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 (https://compsci682-fa19.github.io/assignments2019/assignment1/)](https://compsci682-fa19.github.io/assignments2019/assignment1/) 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]:
```
from __future__ import print_function
import random
import numpy as np
from cs682.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 [4]:
```python
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, num_dev=500):
    """
    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 = 'cs682/datasets/cifar-10-batches-py'

    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


# Cleaning up variables to prevent loading data multiple times (which may cause memory issue)
try:
   del X_train, y_train
   del X_test, y_test
   print('Clear previously loaded data.')
except:
   pass

# 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 **cs682/classifiers/softmax.py**.

In [5]:
```
# First implement the naive softmax loss function with nested loops.
# Open the file cs682/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cs682.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: 2.378209
sanity check: 2.302585
```

Inline Question 1:

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

Your answer: With random weight initialization with values in the range 0.0001, the scores for each of the classes would be very small values in a similar range. The exponentiation of such a small value would be very close to 1 and hence the softmax term becomes -log(s/(10*s)) = -log(0.1).

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 cs682.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: -0.078319 analytic: -0.078319, relative error: 3.021400e-07
numerical: -2.730777 analytic: -2.730777, relative error: 2.790426e-09
numerical: -0.047185 analytic: -0.047185, relative error: 3.914673e-07
numerical: 0.671479 analytic: 0.671479, relative error: 1.064990e-07
numerical: -0.932518 analytic: -0.932518, relative error: 2.339729e-08
numerical: 0.437400 analytic: 0.437400, relative error: 6.794517e-08
numerical: -0.376330 analytic: -0.376330, relative error: 5.018106e-08
numerical: -0.503356 analytic: -0.503356, relative error: 3.716490e-08
numerical: 0.561375 analytic: 0.561375, relative error: 4.432825e-08
numerical: -0.673301 analytic: -0.673301, relative error: 4.080552e-08
numerical: 0.739985 analytic: 0.739985, relative error: 8.303052e-09
numerical: 0.999494 analytic: 0.999494, relative error: 3.116367e-08
numerical: -3.115712 analytic: -3.115712, relative error: 9.551139e-09
numerical: -0.214271 analytic: -0.214271, relative error: 1.673878e-07
numerical: 3.611589 analytic: 3.611589, relative error: 1.273046e-08
numerical: -0.400190 analytic: -0.400190, relative error: 3.922155e-08
numerical: 0.586929 analytic: 0.586929, relative error: 1.090298e-08
numerical: 1.714036 analytic: 1.714036, relative error: 1.378063e-08
numerical: -0.210448 analytic: -0.210448, relative error: 3.865159e-10
numerical: 2.001744 analytic: 2.001744, relative error: 2.666238e-09
```

In [7]:

```python
# Now that we have a naive implementation of the softmax loss function and its gradient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version should be
# 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 cs682.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: 2.378209e+00 computed in 12.622023s
vectorized loss: 2.378209e+00 computed in 0.075435s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [16]:

```python
# 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 cs682.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None
# learning_rates = [3e-7, 4e-7, 5e-7, 6e-7, 7e-7]
# regularization_strengths = [1e3, 5e3, 1e4, 2e4, 3e4, 4e4]

################################################################################
# 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.                          #
################################################################################
np.random.seed(123)
N = X_train.shape[0]
D = X_train.shape[1]
C = W.shape[1]
num_trials = 100
for t in range(num_trials):
#   print ('learning rate: %f, regularization strength: %f' %(lr, rs))
    lr = 10 ** np.random.uniform (-7.5, -5)
    rs = 10 ** np.random.uniform (3,5)
    softmax = Softmax()
    loss = softmax.train(X_train, y_train, learning_rate=lr, reg=rs,
                  num_iters=1000, verbose=False)
    y_train_pred = softmax.predict (X_train)
    train_acc = np.mean(y_train_pred == y_train)

    y_val_pred = softmax.predict (X_val)
    val_acc = np.mean(y_val_pred == y_val)

    results[(lr, rs)] = (train_acc, val_acc)
    if val_acc > best_val:
        best_val = val_acc
        best_softmax = softmax

################################################################################
#                              END OF YOUR CODE                                #
################################################################################

# 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 3.258080e-08 reg 1.348217e+03 train accuracy: 0.191082 val accuracy: 0.209000
lr 3.321815e-08 reg 1.751938e+03 train accuracy: 0.174918 val accuracy: 0.176000
lr 3.559932e-08 reg 1.225499e+04 train accuracy: 0.226776 val accuracy: 0.227000
lr 3.569976e-08 reg 3.549747e+04 train accuracy: 0.292531 val accuracy: 0.306000
lr 4.099158e-08 reg 8.048387e+03 train accuracy: 0.220857 val accuracy: 0.231000
lr 4.766177e-08 reg 2.596421e+04 train accuracy: 0.307796 val accuracy: 0.320000
lr 5.050069e-08 reg 2.733333e+04 train accuracy: 0.313347 val accuracy: 0.328000
lr 5.314633e-08 reg 2.521249e+04 train accuracy: 0.321653 val accuracy: 0.322000
lr 5.759198e-08 reg 6.727783e+03 train accuracy: 0.242776 val accuracy: 0.268000
lr 6.176492e-08 reg 1.735001e+03 train accuracy: 0.212673 val accuracy: 0.204000
lr 6.427551e-08 reg 8.594780e+03 train accuracy: 0.273327 val accuracy: 0.288000
lr 6.663849e-08 reg 2.534973e+04 train accuracy: 0.324367 val accuracy: 0.350000
lr 7.575469e-08 reg 2.505885e+04 train accuracy: 0.326551 val accuracy: 0.340000
lr 7.664551e-08 reg 3.318633e+03 train accuracy: 0.251102 val accuracy: 0.263000
lr 7.726900e-08 reg 1.237432e+03 train accuracy: 0.229082 val accuracy: 0.233000
lr 7.968840e-08 reg 1.446854e+04 train accuracy: 0.328776 val accuracy: 0.346000
lr 8.284518e-08 reg 2.132810e+03 train accuracy: 0.234286 val accuracy: 0.253000
lr 8.346735e-08 reg 1.301628e+04 train accuracy: 0.328939 val accuracy: 0.355000
lr 8.716513e-08 reg 1.584042e+04 train accuracy: 0.335592 val accuracy: 0.350000
lr 9.214246e-08 reg 3.833329e+04 train accuracy: 0.316449 val accuracy: 0.332000
lr 1.018775e-07 reg 3.275782e+04 train accuracy: 0.318694 val accuracy: 0.337000
lr 1.026013e-07 reg 2.098975e+04 train accuracy: 0.334061 val accuracy: 0.353000
lr 1.031189e-07 reg 3.328637e+04 train accuracy: 0.321102 val accuracy: 0.338000
lr 1.033242e-07 reg 1.461397e+04 train accuracy: 0.343633 val accuracy: 0.356000
lr 1.212109e-07 reg 2.340142e+04 train accuracy: 0.332265 val accuracy: 0.347000
lr 1.419320e-07 reg 7.636723e+03 train accuracy: 0.349612 val accuracy: 0.367000
