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

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
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: 2.418624
sanity check: 2.302585
```

Inline Question 1

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

YourAnswer: *Fill this in*

Because in this case W is a (3073, 10) matrix. There are 10 classes, so the exponential of the correct score divided by the sum of the exponentials of all scores of 10 classes is approximately 0.1 and the final loss should be close to -log(0.1)

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: -1.479105 analytic: -1.479105, relative error: 4.890182e-09
numerical: 1.586017 analytic: 1.586017, relative error: 2.929528e-08
```

```
numerical: 1.905107 analytic: 1.905107, relative error: 2.770694e-08
numerical: -1.349112 analytic: -1.349113, relative error: 2.421673e-08
numerical: -0.049770 analytic: -0.049770, relative error: 2.810320e-07
numerical: -0.658285 analytic: -0.658285, relative error: 1.967038e-09
numerical: 4.038681 analytic: 4.038681, relative error: 1.313990e-08
numerical: -2.009209 analytic: -2.009209, relative error: 1.187446e-08
numerical: 0.855620 analytic: 0.855620, relative error: 1.601767e-08
numerical: 0.301801 analytic: 0.301801, relative error: 6.025370e-08
numerical: 3.564181 analytic: 3.564181, relative error: 1.145265e-08
numerical: -0.006918 analytic: -0.006918, relative error: 1.383980e-06
numerical: 1.061391 analytic: 1.061391, relative error: 3.158600e-08
numerical: 2.230215 analytic: 2.230215, relative error: 1.051158e-08
numerical: -0.047113 analytic: -0.047113, relative error: 3.201433e-07
numerical: -0.803367 analytic: -0.803367, relative error: 1.867133e-09
numerical: -0.554875 analytic: -0.554875, relative error: 4.480239e-08
numerical: 2.945169 analytic: 2.945169, relative error: 2.223602e-08
numerical: -2.765114 analytic: -2.765114, relative error: 8.125281e-09
numerical: 2.225074 analytic: 2.225074, relative error: 1.806905e-09
```

In [6]:

```
# 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: 2.418624e+00 computed in 0.110465s
vectorized loss: 2.418624e+00 computed in 0.025973s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [10]:

```
# 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, 2e-7, 3e-7, 4e-7, 5e-7]
regularization_strengths = [2.5e4, 5e4, 7.5e4, 1e3]

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

for rate in learning_rates:
    for strength in regularization_strengths:
        # Initialize a new svm every time, and train the data
        sftm = Softmax()
        sftm.train(X_train, y_train, learning_rate=rate, reg=strength, num_iters=5000)

        y_train_pred = sftm.predict(X_train)
        training_accuracy = np.mean(y_train == y_train_pred)
```

```
        y_val_pred = sftm.predict(X_val)
        validation_accuracy = np.mean(y_val == y_val_pred)

        # Store every result
        results[(rate, strength)] = (training_accuracy, validation_accuracy)

        # Store the best validation accuracy as the best_val
        if validation_accuracy > best_val:
            best_val = validation_accuracy
            best_softmax = sftm

# *****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 1.000000e+03 train accuracy: 0.363245 val accuracy: 0.364000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.327857 val accuracy: 0.345000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.303612 val accuracy: 0.320000
lr 1.000000e-07 reg 7.500000e+04 train accuracy: 0.297469 val accuracy: 0.310000
lr 2.000000e-07 reg 1.000000e+03 train accuracy: 0.400714 val accuracy: 0.399000
lr 2.000000e-07 reg 2.500000e+04 train accuracy: 0.329388 val accuracy: 0.344000
lr 2.000000e-07 reg 5.000000e+04 train accuracy: 0.303204 val accuracy: 0.320000
lr 2.000000e-07 reg 7.500000e+04 train accuracy: 0.276735 val accuracy: 0.287000
lr 3.000000e-07 reg 1.000000e+03 train accuracy: 0.406000 val accuracy: 0.408000
lr 3.000000e-07 reg 2.500000e+04 train accuracy: 0.331082 val accuracy: 0.338000
lr 3.000000e-07 reg 5.000000e+04 train accuracy: 0.310776 val accuracy: 0.324000
lr 3.000000e-07 reg 7.500000e+04 train accuracy: 0.305102 val accuracy: 0.310000
lr 4.000000e-07 reg 1.000000e+03 train accuracy: 0.403735 val accuracy: 0.404000
lr 4.000000e-07 reg 2.500000e+04 train accuracy: 0.325224 val accuracy: 0.345000
lr 4.000000e-07 reg 5.000000e+04 train accuracy: 0.311224 val accuracy: 0.321000
lr 4.000000e-07 reg 7.500000e+04 train accuracy: 0.292061 val accuracy: 0.301000
lr 5.000000e-07 reg 1.000000e+03 train accuracy: 0.406204 val accuracy: 0.409000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.322837 val accuracy: 0.338000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.298327 val accuracy: 0.314000
lr 5.000000e-07 reg 7.500000e+04 train accuracy: 0.297367 val accuracy: 0.311000
best validation accuracy achieved during cross-validation: 0.409000
```

In [11]:

```
# 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: 0.392000
```

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.

YourAnswer: True

YourExplanation: In the SVM, if the score of the new data point is out of the margin range from the correct class score, the loss wouldn't change. But in the Softmax loss, since the calculation formula of loss involves all scores, when a new datapoint is added the loss of Softmax will definitely change.

In [12]:

```
# 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])
```