svm

October 21, 2019

b. Q2:Training a Support Vector Machine 의 결과를 작성한 코와 함께 출력하세요.

1 Multiclass Support Vector Machine 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.

In this exercise you will:

- implement a fully-vectorized loss function for the SVM
- implement the fully-vectorized expression for its analytic gradient
- check your implementation using 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]:
```
# Run some setup code for this notebook.
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

# This is a bit of magic to make matplotlib figures appear inline in the
# notebook rather than in a new window.
%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'

# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

1.1 CIFAR-10 Data Loading and Preprocessing

In [2]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'
```

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

X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

# As a sanity check, we print out the size of the training and test data.
print('Training data shape: ', X_train.shape)
print('Training labels shape: ', y_train.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

In [3]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'tru
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```

In [4]:
```
# Split the data into train, val, and test sets. In addition we will
# create a small development set as a subset of the training data;
# we can use this for development so our code runs faster.
num_training = 49000
num_validation = 1000
num_test = 1000
num_dev = 500

# Our validation set will be num_validation points from the original
# training set.
mask = range(num_training, num_training + num_validation)
X_val = X_train[mask]
y_val = y_train[mask]

# Our training set will be the first num_train points from the original
# training set.
mask = range(num_training)
X_train = X_train[mask]
y_train = y_train[mask]

# We will also make a development set, which is a small subset of
# the training set.
mask = np.random.choice(num_training, num_dev, replace=False)
X_dev = X_train[mask]
```

```
        y_dev = y_train[mask]

        # We use the first num_test points of the original test set as our
        # test set.
        mask = range(num_test)
        X_test = X_test[mask]
        y_test = y_test[mask]

        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)
```

```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

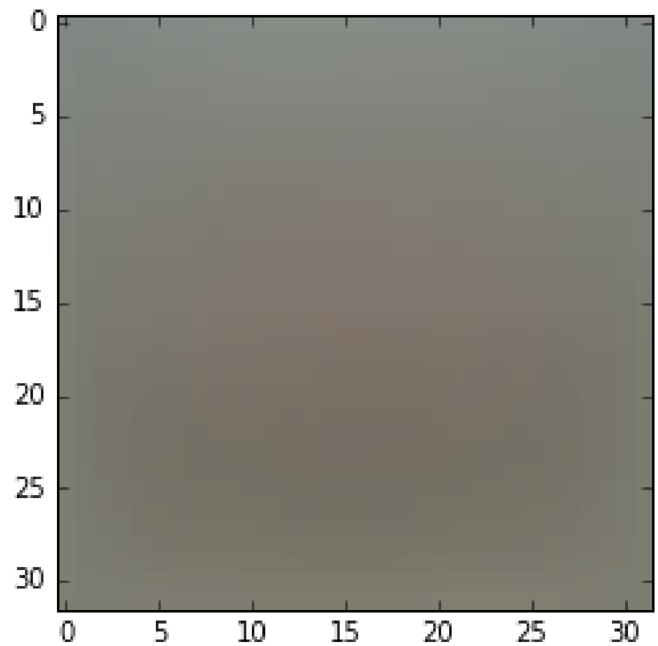
```
In [5]: # Preprocessing: reshape the image data into rows
        X_train = np.reshape(X_train, (X_train.shape[0], -1))
        X_val = np.reshape(X_val, (X_val.shape[0], -1))
        X_test = np.reshape(X_test, (X_test.shape[0], -1))
        X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

        # As a sanity check, print out the shapes of the data
        print('Training data shape: ', X_train.shape)
        print('Validation data shape: ', X_val.shape)
        print('Test data shape: ', X_test.shape)
        print('dev data shape: ', X_dev.shape)
```

```
Training data shape:  (49000, 3072)
Validation data shape:  (1000, 3072)
Test data shape:  (1000, 3072)
dev data shape:  (500, 3072)
```

```
In [6]: # Preprocessing: subtract the mean image
        # first: compute the image mean based on the training data
        mean_image = np.mean(X_train, axis=0)
        print(mean_image[:10]) # print a few of the elements
        plt.figure(figsize=(4,4))
        plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean image
        plt.show()
```

```python
# second: subtract the mean image from train and test data
X_train -= mean_image
X_val -= mean_image
X_test -= mean_image
X_dev -= mean_image

# third: append the bias dimension of ones (i.e. bias trick) so that our SVM
# only has to worry about optimizing a single weight matrix W.
# reference https://docs.scipy.org/doc/numpy/reference/generated/numpy.hstack.html
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))])

print(X_train.shape, X_val.shape, X_test.shape, X_dev.shape)
```

```
[130.64189796 135.98173469 132.47391837 130.05569388 135.34804082
 131.75402041 130.96055102 136.14328571 132.47636735 131.48467347]
```




```
(49000, 3073) (1000, 3073) (1000, 3073) (500, 3073)
```

1.2 SVM Classifier

Your code for this section will all be written inside cs231n/classifiers/linear_svm.py.

As you can see, we have prefilled the function `compute_loss_naive` which uses for loops to evaluate the multiclass SVM loss function.

```
In [7]: # Evaluate the naive implementation of the loss we provided for you:
        from cs231n.classifiers.linear_svm import svm_loss_naive
        import time

        # generate a random SVM weight matrix of small numbers
        W = np.random.randn(3073, 10) * 0.0001

        loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.000005)
        print('loss: %f' % (loss, ))
```

```
loss: 8.362342
```

The `grad` returned from the function above is right now all zero. Derive and implement the gradient for the SVM cost function and implement it inline inside the function `svm_loss_naive`. You will find it helpful to interleave your new code inside the existing function.

To check that you have correctly implemented the gradient correctly, you can numerically estimate the gradient of the loss function and compare the numeric estimate to the gradient that you computed. We have provided code that does this for you:

```
In [8]: # Once you've implemented the gradient, recompute it with the code below
        # and gradient check it with the function we provided for you

        # Compute the loss and its gradient at W.
        loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.0)

        # Numerically compute the gradient along several randomly chosen dimensions, and
        # compare them with your analytically computed gradient. The numbers should match
        # almost exactly along all dimensions.
        from cs231n.gradient_check import grad_check_sparse
        f = lambda w: svm_loss_naive(w, X_dev, y_dev, 0.0)[0]
        grad_numerical = grad_check_sparse(f, W, grad)

        # do the gradient check once again with regularization turned on
        # you didn't forget the regularization gradient did you?
        loss, grad = svm_loss_naive(W, X_dev, y_dev, 5e1)
        f = lambda w: svm_loss_naive(w, X_dev, y_dev, 5e1)[0]
        grad_numerical = grad_check_sparse(f, W, grad)
```

```
numerical: -6.550013 analytic: -6.528892, relative error: 1.614859e-03
numerical: 2.318938 analytic: 2.318938, relative error: 3.212504e-13
numerical: 2.569903 analytic: 2.569903, relative error: 1.150959e-10
numerical: -8.137274 analytic: -8.137274, relative error: 4.142099e-11
numerical: 4.121999 analytic: 4.121999, relative error: 2.494313e-12
numerical: 9.210009 analytic: 9.210009, relative error: 8.214437e-13
numerical: -11.998376 analytic: -12.017013, relative error: 7.760275e-04
```

```
numerical: 31.725840 analytic: 31.725840, relative error: 8.454059e-12
numerical: 22.252359 analytic: 22.252359, relative error: 1.624927e-11
numerical: 20.521757 analytic: 20.521757, relative error: 2.433288e-11
numerical: 27.805344 analytic: 27.805344, relative error: 2.480864e-12
numerical: -3.912636 analytic: -3.912636, relative error: 1.011090e-10
numerical: 11.122959 analytic: 11.112988, relative error: 4.483946e-04
numerical: -14.739596 analytic: -14.739596, relative error: 1.613462e-11
numerical: 27.441613 analytic: 27.441613, relative error: 4.679558e-12
numerical: 13.003308 analytic: 13.003308, relative error: 1.857323e-12
numerical: -14.653793 analytic: -14.653793, relative error: 7.110740e-12
numerical: -34.090059 analytic: -34.090059, relative error: 1.183503e-11
numerical: -40.651603 analytic: -40.651603, relative error: 8.462816e-12
numerical: 17.046818 analytic: 17.046818, relative error: 3.477922e-11
```

Inline Question 1

It is possible that once in a while a dimension in the gradcheck will not match exactly. What could such a discrepancy be caused by? Is it a reason for concern? What is a simple example in one dimension where a gradient check could fail? How would change the margin affect of the frequency of this happening? Hint: the SVM loss function is not strictly speaking differentiable

$\textit{Your Answer}$: fill this in.
gradcheck에서 a dimensiond이 정확하게 match 하지 않는 경우가 가능하다. 특히 예를 들어, max(x,y)와 같은 함수에서 미분이 가능하지 못한 부분은 x=y일 경우로 이러한 미분 불가능한 지점에서 차이로 인해 gradient check의 실패의 원인이 되기도 한다. 예를 들어, 만약에 x= -1e-8의 max(0,x)=0 이지만 analytical gradient는 0이지만 $f(x+h) = max(0, x+h) = c > 0$ 인 겨우에는 numerical gradient는 다를 수가 있다. 이러한 문제를 피하기 위해서는 이러한 빈도수를 줄이기 위해 작은 갯 수의 data를 고려하여 gradient를 체크해야 한다.

```
In [9]: # Next implement the function svm_loss_vectorized; for now only compute the loss;
        # we will implement the gradient in a moment.
        tic = time.time()
        loss_naive, grad_naive = svm_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.linear_svm import svm_loss_vectorized
        tic = time.time()
        loss_vectorized, _ = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
        toc = time.time()
        print('Vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

        # The losses should match but your vectorized implementation should be much faster.
        print('difference: %f' % (loss_naive - loss_vectorized))
```

```
Naive loss: 8.362342e+00 computed in 0.316934s
Vectorized loss: 8.362342e+00 computed in 0.010588s
difference: -0.000000
```

```
In [10]: # Complete the implementation of svm_loss_vectorized, and compute the gradient
         # of the loss function in a vectorized way.

         # The naive implementation and the vectorized implementation should match, but
         # the vectorized version should still be much faster.
         tic = time.time()
         _, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
         toc = time.time()
         print('Naive loss and gradient: computed in %fs' % (toc - tic))

         tic = time.time()
         _, grad_vectorized = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
         toc = time.time()
         print('Vectorized loss and gradient: computed in %fs' % (toc - tic))

         # The loss is a single number, so it is easy to compare the values computed
         # by the two implementations. The gradient on the other hand is a matrix, so
         # we use the Frobenius norm to compare them.
         difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
         print('difference: %f' % difference)
```

```
Naive loss and gradient: computed in 0.379391s
Vectorized loss and gradient: computed in 0.018631s
difference: 0.000000
```

1.2.1 Stochastic Gradient Descent

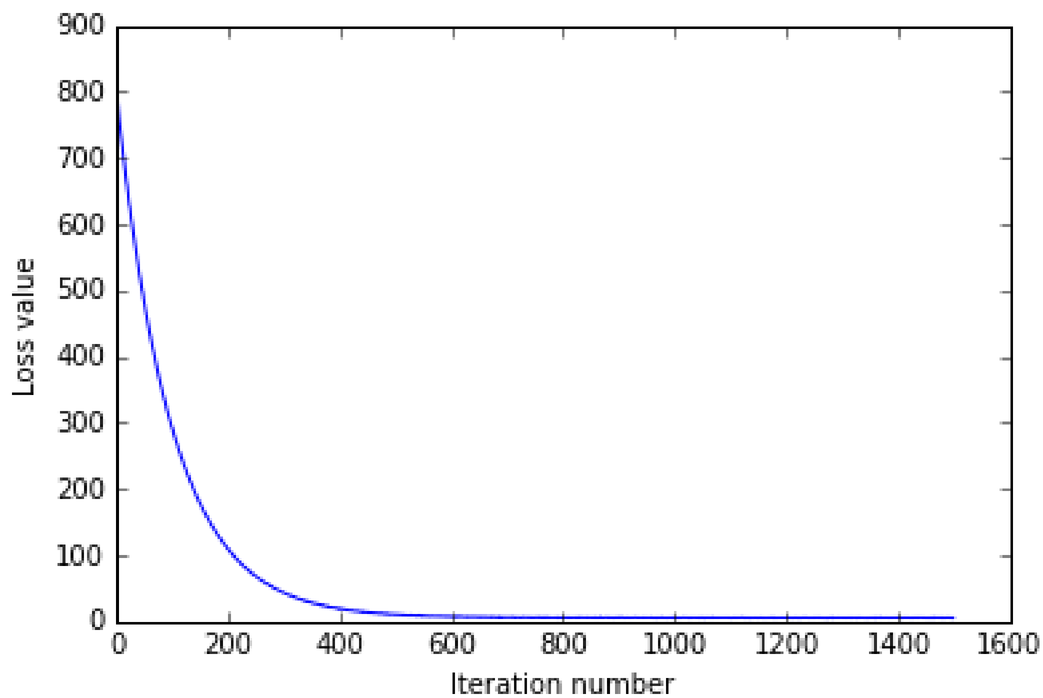
We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss.

```
In [11]: # In the file linear_classifier.py, implement SGD in the function
         # LinearClassifier.train() and then run it with the code below.
         from cs231n.classifiers import LinearSVM
         svm = LinearSVM()
         tic = time.time()
         loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                               num_iters=1500, verbose=True)
         toc = time.time()
         print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 801.423068
iteration 100 / 1500: loss 292.740408
iteration 200 / 1500: loss 109.731829
iteration 300 / 1500: loss 43.046063
iteration 400 / 1500: loss 18.877367
iteration 500 / 1500: loss 10.699088
iteration 600 / 1500: loss 6.656162
iteration 700 / 1500: loss 6.067055
```

```
iteration 800 / 1500: loss 5.879955
iteration 900 / 1500: loss 5.988081
iteration 1000 / 1500: loss 4.962403
iteration 1100 / 1500: loss 4.990319
iteration 1200 / 1500: loss 5.401752
iteration 1300 / 1500: loss 5.378302
iteration 1400 / 1500: loss 5.685536
That took 10.014534s
```

In [12]:
```python
# A useful debugging strategy is to plot the loss as a function of
# iteration number:
plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```



In [13]:
```python
# Write the LinearSVM.predict function and evaluate the performance on both the
# training and validation set
y_train_pred = svm.predict(X_train)
print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
y_val_pred = svm.predict(X_val)
print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

```
training accuracy: 0.376531
validation accuracy: 0.370000
```

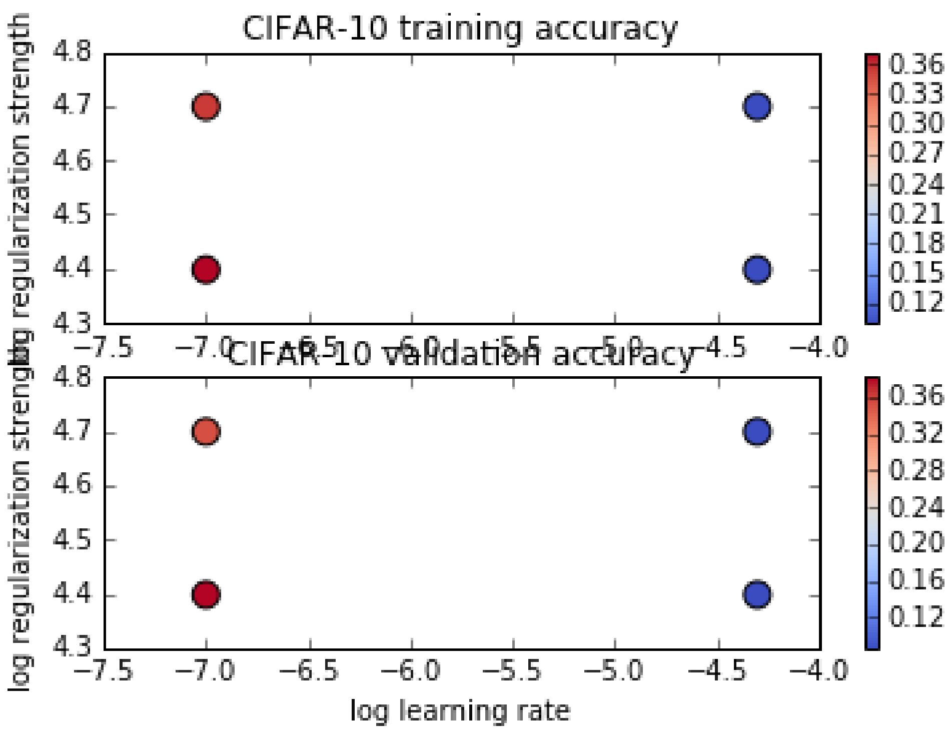
In [14]:
```python
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.39 on the validation set.

#Note: you may see runtime/overflow warnings during hyper-parameter search.
# This may be caused by extreme values, and is not a bug.

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

# results is dictionary mapping tuples of the form
# (learning_rate, regularization_strength) to tuples of the form
# (training_accuracy, validation_accuracy). The accuracy is simply the fraction
# of data points that are correctly classified.
results = {}
best_val = -1   # The highest validation accuracy that we have seen so far.
best_svm = None # The LinearSVM object that achieved the highest validation rate.

################################################################################
# TODO:                                                                        #
# Write code that chooses the best hyperparameters by tuning on the validation #
# set. For each combination of hyperparameters, train a linear SVM on the      #
# training set, compute its accuracy on the training and validation sets, and  #
# store these numbers in the results dictionary. In addition, store the best   #
# validation accuracy in best_val and the LinearSVM object that achieves this  #
# accuracy in best_svm.                                                        #
#                                                                              #
# Hint: You should use a small value for num_iters as you develop your         #
# validation code so that the SVMs don't take much time to train; once you are #
# confident that your validation code works, you should rerun the validation   #
# code with a larger value for num_iters.                                      #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

grid_searches = [(lr, rg) for lr in learning_rates for rg in regularization_strengths

for idx, val in enumerate(grid_searches):
    svm = LinearSVM()

    loss_hist = svm.train(X_train, y_train, learning_rate=val[0], reg=val[1], num_ite

    y_train_pred = svm.predict(X_train)
    # numpy mean reference https://docs.scipy.org/doc/numpy/reference/generated/numpy
```

```
            acc_train = np.mean(y_train == y_train_pred)

            y_val_pred = svm.predict(X_val)
            acc_val = np.mean(y_val == y_val_pred)

            results[(val[0], val[1])] = (acc_train, acc_val)

            if acc_val > best_val:
                best_val = acc_val
                best_svm = svm

        # *****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)
```

```
/home/hyunyoung2/labs/machine_learning_assignment/assignment1_cs231/cs231n/classifiers/linear_s
  regularization = reg*np.sum(np.square(W))
/home/hyunyoung2/.local/lib/python3.5/site-packages/numpy/core/_methods.py:32: RuntimeWarning:
  return umr_sum(a, axis, dtype, out, keepdims)
/home/hyunyoung2/labs/machine_learning_assignment/assignment1_cs231/cs231n/classifiers/linear_s
  regularization = reg*np.sum(np.square(W))
/home/hyunyoung2/labs/machine_learning_assignment/assignment1_cs231/cs231n/classifiers/linear_s
  margin = np.maximum(0, scores-correct_scores+1)
/home/hyunyoung2/labs/machine_learning_assignment/assignment1_cs231/cs231n/classifiers/linear_s
  margin = np.maximum(0, scores-correct_scores+1)
/home/hyunyoung2/labs/machine_learning_assignment/assignment1_cs231/cs231n/classifiers/linear_s
  X_mask[margin>0] = 1
/home/hyunyoung2/labs/machine_learning_assignment/assignment1_cs231/cs231n/classifiers/linear_s
  dW += 2 * reg * W
/home/hyunyoung2/labs/machine_learning_assignment/assignment1_cs231/cs231n/classifiers/linear_c
  self.W += - learning_rate * grad


lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.372633 val accuracy: 0.382000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.355918 val accuracy: 0.354000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.382000
```

```
In [15]: # 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, cmap=plt.cm.coolwarm)
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, cmap=plt.cm.coolwarm)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.show()
```



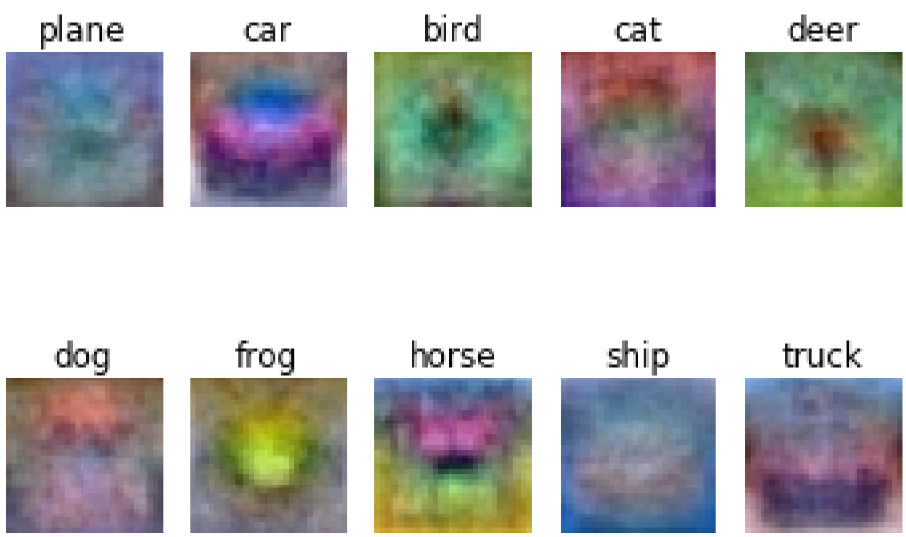

```
In [16]: # Evaluate the best svm on test set
```

```
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.375000
```

In [17]:

```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, these may
# or may not be nice to look at.
w = best_svm.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', 'tru
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])
```



Inline question 2

Describe what your visualized SVM weights look like, and offer a brief explanation for why they look they way that they do.

Your Answer : fill this in

SVM의 weight는 각 class의 image의 조합으로 보편적인 template을 보여준다.