lr 1.425447e-07 reg 1.522328e+04 train accuracy: 0.339980 val accuracy: 0.363000
lr 1.425751e-07 reg 1.047021e+03 train accuracy: 0.262286 val accuracy: 0.269000
lr 1.546193e-07 reg 1.749254e+03 train accuracy: 0.276286 val accuracy: 0.278000
lr 1.576817e-07 reg 5.891604e+04 train accuracy: 0.307102 val accuracy: 0.319000
lr 1.602274e-07 reg 6.825114e+04 train accuracy: 0.283490 val accuracy: 0.294000
lr 1.694010e-07 reg 2.353230e+04 train accuracy: 0.334449 val accuracy: 0.345000
lr 1.749800e-07 reg 4.100961e+03 train accuracy: 0.336959 val accuracy: 0.338000
lr 1.795360e-07 reg 2.595024e+04 train accuracy: 0.320041 val accuracy: 0.336000
lr 1.827506e-07 reg 3.035897e+04 train accuracy: 0.316224 val accuracy: 0.334000
lr 2.168210e-07 reg 3.201254e+04 train accuracy: 0.322653 val accuracy: 0.337000
lr 2.200686e-07 reg 2.197883e+04 train accuracy: 0.328490 val accuracy: 0.338000
lr 2.375602e-07 reg 2.989391e+03 train accuracy: 0.348429 val accuracy: 0.368000
lr 2.446575e-07 reg 3.883925e+03 train accuracy: 0.361837 val accuracy: 0.363000
lr 2.480178e-07 reg 4.489681e+04 train accuracy: 0.304327 val accuracy: 0.318000
lr 2.573337e-07 reg 1.236507e+04 train accuracy: 0.351469 val accuracy: 0.366000
lr 2.597383e-07 reg 2.407501e+03 train accuracy: 0.345837 val accuracy: 0.374000
lr 2.615648e-07 reg 3.740476e+04 train accuracy: 0.314551 val accuracy: 0.326000
lr 2.889722e-07 reg 3.543106e+04 train accuracy: 0.318286 val accuracy: 0.335000
lr 3.042534e-07 reg 1.771001e+03 train accuracy: 0.343306 val accuracy: 0.351000
lr 3.238884e-07 reg 6.296680e+03 train accuracy: 0.366367 val accuracy: 0.390000
lr 3.357618e-07 reg 1.992543e+03 train accuracy: 0.358265 val accuracy: 0.362000
lr 3.385263e-07 reg 1.546869e+03 train accuracy: 0.342898 val accuracy: 0.350000
lr 3.415061e-07 reg 7.319371e+04 train accuracy: 0.303714 val accuracy: 0.320000
lr 3.593983e-07 reg 6.504733e+04 train accuracy: 0.302347 val accuracy: 0.320000
lr 3.849291e-07 reg 5.759985e+04 train accuracy: 0.301776 val accuracy: 0.309000
lr 4.433648e-07 reg 8.990631e+03 train accuracy: 0.360633 val accuracy: 0.372000
lr 4.479656e-07 reg 4.042973e+03 train accuracy: 0.376714 val accuracy: 0.396000
lr 4.569945e-07 reg 4.084633e+03 train accuracy: 0.379714 val accuracy: 0.389000
lr 4.892519e-07 reg 1.109545e+04 train accuracy: 0.349490 val accuracy: 0.358000
lr 4.951370e-07 reg 4.856733e+03 train accuracy: 0.372204 val accuracy: 0.374000
lr 5.071041e-07 reg 3.887843e+04 train accuracy: 0.298531 val accuracy: 0.313000
lr 5.523354e-07 reg 4.033332e+04 train accuracy: 0.317286 val accuracy: 0.326000
lr 5.731314e-07 reg 9.626114e+04 train accuracy: 0.265020 val accuracy: 0.274000
lr 6.821113e-07 reg 9.621291e+04 train accuracy: 0.276286 val accuracy: 0.275000
lr 7.004375e-07 reg 1.596999e+04 train accuracy: 0.334204 val accuracy: 0.347000
lr 7.146074e-07 reg 1.777744e+04 train accuracy: 0.335388 val accuracy: 0.357000
lr 8.174599e-07 reg 5.482741e+04 train accuracy: 0.279061 val accuracy: 0.285000
lr 8.490516e-07 reg 5.395744e+04 train accuracy: 0.290735 val accuracy: 0.298000
lr 8.684032e-07 reg 3.070606e+04 train accuracy: 0.304816 val accuracy: 0.318000
lr 1.056483e-06 reg 7.080378e+04 train accuracy: 0.283082 val accuracy: 0.304000
lr 1.110817e-06 reg 1.374484e+03 train accuracy: 0.398898 val accuracy: 0.404000
lr 1.136732e-06 reg 9.745731e+03 train accuracy: 0.348531 val accuracy: 0.352000
lr 1.188787e-06 reg 1.156345e+03 train accuracy: 0.394449 val accuracy: 0.399000
lr 1.263509e-06 reg 1.559978e+03 train accuracy: 0.389143 val accuracy: 0.394000
lr 1.320554e-06 reg 2.454567e+04 train accuracy: 0.313673 val accuracy: 0.316000
lr 1.341472e-06 reg 1.306028e+03 train accuracy: 0.394408 val accuracy: 0.397000
lr 1.615486e-06 reg 1.558226e+03 train accuracy: 0.388082 val accuracy: 0.383000
lr 1.641789e-06 reg 2.585399e+03 train accuracy: 0.383122 val accuracy: 0.396000
lr 1.644336e-06 reg 2.738173e+04 train accuracy: 0.303143 val accuracy: 0.307000
lr 1.662881e-06 reg 9.937517e+03 train accuracy: 0.344939 val accuracy: 0.350000
```

