Erik Arriaga
SID: 015707183

Softmax exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission.

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 [26]:
```
from __future__ import print_function

import random
import numpy as np
from cecs551.data_utils import load_CIFAR10
import matplotlib.pyplot as plt


%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plo
ts
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
```

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

In [27]:
```
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 p
repare
    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 = 'cecs551/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_de
v


# 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_CIF
AR10_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 **cecs551/classifiers/softmax.py**.

In [28]:
```
# First implement the naive softmax loss function with nested loops.
# Open the file cecs551/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cecs551.classifiers.softmax import softmax_loss_naive
import time

# Generate a random softmax weight matrix and use it to compute the lo
ss.
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: 2.363409
sanity check: 2.302585
```

Inline Question 1:

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

Your answer: The reason we expect our loss to be close to -log(0.1) is becuase every class has an equal chance of being chosen therefore you have 1/10= 0.1.

In [29]:
```
# Complete the implementation of softmax_loss_naive and implement a (n
aive)
# 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 cecs551.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: 3.399946 analytic: 3.399946, relative error: 2.378064e-08
numerical: -0.217765 analytic: -0.217765, relative error: 1.932822e-
07
numerical: 0.112358 analytic: 0.112358, relative error: 3.229609e-07
numerical: 2.274531 analytic: 2.274531, relative error: 5.190869e-09
numerical: -2.625791 analytic: -2.625791, relative error: 2.831508e-
09
numerical: 0.378763 analytic: 0.378763, relative error: 1.270786e-07
numerical: 0.542166 analytic: 0.542166, relative error: 2.494940e-08
numerical: -0.381455 analytic: -0.381455, relative error: 1.158111e-
08
numerical: 0.452245 analytic: 0.452244, relative error: 1.120004e-07
numerical: 1.273186 analytic: 1.273186, relative error: 1.153588e-08
numerical: 1.583221 analytic: 1.583221, relative error: 4.334338e-08
numerical: 2.149624 analytic: 2.149624, relative error: 1.579062e-09
numerical: -0.822888 analytic: -0.822888, relative error: 1.469103e-
08
numerical: -0.487875 analytic: -0.487875, relative error: 1.212396e-
07
numerical: 2.264784 analytic: 2.264784, relative error: 2.036152e-08
numerical: -0.802336 analytic: -0.802336, relative error: 1.251531e-
08
numerical: -0.705462 analytic: -0.705462, relative error: 7.348280e-
08
numerical: -0.449860 analytic: -0.449860, relative error: 1.044429e-
07
numerical: -2.141373 analytic: -2.141373, relative error: 9.323800e-
10
numerical: 1.254065 analytic: 1.254065, relative error: 3.867278e-08
```

In [30]:
```
# 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 cecs551.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='fr
o')
print('Loss difference: %f' % np.abs(loss_naive - loss_vectorized))
print('Gradient difference: %f' % grad_difference)
```

```
naive loss: 2.363409e+00 computed in 0.076567s
vectorized loss: 2.363409e+00 computed in 0.007196s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [40]:
```
# Use the validation set to tune hyperparameters (regularization stren
gth 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 cecs551.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 s
trength. #
# This should be identical to the validation that you did for the SVM;
save    #
# the best trained softmax classifer in best_softmax.
#
######################################################################
##########
for i in learning_rates:
    for j in regularization_strengths:
        softmax=Softmax()
        softmax.train(X_train, y_train, learning_rate=i, reg=j,num_ite
rs=4000, verbose=False)

        y_train_pred=softmax.predict(X_train)
        train_accuracy = np.mean(y_train_pred == y_train)

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

        results[(i,j)] = (train_accuracy, val_accuracy)

        if best_val < val_accuracy:
            best_val = val_accuracy
            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 1.000000e-07 reg 2.500000e+04 train accuracy: 0.329714 val accura
cy: 0.344000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.307510 val accura
cy: 0.321000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.327878 val accura
cy: 0.352000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.298000 val accura
cy: 0.312000
best validation accuracy achieved during cross-validation: 0.352000
```

In [41]:

```
# 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_accu
racy, ))
```

```
softmax on raw pixels final test set accuracy: 0.347000
```

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

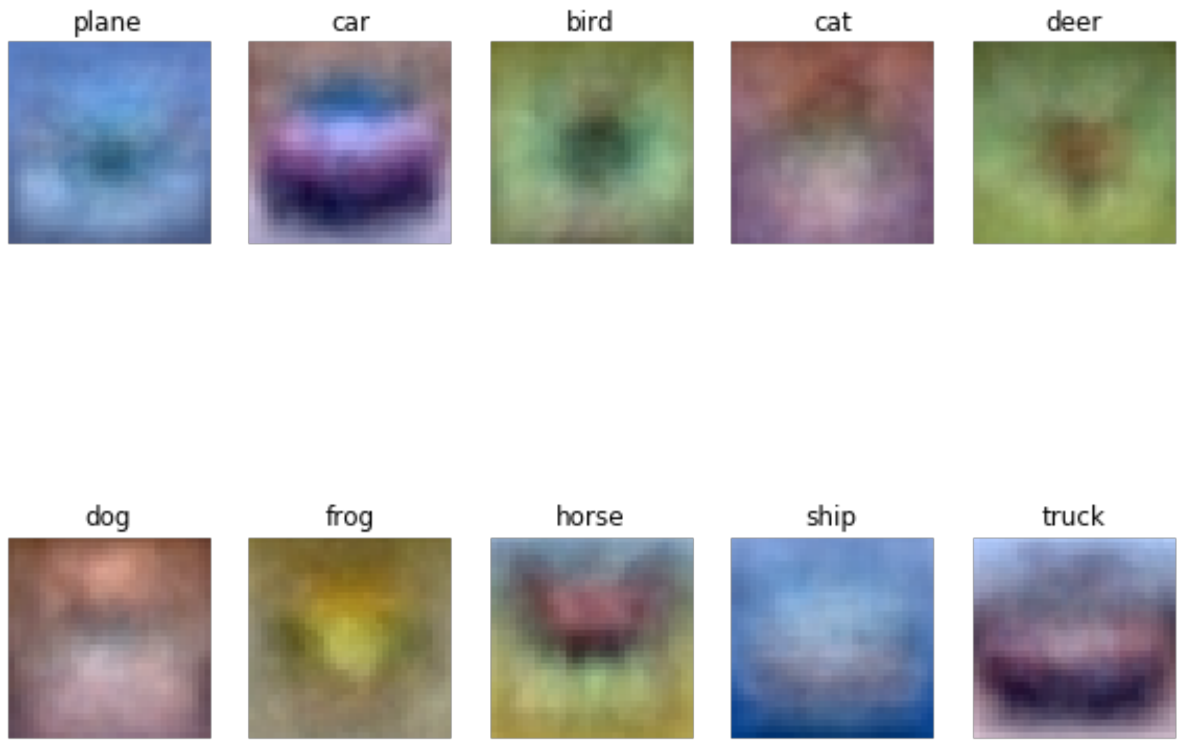
Your explanation: SVM you can add a point and as long as its smaller by a margine it will not care about it and it will remain unchanged. On the other hand softmax will compute the differences which would give higher probablity to the correct classes and lower probablility to the incorrect ones.

In [42]:
```
# 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', 'hors
e', '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 [ ]: