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 [5]:

```
from __future__ import print_function
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
print ("setup done\n")
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
setup done
```

In [6]:

```
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 = 'cs231n/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)
```

```
Clear previously loaded data.
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 [8]:

```
# 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: 2.334109
sanity check: 2.302585
```

Inline Question 1:

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

Your answer: *For our data, there are 10 possible classes and only one correct class for the ground truth tables. There is a 0.1 chance of random guess for each class, our loss is giving unnormalised negative log probabilities of each class so should be close to -log(0.1) at the start (random guess).*

In [10]:

```
# 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)
print ("done checking :) \n ")
```

```
numerical: 2.165246 analytic: 2.165246, relative error: 4.476693e-08
numerical: -0.863869 analytic: -0.863870, relative error: 1.692829e-08
numerical: -0.243418 analytic: -0.243418, relative error: 1.595240e-07
numerical: 3.573190 analytic: 3.573190, relative error: 2.497173e-08
numerical: -1.534077 analytic: -1.534077, relative error: 9.325362e-09
numerical: -0.585371 analytic: -0.585371, relative error: 5.199210e-08
numerical: -2.554850 analytic: -2.554850, relative error: 5.046925e-09
numerical: 0.770071 analytic: 0.770071, relative error: 6.735276e-08
numerical: 0.996441 analytic: 0.996440, relative error: 1.859009e-08
numerical: 3.407291 analytic: 3.407291, relative error: 8.668087e-09
numerical: 2.490740 analytic: 2.490739, relative error: 3.269095e-08
numerical: 0.018444 analytic: 0.018444, relative error: 1.160115e-06
numerical: 0.086835 analytic: 0.086835, relative error: 6.972341e-07
numerical: -0.788320 analytic: -0.788320, relative error: 5.693500e-09
numerical: 0.637140 analytic: 0.637140, relative error: 4.338068e-08
numerical: 0.027111 analytic: 0.027111, relative error: 7.831247e-07
numerical: 0.908870 analytic: 0.908870, relative error: 6.808412e-08
numerical: 0.228242 analytic: 0.228242, relative error: 4.526002e-08
numerical: 1.509984 analytic: 1.509984, relative error: 3.600044e-08
numerical: -1.879903 analytic: -1.879903, relative error: 4.262893e-08
done checking :)
```

In [11]:

```
# 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 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: 2.334109e+00 computed in 0.231904s
vectorized loss: 2.334109e+00 computed in 0.015656s
```

```
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [13]:

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

################################################################################
# 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.                          #
################################################################################

# additional variables
best_lr = None
best_reg = None

range_lr = np.linspace(learning_rates[0],learning_rates[1],5)
range_reg = np.linspace(regularization_strengths[0],regularization_strengths[1],5)

print ("strating loop ! This may take a while\n\n")
# loop through all the combinations
for cur_lr in range_lr: #go over the learning rates
    for cur_reg in range_reg:#go over the regularization strength
        # initiate linear classifier with hyperparameters
        smx = Softmax()
        smx.train(X_train, y_train, learning_rate=cur_lr, reg=cur_reg,num_iters=1600, verbose=True)

        # Training
        y_pred = smx.predict(X_train)
        train_accuracy = np.mean(np.equal(y_train, y_pred, dtype=float))

        # Validation
        y_pred = smx.predict(X_val)
        val_accuracy = np.mean(np.equal(y_val, y_pred, dtype=float))

        results[(cur_lr, cur_reg)] = (train_accuracy, val_accuracy)

        if val_accuracy > best_val:
            best_val = val_accuracy
            best_softmax = smx
            best_lr = cur_lr
            best_reg = cur_reg

# best results printed (from my run I got these values, also shown below)
# best_val      0.369
# best_lr       2e-07
# best_reg      37500.0
# best validation during cross-validation: 0.369000

print ("\n****\ndone comparing\n****")
print ("\nbest_val    ",best_val)
print ("\nbest_lr     ",best_lr)
print ("\nbest_reg    ",best_reg)
print ("\nbest Softmax model saved :)\n")

# 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)
```

```
strating loop ! This may take a while
```

```
iteration 0 / 1600: loss 395.338651
iteration 100 / 1600: loss 239.013424
iteration 200 / 1600: loss 145.153142
iteration 300 / 1600: loss 88.584269
iteration 400 / 1600: loss 54.297126
iteration 500 / 1600: loss 33.750134
iteration 600 / 1600: loss 21.178037
iteration 700 / 1600: loss 13.619623
iteration 800 / 1600: loss 9.068348
iteration 900 / 1600: loss 6.222027
iteration 1000 / 1600: loss 4.629920
iteration 1100 / 1600: loss 3.584675
iteration 1200 / 1600: loss 2.901236
iteration 1300 / 1600: loss 2.573211
iteration 1400 / 1600: loss 2.359519
iteration 1500 / 1600: loss 2.220186
iteration 0 / 1600: loss 483.373433
iteration 100 / 1600: loss 258.306914
iteration 200 / 1600: loss 138.624373
iteration 300 / 1600: loss 74.983773
iteration 400 / 1600: loss 40.859433
iteration 500 / 1600: loss 22.810248
iteration 600 / 1600: loss 13.058675
iteration 700 / 1600: loss 7.871121
iteration 800 / 1600: loss 5.160931
iteration 900 / 1600: loss 3.731637
iteration 1000 / 1600: loss 2.904479
iteration 1100 / 1600: loss 2.528300
iteration 1200 / 1600: loss 2.357585
iteration 1300 / 1600: loss 2.151734
iteration 1400 / 1600: loss 2.088765
iteration 1500 / 1600: loss 2.137322
iteration 0 / 1600: loss 579.780248
iteration 100 / 1600: loss 273.887898
iteration 200 / 1600: loss 129.679805
iteration 300 / 1600: loss 62.085737
iteration 400 / 1600: loss 30.328453
iteration 500 / 1600: loss 15.368541
iteration 600 / 1600: loss 8.306926
iteration 700 / 1600: loss 5.036042
iteration 800 / 1600: loss 3.443271
iteration 900 / 1600: loss 2.765635
iteration 1000 / 1600: loss 2.381657
iteration 1100 / 1600: loss 2.221403
iteration 1200 / 1600: loss 2.158739
iteration 1300 / 1600: loss 2.107679
iteration 1400 / 1600: loss 2.090771
iteration 1500 / 1600: loss 2.005758
iteration 0 / 1600: loss 676.237765
iteration 100 / 1600: loss 281.259409
iteration 200 / 1600: loss 118.043836
iteration 300 / 1600: loss 50.237832
iteration 400 / 1600: loss 21.981392
iteration 500 / 1600: loss 10.333832
iteration 600 / 1600: loss 5.522526
iteration 700 / 1600: loss 3.505078
iteration 800 / 1600: loss 2.708618
iteration 900 / 1600: loss 2.326545
iteration 1000 / 1600: loss 2.183757
iteration 1100 / 1600: loss 2.118260
iteration 1200 / 1600: loss 2.073882
iteration 1300 / 1600: loss 2.154066
iteration 1400 / 1600: loss 2.130312
iteration 1500 / 1600: loss 2.086252
iteration 0 / 1600: loss 779.139996
iteration 100 / 1600: loss 285.764254
iteration 200 / 1600: loss 105.966744
iteration 300 / 1600: loss 40.032313
iteration 400 / 1600: loss 15.991679
iteration 500 / 1600: loss 7.159524
iteration 600 / 1600: loss 3.982132
iteration 700 / 1600: loss 2.763481
iteration 800 / 1600: loss 2.374113
iteration 900 / 1600: loss 2.187468
iteration 1000 / 1600: loss 2.035153
```

```
iteration 1100 / 1600: loss 2.076702
iteration 1200 / 1600: loss 2.093679
iteration 1300 / 1600: loss 2.052496
iteration 1400 / 1600: loss 2.027612
iteration 1500 / 1600: loss 2.073934
iteration 0 / 1600: loss 393.947795
iteration 100 / 1600: loss 144.404964
iteration 200 / 1600: loss 53.867630
iteration 300 / 1600: loss 20.969404
iteration 400 / 1600: loss 8.943559
iteration 500 / 1600: loss 4.610170
iteration 600 / 1600: loss 2.999448
iteration 700 / 1600: loss 2.413722
iteration 800 / 1600: loss 2.198936
iteration 900 / 1600: loss 2.045505
iteration 1000 / 1600: loss 2.128481
iteration 1100 / 1600: loss 2.025738
iteration 1200 / 1600: loss 2.001638
iteration 1300 / 1600: loss 1.998919
iteration 1400 / 1600: loss 2.026524
iteration 1500 / 1600: loss 1.985756
iteration 0 / 1600: loss 485.872845
iteration 100 / 1600: loss 138.615257
iteration 200 / 1600: loss 40.922240
iteration 300 / 1600: loss 13.111542
iteration 400 / 1600: loss 5.197297
iteration 500 / 1600: loss 2.962451
iteration 600 / 1600: loss 2.318063
iteration 700 / 1600: loss 2.106254
iteration 800 / 1600: loss 2.024339
iteration 900 / 1600: loss 2.021715
iteration 1000 / 1600: loss 2.029740
iteration 1100 / 1600: loss 2.038453
iteration 1200 / 1600: loss 2.004126
iteration 1300 / 1600: loss 2.012466
iteration 1400 / 1600: loss 2.067397
iteration 1500 / 1600: loss 2.009766
iteration 0 / 1600: loss 584.910059
iteration 100 / 1600: loss 130.436418
iteration 200 / 1600: loss 30.386132
iteration 300 / 1600: loss 8.309932
iteration 400 / 1600: loss 3.510046
iteration 500 / 1600: loss 2.363582
iteration 600 / 1600: loss 2.059584
iteration 700 / 1600: loss 2.058715
iteration 800 / 1600: loss 2.120440
iteration 900 / 1600: loss 2.004193
iteration 1000 / 1600: loss 2.065732
iteration 1100 / 1600: loss 2.081511
iteration 1200 / 1600: loss 2.126301
iteration 1300 / 1600: loss 2.084860
iteration 1400 / 1600: loss 2.040726
iteration 1500 / 1600: loss 2.037489
iteration 0 / 1600: loss 679.223072
iteration 100 / 1600: loss 117.773963
iteration 200 / 1600: loss 21.949637
iteration 300 / 1600: loss 5.554129
iteration 400 / 1600: loss 2.731647
iteration 500 / 1600: loss 2.194066
iteration 600 / 1600: loss 2.101777
iteration 700 / 1600: loss 2.060093
iteration 800 / 1600: loss 2.116886
iteration 900 / 1600: loss 2.131866
iteration 1000 / 1600: loss 2.095893
iteration 1100 / 1600: loss 1.998760
iteration 1200 / 1600: loss 2.070126
iteration 1300 / 1600: loss 2.084637
iteration 1400 / 1600: loss 2.100373
iteration 1500 / 1600: loss 2.082595
iteration 0 / 1600: loss 776.143791
iteration 100 / 1600: loss 104.971835
iteration 200 / 1600: loss 15.826922
iteration 300 / 1600: loss 3.949945
iteration 400 / 1600: loss 2.331116
iteration 500 / 1600: loss 2.104409
iteration 600 / 1600: loss 2.101880
iteration 700 / 1600: loss 2.074515
```

```
iteration 800 / 1600: loss 2.070585
iteration 900 / 1600: loss 2.063432
iteration 1000 / 1600: loss 2.075900
iteration 1100 / 1600: loss 2.102091
iteration 1200 / 1600: loss 2.070459
iteration 1300 / 1600: loss 2.114980
iteration 1400 / 1600: loss 2.060056
iteration 1500 / 1600: loss 2.061295
iteration 0 / 1600: loss 392.213372
iteration 100 / 1600: loss 87.401826
iteration 200 / 1600: loss 20.878887
iteration 300 / 1600: loss 6.253804
iteration 400 / 1600: loss 2.950693
iteration 500 / 1600: loss 2.250532
iteration 600 / 1600: loss 2.074851
iteration 700 / 1600: loss 1.955204
iteration 800 / 1600: loss 2.078445
iteration 900 / 1600: loss 2.026476
iteration 1000 / 1600: loss 2.097814
iteration 1100 / 1600: loss 2.009314
iteration 1200 / 1600: loss 2.042749
iteration 1300 / 1600: loss 1.985079
iteration 1400 / 1600: loss 2.093901
iteration 1500 / 1600: loss 1.980142
iteration 0 / 1600: loss 487.035348
iteration 100 / 1600: loss 75.012151
iteration 200 / 1600: loss 13.051090
iteration 300 / 1600: loss 3.755341
iteration 400 / 1600: loss 2.274365
iteration 500 / 1600: loss 2.119243
iteration 600 / 1600: loss 2.035407
iteration 700 / 1600: loss 1.976939
iteration 800 / 1600: loss 2.002256
iteration 900 / 1600: loss 2.047338
iteration 1000 / 1600: loss 2.018707
iteration 1100 / 1600: loss 1.987311
iteration 1200 / 1600: loss 2.029764
iteration 1300 / 1600: loss 2.022479
iteration 1400 / 1600: loss 2.012726
iteration 1500 / 1600: loss 2.088762
iteration 0 / 1600: loss 592.191118
iteration 100 / 1600: loss 62.837014
iteration 200 / 1600: loss 8.357127
iteration 300 / 1600: loss 2.651312
iteration 400 / 1600: loss 2.133271
iteration 500 / 1600: loss 2.060363
iteration 600 / 1600: loss 2.019282
iteration 700 / 1600: loss 2.063272
iteration 800 / 1600: loss 2.059574
iteration 900 / 1600: loss 2.080019
iteration 1000 / 1600: loss 2.067819
iteration 1100 / 1600: loss 2.062447
iteration 1200 / 1600: loss 2.127812
iteration 1300 / 1600: loss 2.069283
iteration 1400 / 1600: loss 2.073945
iteration 1500 / 1600: loss 2.097180
iteration 0 / 1600: loss 682.894030
iteration 100 / 1600: loss 50.113322
iteration 200 / 1600: loss 5.479543
iteration 300 / 1600: loss 2.319687
iteration 400 / 1600: loss 2.109819
iteration 500 / 1600: loss 2.115183
iteration 600 / 1600: loss 2.094736
iteration 700 / 1600: loss 2.031515
iteration 800 / 1600: loss 2.090897
iteration 900 / 1600: loss 2.068028
iteration 1000 / 1600: loss 2.095652
iteration 1100 / 1600: loss 2.018039
iteration 1200 / 1600: loss 2.108649
iteration 1300 / 1600: loss 2.106023
iteration 1400 / 1600: loss 2.058843
iteration 1500 / 1600: loss 2.131564
iteration 0 / 1600: loss 768.836897
iteration 100 / 1600: loss 38.977949
iteration 200 / 1600: loss 3.831379
iteration 300 / 1600: loss 2.208679
iteration 400 / 1600: loss 2.084622
```

```
iteration 500 / 1600: loss 2.043094
iteration 600 / 1600: loss 2.067673
iteration 700 / 1600: loss 2.102761
iteration 800 / 1600: loss 2.071990
iteration 900 / 1600: loss 2.015166
iteration 1000 / 1600: loss 2.139711
iteration 1100 / 1600: loss 2.051476
iteration 1200 / 1600: loss 2.087506
iteration 1300 / 1600: loss 2.089236
iteration 1400 / 1600: loss 2.095975
iteration 1500 / 1600: loss 2.090713
iteration 0 / 1600: loss 393.560552
iteration 100 / 1600: loss 53.639321
iteration 200 / 1600: loss 8.917883
iteration 300 / 1600: loss 3.008684
iteration 400 / 1600: loss 2.085048
iteration 500 / 1600: loss 2.025506
iteration 600 / 1600: loss 1.989612
iteration 700 / 1600: loss 2.002224
iteration 800 / 1600: loss 2.049280
iteration 900 / 1600: loss 2.091533
iteration 1000 / 1600: loss 1.980554
iteration 1100 / 1600: loss 2.045975
iteration 1200 / 1600: loss 2.059084
iteration 1300 / 1600: loss 2.019877
iteration 1400 / 1600: loss 1.999089
iteration 1500 / 1600: loss 2.026256
iteration 0 / 1600: loss 483.153852
iteration 100 / 1600: loss 40.453720
iteration 200 / 1600: loss 5.069005
iteration 300 / 1600: loss 2.314994
iteration 400 / 1600: loss 2.037598
iteration 500 / 1600: loss 2.021294
iteration 600 / 1600: loss 2.079169
iteration 700 / 1600: loss 2.008426
iteration 800 / 1600: loss 2.034038
iteration 900 / 1600: loss 2.022563
iteration 1000 / 1600: loss 1.978705
iteration 1100 / 1600: loss 1.965484
iteration 1200 / 1600: loss 2.090652
iteration 1300 / 1600: loss 2.120992
iteration 1400 / 1600: loss 2.055286
iteration 1500 / 1600: loss 2.112776
iteration 0 / 1600: loss 588.580416
iteration 100 / 1600: loss 30.181105
iteration 200 / 1600: loss 3.405222
iteration 300 / 1600: loss 2.071667
iteration 400 / 1600: loss 2.038566
iteration 500 / 1600: loss 2.056223
iteration 600 / 1600: loss 2.098403
iteration 700 / 1600: loss 2.092812
iteration 800 / 1600: loss 2.019171
iteration 900 / 1600: loss 2.046801
iteration 1000 / 1600: loss 2.039109
iteration 1100 / 1600: loss 2.063910
iteration 1200 / 1600: loss 2.048378
iteration 1300 / 1600: loss 2.035152
iteration 1400 / 1600: loss 2.029736
iteration 1500 / 1600: loss 2.006456
iteration 0 / 1600: loss 677.318345
iteration 100 / 1600: loss 21.607735
iteration 200 / 1600: loss 2.632988
iteration 300 / 1600: loss 2.097410
iteration 400 / 1600: loss 2.091945
iteration 500 / 1600: loss 2.078524
iteration 600 / 1600: loss 2.033478
iteration 700 / 1600: loss 2.055746
iteration 800 / 1600: loss 2.059050
iteration 900 / 1600: loss 2.108138
iteration 1000 / 1600: loss 2.134962
iteration 1100 / 1600: loss 2.071335
iteration 1200 / 1600: loss 2.060511
iteration 1300 / 1600: loss 2.052799
iteration 1400 / 1600: loss 2.070894
iteration 1500 / 1600: loss 2.076145
iteration 0 / 1600: loss 766.998740
iteration 100 / 1600: loss 15.371568
```

```
iteration 200 / 1600: loss 2.335287
iteration 300 / 1600: loss 2.105824
iteration 400 / 1600: loss 2.053600
iteration 500 / 1600: loss 2.121679
iteration 600 / 1600: loss 2.080746
iteration 700 / 1600: loss 2.033233
iteration 800 / 1600: loss 2.161257
iteration 900 / 1600: loss 2.108820
iteration 1000 / 1600: loss 2.033108
iteration 1100 / 1600: loss 2.039316
iteration 1200 / 1600: loss 2.047719
iteration 1300 / 1600: loss 2.100564
iteration 1400 / 1600: loss 2.099164
iteration 1500 / 1600: loss 2.057109
iteration 0 / 1600: loss 395.135583
iteration 100 / 1600: loss 33.289213
iteration 200 / 1600: loss 4.608725
iteration 300 / 1600: loss 2.133125
iteration 400 / 1600: loss 2.060672
iteration 500 / 1600: loss 2.021403
iteration 600 / 1600: loss 1.965202
iteration 700 / 1600: loss 2.086646
iteration 800 / 1600: loss 2.081815
iteration 900 / 1600: loss 2.081936
iteration 1000 / 1600: loss 2.063446
iteration 1100 / 1600: loss 2.029845
iteration 1200 / 1600: loss 2.106432
iteration 1300 / 1600: loss 2.055154
iteration 1400 / 1600: loss 1.973042
iteration 1500 / 1600: loss 1.988735
iteration 0 / 1600: loss 484.342771
iteration 100 / 1600: loss 22.396722
iteration 200 / 1600: loss 2.866828
iteration 300 / 1600: loss 2.067929
iteration 400 / 1600: loss 2.005821
iteration 500 / 1600: loss 2.020192
iteration 600 / 1600: loss 2.070837
iteration 700 / 1600: loss 2.014329
iteration 800 / 1600: loss 2.031582
iteration 900 / 1600: loss 2.096414
iteration 1000 / 1600: loss 2.101531
iteration 1100 / 1600: loss 2.070827
iteration 1200 / 1600: loss 2.070121
iteration 1300 / 1600: loss 2.015660
iteration 1400 / 1600: loss 2.055274
iteration 1500 / 1600: loss 2.051645
iteration 0 / 1600: loss 584.876492
iteration 100 / 1600: loss 15.104846
iteration 200 / 1600: loss 2.323504
iteration 300 / 1600: loss 2.065238
iteration 400 / 1600: loss 2.062476
iteration 500 / 1600: loss 2.131096
iteration 600 / 1600: loss 2.019215
iteration 700 / 1600: loss 2.085009
iteration 800 / 1600: loss 2.093049
iteration 900 / 1600: loss 2.051019
iteration 1000 / 1600: loss 2.030579
iteration 1100 / 1600: loss 2.059834
iteration 1200 / 1600: loss 2.070101
iteration 1300 / 1600: loss 2.016472
iteration 1400 / 1600: loss 2.024651
iteration 1500 / 1600: loss 2.099518
iteration 0 / 1600: loss 678.129238
iteration 100 / 1600: loss 10.071663
iteration 200 / 1600: loss 2.191111
iteration 300 / 1600: loss 2.112896
iteration 400 / 1600: loss 2.127791
iteration 500 / 1600: loss 2.071063
iteration 600 / 1600: loss 2.090938
iteration 700 / 1600: loss 2.081788
iteration 800 / 1600: loss 2.079494
iteration 900 / 1600: loss 2.123902
iteration 1000 / 1600: loss 2.021544
iteration 1100 / 1600: loss 2.121497
iteration 1200 / 1600: loss 2.117409
iteration 1300 / 1600: loss 2.056931
iteration 1400 / 1600: loss 2.082827
```

```
iteration 1500 / 1600: loss 2.098174
iteration 0 / 1600: loss 770.064107
iteration 100 / 1600: loss 6.899345
iteration 200 / 1600: loss 2.117444
iteration 300 / 1600: loss 2.114202
iteration 400 / 1600: loss 2.104328
iteration 500 / 1600: loss 2.112189
iteration 600 / 1600: loss 2.158395
iteration 700 / 1600: loss 2.109149
iteration 800 / 1600: loss 2.101754
iteration 900 / 1600: loss 2.107814
iteration 1000 / 1600: loss 2.155126
iteration 1100 / 1600: loss 2.082351
iteration 1200 / 1600: loss 2.097180
iteration 1300 / 1600: loss 2.063837
iteration 1400 / 1600: loss 2.130410
iteration 1500 / 1600: loss 2.072195

****
done comparing
****

best_val     0.369

best_lr      2e-07

best_reg     37500.0

best Softmax model saved :)

lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.352306 val accuracy: 0.366000
lr 1.000000e-07 reg 3.125000e+04 train accuracy: 0.343857 val accuracy: 0.355000
lr 1.000000e-07 reg 3.750000e+04 train accuracy: 0.338367 val accuracy: 0.357000
lr 1.000000e-07 reg 4.375000e+04 train accuracy: 0.334571 val accuracy: 0.341000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.332224 val accuracy: 0.349000
lr 2.000000e-07 reg 2.500000e+04 train accuracy: 0.353408 val accuracy: 0.366000
lr 2.000000e-07 reg 3.125000e+04 train accuracy: 0.342327 val accuracy: 0.360000
lr 2.000000e-07 reg 3.750000e+04 train accuracy: 0.342551 val accuracy: 0.369000
lr 2.000000e-07 reg 4.375000e+04 train accuracy: 0.330980 val accuracy: 0.342000
lr 2.000000e-07 reg 5.000000e+04 train accuracy: 0.330571 val accuracy: 0.343000
lr 3.000000e-07 reg 2.500000e+04 train accuracy: 0.351469 val accuracy: 0.369000
lr 3.000000e-07 reg 3.125000e+04 train accuracy: 0.341388 val accuracy: 0.361000
lr 3.000000e-07 reg 3.750000e+04 train accuracy: 0.341694 val accuracy: 0.348000
lr 3.000000e-07 reg 4.375000e+04 train accuracy: 0.336020 val accuracy: 0.341000
lr 3.000000e-07 reg 5.000000e+04 train accuracy: 0.329735 val accuracy: 0.347000
lr 4.000000e-07 reg 2.500000e+04 train accuracy: 0.349306 val accuracy: 0.366000
lr 4.000000e-07 reg 3.125000e+04 train accuracy: 0.334796 val accuracy: 0.347000
lr 4.000000e-07 reg 3.750000e+04 train accuracy: 0.337837 val accuracy: 0.345000
lr 4.000000e-07 reg 4.375000e+04 train accuracy: 0.332041 val accuracy: 0.344000
lr 4.000000e-07 reg 5.000000e+04 train accuracy: 0.324918 val accuracy: 0.334000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.344061 val accuracy: 0.363000
lr 5.000000e-07 reg 3.125000e+04 train accuracy: 0.344367 val accuracy: 0.346000
lr 5.000000e-07 reg 3.750000e+04 train accuracy: 0.341918 val accuracy: 0.355000
lr 5.000000e-07 reg 4.375000e+04 train accuracy: 0.331347 val accuracy: 0.343000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.324061 val accuracy: 0.339000
best validation accuracy achieved during cross-validation: 0.369000
```

In [14]:

```
# 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: 0.360000
```

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: The SVM does not care much about the margin differences, whilst for the Softmax classifier, this improves the probability, as these are based on the magnitudes of the scores. From notes, compared to the Softmax classifier, the SVM is a more local objective, which could be thought of either as a bug or a feature. The Softmax classifier is never fully happy with the scores it produces: the correct class could always have a higher probability and the incorrect classes always a lower probability and the loss would always get better. An additional data point could thus impact the loss significantly or insignificantly depending on the sample size.

These weights(class templates) below are similar to the weights I got from the linear_SVM, which is of no suprise, as the same dataset was used for training. They look much better and more refined than the SVM, from a human eye perspective

In [15]:

```
# 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 [ ]:

```
```

In [ ]:

```
```