.777778e+04 train accuracy: 0.329469 val accuracy: 0.338000
lr 3.666667e-07 reg 3.055556e+04 train accuracy: 0.323061 val accuracy: 0.342000
lr 3.666667e-07 reg 3.333333e+04 train accuracy: 0.311388 val accuracy: 0.317000
lr 3.666667e-07 reg 3.611111e+04 train accuracy: 0.315122 val accuracy: 0.336000
lr 3.666667e-07 reg 3.888889e+04 train accuracy: 0.322755 val accuracy: 0.340000
lr 3.666667e-07 reg 4.166667e+04 train accuracy: 0.296837 val accuracy: 0.314000
lr 3.666667e-07 reg 4.444444e+04 train accuracy: 0.311980 val accuracy: 0.315000
lr 3.666667e-07 reg 4.722222e+04 train accuracy: 0.309592 val accuracy: 0.317000
lr 3.666667e-07 reg 5.000000e+04 train accuracy: 0.296857 val accuracy: 0.316000
lr 3.888889e-07 reg 2.500000e+04 train accuracy: 0.333612 val accuracy: 0.343000
lr 3.888889e-07 reg 2.777778e+04 train accuracy: 0.327265 val accuracy: 0.342000
lr 3.888889e-07 reg 3.055556e+04 train accuracy: 0.327286 val accuracy: 0.340000
lr 3.888889e-07 reg 3.333333e+04 train accuracy: 0.318184 val accuracy: 0.327000
lr 3.888889e-07 reg 3.611111e+04 train accuracy: 0.317163 val accuracy: 0.337000
lr 3.888889e-07 reg 3.888889e+04 train accuracy: 0.313816 val accuracy: 0.314000
lr 3.888889e-07 reg 4.166667e+04 train accuracy: 0.315449 val accuracy: 0.323000
lr 3.888889e-07 reg 4.444444e+04 train accuracy: 0.303714 val accuracy: 0.328000
lr 3.888889e-07 reg 4.722222e+04 train accuracy: 0.308041 val accuracy: 0.320000
lr 3.888889e-07 reg 5.000000e+04 train accuracy: 0.307204 val accuracy: 0.315000
lr 4.111111e-07 reg 2.500000e+04 train accuracy: 0.330265 val accuracy: 0.339000
lr 4.111111e-07 reg 2.777778e+04 train accuracy: 0.326429 val accuracy: 0.325000
lr 4.111111e-07 reg 3.055556e+04 train accuracy: 0.311694 val accuracy: 0.324000
lr 4.111111e-07 reg 3.333333e+04 train accuracy: 0.310163 val accuracy: 0.318000
lr 4.111111e-07 reg 3.611111e+04 train accuracy: 0.323082 val accuracy: 0.337000
lr 4.111111e-07 reg 3.888889e+04 train accuracy: 0.309959 val accuracy: 0.325000
lr 4.111111e-07 reg 4.166667e+04 train accuracy: 0.307082 val accuracy: 0.325000
lr 4.111111e-07 reg 4.444444e+04 train accuracy: 0.310020 val accuracy: 0.321000
lr 4.111111e-07 reg 4.722222e+04 train accuracy: 0.317898 val accuracy: 0.329000
lr 4.111111e-07 reg 5.000000e+04 train accuracy: 0.291245 val accuracy: 0.299000
```

```
lr 4.333333e-07 reg 2.500000e+04 train accuracy: 0.328286 val accuracy: 0.328000
lr 4.333333e-07 reg 2.777778e+04 train accuracy: 0.328429 val accuracy: 0.343000
lr 4.333333e-07 reg 3.055556e+04 train accuracy: 0.323388 val accuracy: 0.343000
lr 4.333333e-07 reg 3.333333e+04 train accuracy: 0.310939 val accuracy: 0.320000
lr 4.333333e-07 reg 3.611111e+04 train accuracy: 0.308245 val accuracy: 0.315000
lr 4.333333e-07 reg 3.888889e+04 train accuracy: 0.310633 val accuracy: 0.326000
lr 4.333333e-07 reg 4.166667e+04 train accuracy: 0.312102 val accuracy: 0.333000
lr 4.333333e-07 reg 4.444444e+04 train accuracy: 0.312755 val accuracy: 0.323000
lr 4.333333e-07 reg 4.722222e+04 train accuracy: 0.305449 val accuracy: 0.325000
lr 4.333333e-07 reg 5.000000e+04 train accuracy: 0.310020 val accuracy: 0.337000
lr 4.555556e-07 reg 2.500000e+04 train accuracy: 0.327653 val accuracy: 0.344000
lr 4.555556e-07 reg 2.777778e+04 train accuracy: 0.316694 val accuracy: 0.323000
lr 4.555556e-07 reg 3.055556e+04 train accuracy: 0.323796 val accuracy: 0.346000
lr 4.555556e-07 reg 3.333333e+04 train accuracy: 0.302714 val accuracy: 0.306000
lr 4.555556e-07 reg 3.611111e+04 train accuracy: 0.319347 val accuracy: 0.333000
lr 4.555556e-07 reg 3.888889e+04 train accuracy: 0.319061 val accuracy: 0.337000
lr 4.555556e-07 reg 4.166667e+04 train accuracy: 0.309429 val accuracy: 0.336000
lr 4.555556e-07 reg 4.444444e+04 train accuracy: 0.299694 val accuracy: 0.313000
lr 4.555556e-07 reg 4.722222e+04 train accuracy: 0.293306 val accuracy: 0.306000
lr 4.555556e-07 reg 5.000000e+04 train accuracy: 0.305000 val accuracy: 0.319000
lr 4.777778e-07 reg 2.500000e+04 train accuracy: 0.313408 val accuracy: 0.327000
lr 4.777778e-07 reg 2.777778e+04 train accuracy: 0.326367 val accuracy: 0.342000
lr 4.777778e-07 reg 3.055556e+04 train accuracy: 0.317551 val accuracy: 0.317000
lr 4.777778e-07 reg 3.333333e+04 train accuracy: 0.319918 val accuracy: 0.335000
lr 4.777778e-07 reg 3.611111e+04 train accuracy: 0.323306 val accuracy: 0.333000
lr 4.777778e-07 reg 3.888889e+04 train accuracy: 0.307041 val accuracy: 0.314000
lr 4.777778e-07 reg 4.166667e+04 train accuracy: 0.306714 val accuracy: 0.321000
lr 4.777778e-07 reg 4.444444e+04 train accuracy: 0.303367 val accuracy: 0.315000
lr 4.777778e-07 reg 4.722222e+04 train accuracy: 0.298959 val accuracy: 0.312000
lr 4.777778e-07 reg 5.000000e+04 train accuracy: 0.302837 val accuracy: 0.320000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.330653 val accuracy: 0.339000
lr 5.000000e-07 reg 2.777778e+04 train accuracy: 0.322755 val accuracy: 0.323000
lr 5.000000e-07 reg 3.055556e+04 train accuracy: 0.310918 val accuracy: 0.338000
lr 5.000000e-07 reg 3.333333e+04 train accuracy: 0.323204 val accuracy: 0.350000
lr 5.000000e-07 reg 3.611111e+04 train accuracy: 0.314327 val accuracy: 0.336000
lr 5.000000e-07 reg 3.888889e+04 train accuracy: 0.314204 val accuracy: 0.321000
lr 5.000000e-07 reg 4.166667e+04 train accuracy: 0.300184 val accuracy: 0.310000
lr 5.000000e-07 reg 4.444444e+04 train accuracy: 0.296122 val accuracy: 0.317000
lr 5.000000e-07 reg 4.722222e+04 train accuracy: 0.302429 val accuracy: 0.315000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.309245 val accuracy: 0.321000
best validation accuracy achieved during cross-validation: 0.353000
```

In [ ]:
```python
# evaluate on test set
# Evaluate the best softmax on test set
y_test_pred = best_softmax.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('softmax on raw pixels final test set accuracy: %f' % (test_accuracy, ))
```

Inline Question 2 - *True or False*

Suppose the overall training loss is defined as the sum of the per-datapoint loss over all training examples. It is possible to add a new datapoint to a training set that would leave the SVM loss unchanged, but this is not the case with the Softmax classifier loss.

$\mathit{Your Answer}$: It is hardly possible to leave the loss unchanged.

YourExplanation : The Softmax classifier loss is based on the probability distribution for each class. Adding a new datapoint will change the sum of the exponentials in the Softmax function, then probabilities for the existing classes will be changed, thus changing the overall loss.

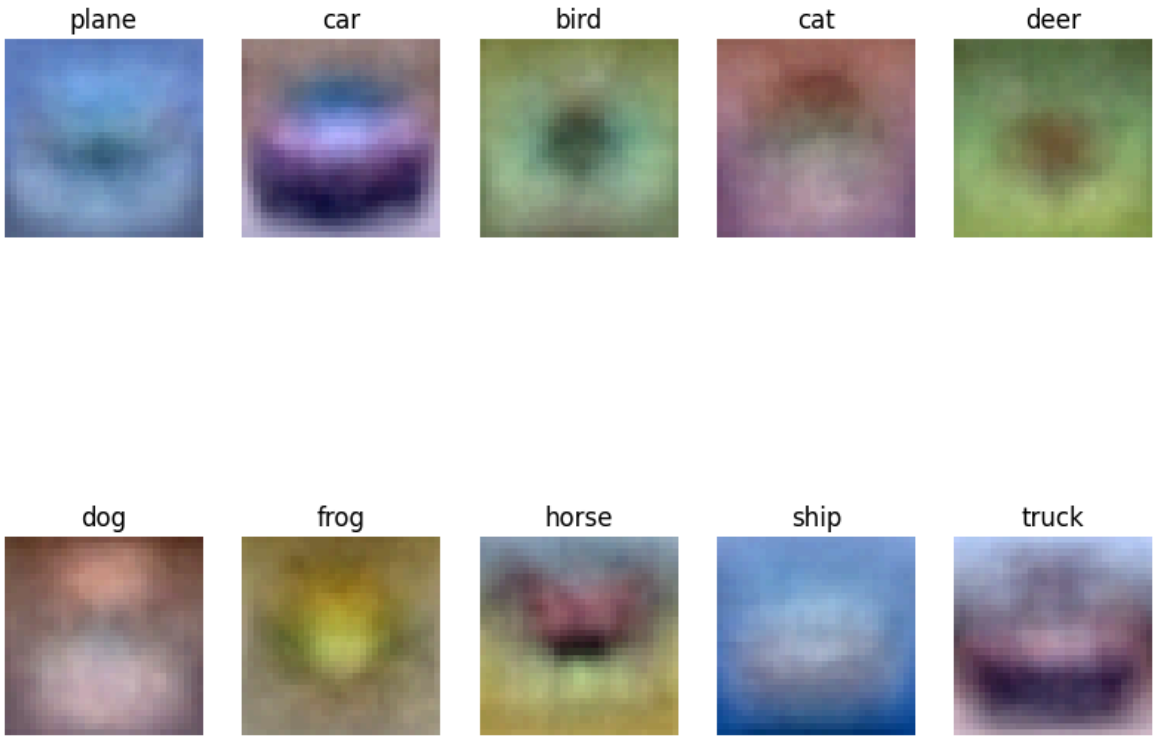
Therefore, Softmax loss typically does not remain unchanged when a new data point is added, as it directly affects the loss computation.

In [27]:
```
# Visualize the learned weights for each class
w = best_softmax.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)

    # 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])
```



In [ ]: