In [1]:

```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = "Colab Notebooks/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 drive/My₩ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My₩ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/Colab Notebooks/assignment1/cs231n/datasets
/content/drive/My Drive/Colab Notebooks/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 [2]:

```
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-ipython
%load_ext autoreload
%autoreload 2
```

In [3]:

```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, num_dev=
```

```python
    """
    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 cause me
    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 [5]:

```
# 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: 2.334964
sanity check: 2.302585
```

Inline Question 1

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

$\color{blue}{Your Answer:}$ 우리가 작성한 코드를 보면 *label class*의 수는 *10*개이고, 가중치를 *random*함수를 사용해 무작위로 설정하였으므로, 평균적으로 *1/10*의 정확도를 가지게 된다. 그러므로 수식에 따라 *-log(1/10) = -log(0.1)*에 가까운 *loss*를 갖게 된다.

In [6]:

```
# 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: 0.663658 analytic: 0.663658, relative error: 3.057256e-08
numerical: 0.863740 analytic: 0.863740, relative error: 6.476761e-08
numerical: -0.140858 analytic: -0.140858, relative error: 1.135148e-07
numerical: -0.002281 analytic: -0.002281, relative error: 2.127957e-06
numerical: -0.731742 analytic: -0.731742, relative error: 7.638961e-09
numerical: -0.131487 analytic: -0.131487, relative error: 3.927650e-08
numerical: 0.868624 analytic: 0.868624, relative error: 4.992314e-09
numerical: 1.656665 analytic: 1.656665, relative error: 1.203253e-08
```

```
numerical: -2.508937 analytic: -2.508937, relative error: 9.879609e-09
numerical: -2.908926 analytic: -2.908926, relative error: 2.805085e-08
numerical: 1.088320 analytic: 1.088320, relative error: 8.855573e-09
numerical: 0.478719 analytic: 0.478719, relative error: 1.649932e-08
numerical: -0.449663 analytic: -0.449663, relative error: 9.521726e-08
numerical: 0.999332 analytic: 0.999332, relative error: 9.147471e-09
numerical: 0.307334 analytic: 0.307334, relative error: 1.249076e-08
numerical: 0.671166 analytic: 0.671166, relative error: 4.660121e-08
numerical: 1.293835 analytic: 1.293835, relative error: 6.446288e-09
numerical: 2.931522 analytic: 2.931522, relative error: 1.318756e-08
numerical: 3.886253 analytic: 3.886253, relative error: 3.958368e-09
numerical: 3.456118 analytic: 3.456118, relative error: 1.277052e-09
```

In [7]:

```
# Now that we have a naive implementation of the softmax loss function and its gradier
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version should
# 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, 0.000005)
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: 2.334964e+00 computed in 0.148034s
vectorized loss: 2.334964e+00 computed in 0.018492s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [18]:

```
# 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 hyperparameters
learning_rates = [1e-7, 5e-7]
regularization_strengths = [2.5e4, 5e4]

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

from itertools import product
grid_search = list(product(*[learning_rates,regularization_strengths]))
```

```
for lr, rg in grid_search :
        soft = Softmax()

        soft.train(X_train, y_train, learning_rate = lr, reg = rg,
                   num_iters = 1000)

        y_train_predict = soft.predict(X_train)
        learning_accuracy = np.mean(y_train_predict == y_train)

        y_validation_predict = soft.predict(X_val)
        validation_accuracy = np.mean(y_validation_predict == y_val)

        results[(lr,rg)] = (learning_accuracy, validation_accuracy)

        if best_val < validation_accuracy :
          best_val = validation_accuracy
          best_softmax = soft

# *****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: 0.327000 val accuracy: 0.340000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.309673 val accuracy: 0.320000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.332939 val accuracy: 0.351000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.293857 val accuracy: 0.307000
best validation accuracy achieved during cross-validation: 0.351000
```

In [19]:

```
# 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, ))
```

```
softmax on raw pixels final test set accuracy: 0.351000
```

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.

$Your Answer$: True

$Your Explanation$: 일반적으로, SVM은 robust하며 어느정도 이상의 성능을 갖추게 되면 loss가 잘 변하지 않는다. 하지만, Softmax는 정확도가 1에 근접할 때까지 계속 성능을 계속하는 것이 특징이므로 True이다.

In [20]:

```
# 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', 'tru
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])
```