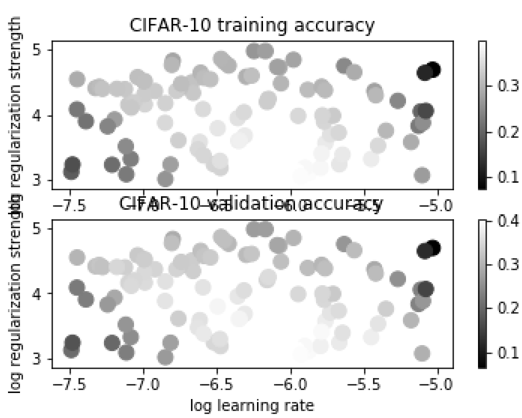
```
lr 1.703983e-06 reg 7.937453e+03 train accuracy: 0.345714 val accuracy: 0.352000
lr 1.734487e-06 reg 2.412066e+04 train accuracy: 0.311224 val accuracy: 0.319000
lr 1.742501e-06 reg 3.734897e+03 train accuracy: 0.372000 val accuracy: 0.379000
lr 1.878426e-06 reg 5.054512e+03 train accuracy: 0.360490 val accuracy: 0.367000
lr 1.966854e-06 reg 9.815506e+03 train accuracy: 0.331388 val accuracy: 0.332000
lr 2.149954e-06 reg 5.393259e+03 train accuracy: 0.347959 val accuracy: 0.357000
lr 2.167211e-06 reg 1.760490e+03 train accuracy: 0.366408 val accuracy: 0.375000
lr 2.280421e-06 reg 2.482905e+03 train accuracy: 0.370122 val accuracy: 0.363000
lr 2.340774e-06 reg 5.699793e+04 train accuracy: 0.248490 val accuracy: 0.264000
lr 2.740301e-06 reg 4.567705e+04 train accuracy: 0.293204 val accuracy: 0.315000
lr 3.574047e-06 reg 2.125066e+03 train accuracy: 0.374041 val accuracy: 0.381000
lr 3.762056e-06 reg 2.342654e+04 train accuracy: 0.278735 val accuracy: 0.289000
lr 3.839631e-06 reg 2.013811e+04 train accuracy: 0.307980 val accuracy: 0.314000
lr 3.983799e-06 reg 3.419115e+03 train accuracy: 0.349612 val accuracy: 0.368000
lr 4.508858e-06 reg 5.783580e+03 train accuracy: 0.306776 val accuracy: 0.338000
lr 5.427528e-06 reg 1.660644e+04 train accuracy: 0.245510 val accuracy: 0.256000
lr 6.683194e-06 reg 3.864594e+03 train accuracy: 0.292245 val accuracy: 0.306000
lr 7.515159e-06 reg 6.861880e+03 train accuracy: 0.228551 val accuracy: 0.238000
lr 7.776557e-06 reg 1.141131e+04 train accuracy: 0.201347 val accuracy: 0.235000
lr 7.954041e-06 reg 1.195963e+03 train accuracy: 0.272633 val accuracy: 0.299000
lr 8.060313e-06 reg 7.657722e+03 train accuracy: 0.254306 val accuracy: 0.266000
lr 8.301652e-06 reg 4.454291e+04 train accuracy: 0.122449 val accuracy: 0.122000
lr 8.424696e-06 reg 1.163300e+04 train accuracy: 0.167184 val accuracy: 0.156000
lr 9.378184e-06 reg 4.972532e+04 train accuracy: 0.073776 val accuracy: 0.065000
best validation accuracy achieved during cross-validation: 0.404000
```

In [17]:

```python
# Visualize the cross-validation results
import math
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.scatter(x_scatter, y_scatter, marker_size, c=colors)
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)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.show()
```



In [18]:

```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, ))
```

```
softmax on raw pixels final test set accuracy: 0.376000
```

Inline Question - *True or False*

It's 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: Any datapoint which has a score beyond the margin with respect to the correct class score will contribute a 0 towards the loss. Adding such a datapoint would leave the SVM loss unchanged. However, in the case of softmax, this datapoint would have a non-zero contribution towards the loss.

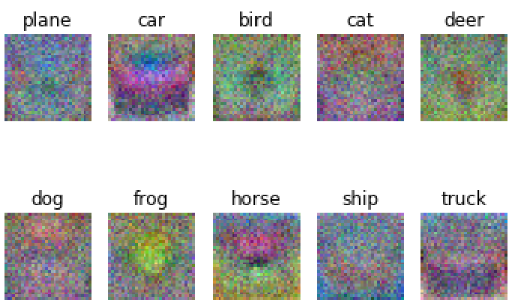
In [19]:

```python
# 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', 'truck']
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 [ ]: