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 (http://vision.stanford.edu/teaching/cs231n/assignments.html) 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 [1]:

```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt
from __future__ import print_function

%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
n
%load_ext autoreload
%autoreload 2
```

In [2]:

```
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)
```

```
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 [4]:

```
# 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, softmax_loss_vectoriz
ed
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)
#loss, grad = softmax_loss_vectorized(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: 2.337193
sanity check: 2.302585
```

Inline Question 1:

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

Your answer: There are 10 classes, and we initialized the weight matrix randomly s.t. each of the 10 classes should receive approximately the same score. Then softmax(xi) = e^score_yi / sum e^score_j ~= 1/10; softmax loss is average of -log(softmax(xi)).

In [5]:

```
# 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: 0.567849 analytic: 0.567849, relative error: 1.022578e-07
numerical: 1.100079 analytic: 1.100079, relative error: 2.135127e-08
numerical: -2.462063 analytic: -2.462063, relative error: 2.582816e-
09
numerical: 2.206644 analytic: 2.206644, relative error: 1.724082e-08
numerical: 3.106253 analytic: 3.106253, relative error: 1.736512e-08
numerical: -1.265471 analytic: -1.265471, relative error: 1.625567e-
08
numerical: -0.510681 analytic: -0.510681, relative error: 6.360931e-
08
numerical: 1.432267 analytic: 1.432267, relative error: 5.457080e-08
numerical: 0.851139 analytic: 0.851139, relative error: 8.969135e-10
numerical: 1.133305 analytic: 1.133304, relative error: 4.096189e-08
numerical: 1.635839 analytic: 1.635840, relative error: 2.319247e-08
numerical: 0.500283 analytic: 0.500283, relative error: 1.672209e-08
numerical: 1.967745 analytic: 1.967745, relative error: 3.375964e-08
numerical: -1.071423 analytic: -1.071423, relative error: 3.551398e-
08
numerical: -1.091325 analytic: -1.091325, relative error: 5.064061e-
08
numerical: -2.049836 analytic: -2.049836, relative error: 3.272740e-
08
numerical: 0.887626 analytic: 0.887626, relative error: 5.251666e-09
numerical: 2.146585 analytic: 2.146585, relative error: 2.256034e-08
numerical: 2.850963 analytic: 2.850963, relative error: 9.449120e-09
numerical: -0.237759 analytic: -0.237759, relative error: 8.487378e-
08
```

In [6]:

```
# Now that we have a naive implementation of the softmax loss function and its g
radient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version s
hould 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, 0.00
0005)
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: 2.337193e+00 computed in 0.071997s
vectorized loss: 2.337193e+00 computed in 0.022640s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [15]:

```
# 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
best_params = None
learning_rates = [5e-8, 1e-7, 5e-7]
regularization_strengths = [1e4, 2.5e4, 5e4]

for l in learning_rates:
    for r in regularization_strengths:
        soft = Softmax()
        curr_loss = soft.train(X_train, y_train, learning_rate=l, reg=r,
                               num_iters=1500, verbose=True)
        y_train_pred = soft.predict(X_train)
        y_train_acc = np.mean(y_train == y_train_pred)
        y_val_pred = soft.predict(X_val)
        y_val_acc = np.mean(y_val == y_val_pred)
        results[(l, r)] = (y_train_acc, y_val_acc)
        if y_val_acc > best_val:
            best_softmax = soft
            best_val = y_val_acc
            best_params = (l, r)

################################################################################
#                              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
)
print(best_params)
```

```
iteration 0 / 1500: loss 314.378890
iteration 100 / 1500: loss 256.343739
iteration 200 / 1500: loss 210.005278
iteration 300 / 1500: loss 171.735479
iteration 400 / 1500: loss 140.718794
iteration 500 / 1500: loss 115.403647
iteration 600 / 1500: loss 94.755554
iteration 700 / 1500: loss 77.763083
iteration 800 / 1500: loss 63.903976
iteration 900 / 1500: loss 52.885643
```

```
iteration 1000 / 1500: loss 43.416895
iteration 1100 / 1500: loss 35.993682
iteration 1200 / 1500: loss 29.782689
iteration 1300 / 1500: loss 24.691988
iteration 1400 / 1500: loss 20.494863
iteration 0 / 1500: loss 784.983024
iteration 100 / 1500: loss 474.939836
iteration 200 / 1500: loss 288.524443
iteration 300 / 1500: loss 175.513011
iteration 400 / 1500: loss 106.944714
iteration 500 / 1500: loss 65.541919
iteration 600 / 1500: loss 40.464141
iteration 700 / 1500: loss 25.454078
iteration 800 / 1500: loss 16.140550
iteration 900 / 1500: loss 10.592509
iteration 1000 / 1500: loss 7.199630
iteration 1100 / 1500: loss 5.176507
iteration 1200 / 1500: loss 3.956560
iteration 1300 / 1500: loss 3.203671
iteration 1400 / 1500: loss 2.765133
iteration 0 / 1500: loss 1540.514962
iteration 100 / 1500: loss 565.337485
iteration 200 / 1500: loss 208.378826
iteration 300 / 1500: loss 77.735027
iteration 400 / 1500: loss 29.853285
iteration 500 / 1500: loss 12.270621
iteration 600 / 1500: loss 5.854505
iteration 700 / 1500: loss 3.491227
iteration 800 / 1500: loss 2.644319
iteration 900 / 1500: loss 2.365326
iteration 1000 / 1500: loss 2.205371
iteration 1100 / 1500: loss 2.186416
iteration 1200 / 1500: loss 2.182300
iteration 1300 / 1500: loss 2.171551
iteration 1400 / 1500: loss 2.079532
iteration 0 / 1500: loss 310.737400
iteration 100 / 1500: loss 206.861984
iteration 200 / 1500: loss 138.510751
iteration 300 / 1500: loss 93.337344
iteration 400 / 1500: loss 63.009215
iteration 500 / 1500: loss 42.864810
iteration 600 / 1500: loss 29.195303
iteration 700 / 1500: loss 20.230915
iteration 800 / 1500: loss 14.230522
iteration 900 / 1500: loss 10.132090
iteration 1000 / 1500: loss 7.520559
iteration 1100 / 1500: loss 5.596569
iteration 1200 / 1500: loss 4.496602
iteration 1300 / 1500: loss 3.632488
iteration 1400 / 1500: loss 3.178314
iteration 0 / 1500: loss 771.884038
iteration 100 / 1500: loss 283.140275
iteration 200 / 1500: loss 104.975657
```

```
iteration 300 / 1500: loss 39.794251
iteration 400 / 1500: loss 15.861718
iteration 500 / 1500: loss 7.124673
iteration 600 / 1500: loss 3.922526
iteration 700 / 1500: loss 2.728889
iteration 800 / 1500: loss 2.305585
iteration 900 / 1500: loss 2.191911
iteration 1000 / 1500: loss 2.108062
iteration 1100 / 1500: loss 2.040261
iteration 1200 / 1500: loss 2.122895
iteration 1300 / 1500: loss 2.086892
iteration 1400 / 1500: loss 2.070165
iteration 0 / 1500: loss 1569.807557
iteration 100 / 1500: loss 211.542831
iteration 200 / 1500: loss 30.125867
iteration 300 / 1500: loss 5.896750
iteration 400 / 1500: loss 2.632760
iteration 500 / 1500: loss 2.200729
iteration 600 / 1500: loss 2.157806
iteration 700 / 1500: loss 2.166693
iteration 800 / 1500: loss 2.115097
iteration 900 / 1500: loss 2.131973
iteration 1000 / 1500: loss 2.153319
iteration 1100 / 1500: loss 2.122642
iteration 1200 / 1500: loss 2.152296
iteration 1300 / 1500: loss 2.166793
iteration 1400 / 1500: loss 2.107457
iteration 0 / 1500: loss 312.407420
iteration 100 / 1500: loss 42.712501
iteration 200 / 1500: loss 7.417511
iteration 300 / 1500: loss 2.716093
iteration 400 / 1500: loss 2.092821
iteration 500 / 1500: loss 2.029881
iteration 600 / 1500: loss 2.016174
iteration 700 / 1500: loss 2.095447
iteration 800 / 1500: loss 1.936520
iteration 900 / 1500: loss 2.098716
iteration 1000 / 1500: loss 2.050053
iteration 1100 / 1500: loss 1.981984
iteration 1200 / 1500: loss 2.037626
iteration 1300 / 1500: loss 1.978531
iteration 1400 / 1500: loss 1.960792
iteration 0 / 1500: loss 783.468668
iteration 100 / 1500: loss 6.962788
iteration 200 / 1500: loss 2.155492
iteration 300 / 1500: loss 2.089235
iteration 400 / 1500: loss 2.045556
iteration 500 / 1500: loss 2.068149
iteration 600 / 1500: loss 2.066446
iteration 700 / 1500: loss 2.076635
iteration 800 / 1500: loss 2.057205
iteration 900 / 1500: loss 2.138281
iteration 1000 / 1500: loss 2.095712
```

```
iteration 1100 / 1500: loss 2.062105
iteration 1200 / 1500: loss 2.111504
iteration 1300 / 1500: loss 2.067248
iteration 1400 / 1500: loss 2.052028
iteration 0 / 1500: loss 1557.437666
iteration 100 / 1500: loss 2.220458
iteration 200 / 1500: loss 2.141399
iteration 300 / 1500: loss 2.189879
iteration 400 / 1500: loss 2.147661
iteration 500 / 1500: loss 2.144514
iteration 600 / 1500: loss 2.096290
iteration 700 / 1500: loss 2.167907
iteration 800 / 1500: loss 2.161713
iteration 900 / 1500: loss 2.168270
iteration 1000 / 1500: loss 2.180228
iteration 1100 / 1500: loss 2.179335
iteration 1200 / 1500: loss 2.163926
iteration 1300 / 1500: loss 2.086831
iteration 1400 / 1500: loss 2.185150
lr 5.000000e-08 reg 1.000000e+04 train accuracy: 0.297510 val accura
cy: 0.303000
lr 5.000000e-08 reg 2.500000e+04 train accuracy: 0.326735 val accura
cy: 0.331000
lr 5.000000e-08 reg 5.000000e+04 train accuracy: 0.311776 val accura
cy: 0.327000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.353204 val accura
cy: 0.366000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.329041 val accura
cy: 0.337000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.307714 val accura
cy: 0.329000
lr 5.000000e-07 reg 1.000000e+04 train accuracy: 0.350714 val accura
cy: 0.365000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.325653 val accura
cy: 0.341000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.301816 val accura
cy: 0.315000
best validation accuracy achieved during cross-validation: 0.366000
(1e-07, 10000.0)
```

In [16]:

```
# 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: 0.354000
```

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: With SVM loss, as long as the new image is predicted with the correct label over all other labels by greater than the margin, there will be an additional loss of 0. With softmax, as we predict the correct class with more confidence the additional loss approaches 0 (e^score_yi --> inf, log(e^score_yi / sum e^score_j --> 1) but never truly reaches.

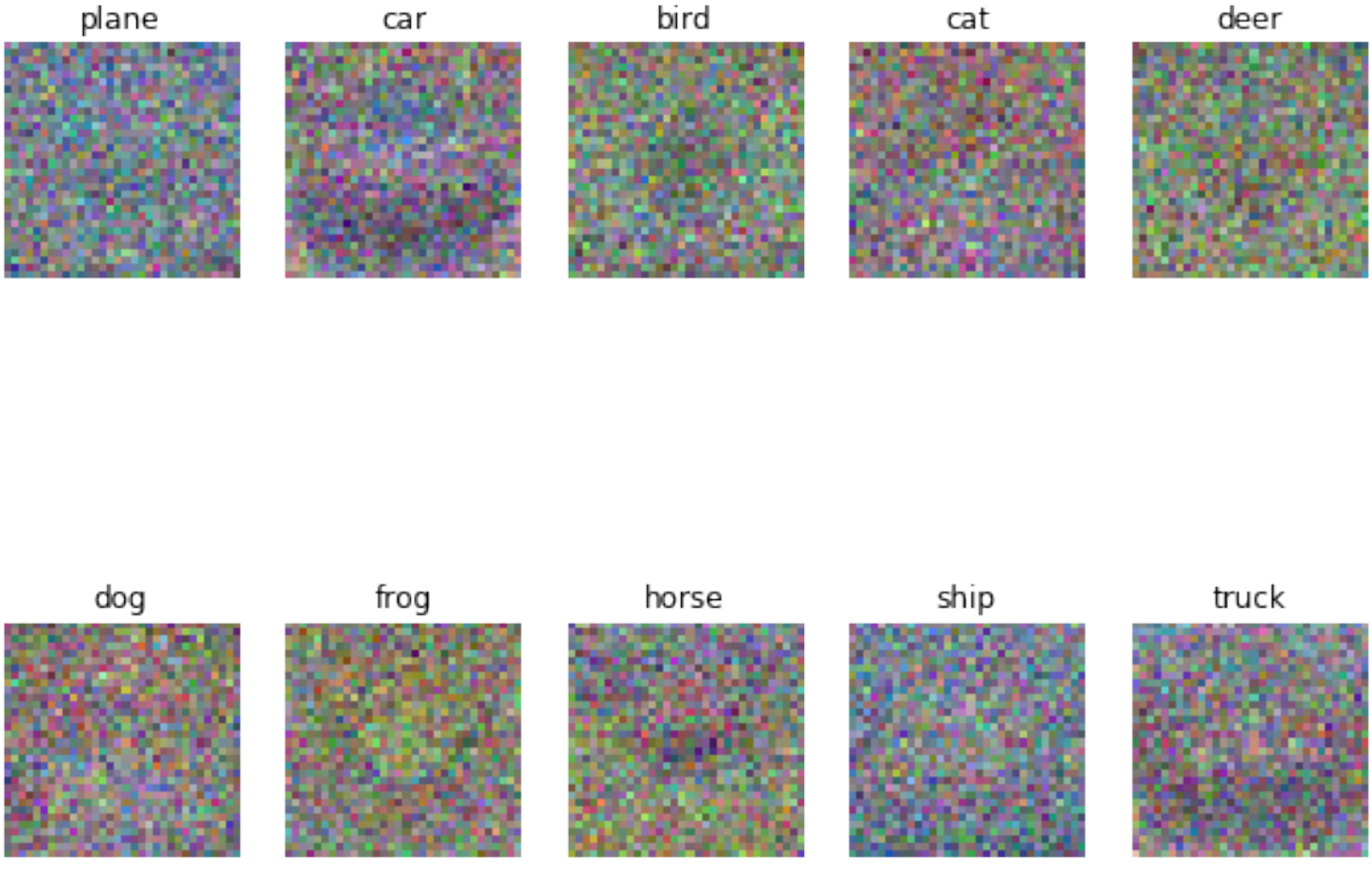
In [17]:

```
# 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])
```


plane
car
bird
cat
deer
dog
frog
horse
ship
truck