CS280 Fall 2022 Assignment 1
Part A

ML Background

October 16, 2022

Name: Dai ZiJia

Student ID:2022233158

1. MLE (5 points)

Given a dataset $\mathcal{D} = \{x_1, \cdots, x_n\}$. Let $p_{emp}(x)$ be the empirical distribution, i.e., $p_{emp}(x) = \frac{1}{n}\sum_{i=1}^{n}\delta(x, x_i)$ where $\delta(x, a)$ is the Dirac delta function[1] centered at a. Assume $q(x|\theta)$ be some probabilistic model.

- Show that $\arg\min_q KL(p_{emp}||q)$ is obtained by $q(x) = q(x; \hat{\theta})$, where $\hat{\theta}$ is the Maximum Likelihood Estimator and $KL(p||q) = \int p(x)(\log p(x) - \log q(x))dx$ is the KL divergence.

Proof.

$$\begin{aligned}
\arg\min_{\theta} KL(p_{emp}||q) &= arg\min_{\theta} \int (p_{emp}\log p_{emp} - p_{emp}\log q)dx \\
&= arg\min_{\theta}(\int (p_{emp}\log p_{emp})dx - \int (p_{emp}\log q)dx) \\
&= arg\min_{\theta} - \int (p_{emp}\log q)dx \\
&= arg\max_{\theta} \int (p_{emp}\log q)dx \\
&= arg\max_{\theta} \int \frac{1}{n}\sum [\log q(x_i|\theta)\delta(x, x_i)]dx \\
&= arg\max_{\theta} \frac{1}{n}\sum \log q(x_i|\theta) \int \delta(x, x_i)dx \\
&= arg\max_{\theta} \frac{1}{n}\sum \log q(x_i|\theta)
\end{aligned}$$

[1] https://en.wikipedia.org/wiki/Dirac_delta_function

2. Gradient descent for fitting GMM (10 points)

Consider the Gaussian mixture model

$$p(\mathbf{x}|\theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

where $\pi_j \geq 0, \sum_{j=1}^{K} \pi_j = 1$. (Assume $\mathbf{x}, \boldsymbol{\mu}_k \in \mathbb{R}^d, \boldsymbol{\Sigma}_k \in \mathbb{R}^{d\times d}$)

Define the log likelihood as

$$l(\theta) = \sum_{n=1}^{N} \log p(\mathbf{x}_n|\theta)$$

Denote the posterior responsibility that cluster k has for datapoint n as follows:

$$r_{nk} := p(z_n = k|\mathbf{x}_n, \theta) = \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{k'} \pi_{k'} \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_{k'}, \boldsymbol{\Sigma}_{k'})}$$

- Show that the gradient of the log-likelihood wrt $\boldsymbol{\mu}_k$ is

$$\frac{d}{d\boldsymbol{\mu}_k} l(\theta) = \sum_n r_{nk} \boldsymbol{\Sigma}_k^{-1} (\mathbf{x}_n - \boldsymbol{\mu}_k)$$

- Derive the gradient of the log-likelihood wrt π_k without considering any constraint on π_k. (bonus 2 points: with constraint $\sum_k \pi_k = 1$.)

$$\begin{aligned}
r_{nk} &= \frac{\pi_k \mathcal{N}\left(\mathbf{x}_n \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\right)}{\sum_{k'} \pi_{k'} \mathcal{N}\left(\mathbf{x}_n \mid \boldsymbol{\mu}_{k'}, \boldsymbol{\Sigma}_{k'}\right)} \\
\sum_n r_{nk} &= \frac{\pi_k \mathcal{N}_{\mathbf{x}}\left(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\right)}{\sum_{k'} \pi_{k'} \mathcal{N}_{\mathbf{x}}\left(\boldsymbol{\mu}_{k'}, \boldsymbol{\Sigma}_{k'}\right)} \\
\frac{dl(\theta)}{d\boldsymbol{\mu}_k} &= \frac{dl(\theta)}{dp(x_n|\theta)} \cdot \frac{dp(x_n|\theta)}{d\mathcal{N}(x_n|\boldsymbol{\mu}_k, \sum_k)} \cdot \frac{d\mathcal{N}(x_n|\boldsymbol{\mu}_k, \sum_k)}{d\boldsymbol{\mu}_k} \\
&= \frac{\pi_k \mathcal{N}_{\mathbf{x}}\left(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\right)}{\sum_{k'} \pi_{k'} \mathcal{N}_{\mathbf{x}}\left(\boldsymbol{\mu}_{k'}, \boldsymbol{\Sigma}_{k'}\right)} \frac{d}{d\boldsymbol{\mu}_k} \ln\left[\pi_k \mathcal{N}_{\mathbf{x}}\left(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\right)\right] \\
&= \frac{\pi_k \mathcal{N}_{\mathbf{x}}\left(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\right)}{\sum_{k'} \pi_{k'} \mathcal{N}_{\mathbf{x}}\left(\boldsymbol{\mu}_{k'}, \boldsymbol{\Sigma}_{k'}\right)} \left[\boldsymbol{\Sigma}_k^{-1}\left(\mathbf{x_n} - \boldsymbol{\mu}_k\right)\right] \\
&= \sum_n r_{nk} \left[\boldsymbol{\Sigma}_k^{-1}\left(\mathbf{x_n} - \boldsymbol{\mu}_k\right)\right]
\end{aligned}$$

Perceptron Learning Algorithm

The perceptron is a simple supervised machine learning algorithm and one of the earliest neural network architectures. It was introduced by Rosenblatt in the late 1950s. A perceptron represents a binary linear classifier that maps a set of training examples (of d dimensional input vectors) onto binary output values using a $d-1$ dimensional hyperplane. But Today, we will implement **Multi-Classes Perceptron Learning Algorithm**
Given:

- dataset $\{(x^i, y^i)\}$, $i \in (1, M)$
- x^i is d dimension vector, $x^i = (x^i_1, \dots x^i_d)$
- y^i is multi-class target varible $y^i \in \{0, 1, 2\}$

A perceptron is trained using gradient descent. The training algorithm has different steps. In the beginning (step 0) the model parameters are initialized. The other steps (see below) are repeated for a specified number of training iterations or until the parameters have converged.

Step0: Initial the weight vector and bias with zeros
Step1: Compute the linear combination of the input features and weight.
$y^i_{pred} = \arg\max_k W_k * x^i + b$
Step2: Compute the gradients for parameters W_k, b. **Derive the parameter update equation Here (5 points)**

##################################

TODO: Derive you answer hear

##################################

If correct, no change!

If wrong: lower score of wrong answer,raise score of right answer

$w_y = w_y - f(x)$

$w^*_y = w^*_y + f(x)$

```
from sklearn import datasets
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import random

np.random.seed(0)
random.seed(0)
```

```
iris = datasets.load_iris()
X = iris.data
print(type(X))
y = iris.target
y = np.array(y)
print('X_Shape:', X.shape)
print('y_Shape:', y.shape)
print('Label Space:', np.unique(y))
```

```
<class 'numpy.ndarray'>
X_Shape: (150, 4)
y_Shape: (150,)
Label Space: [0 1 2]
```

```
## split the training set and test set
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.3,
random_state=0)
print('X_train_Shape:', X_train.shape)
print('X_test_Shape:',  X_test.shape)
print('y_train_Shape:', y_train.shape)
print('y_test_Shape:',  y_train.shape)

print(type(y_train))
```

```
X_train_Shape: (105, 4)
X_test_Shape: (45, 4)
y_train_Shape: (105,)
y_test_Shape: (105,)
<class 'numpy.ndarray'>
```

```
class MultiClsPLA(object):

    ## We recommend to absorb the bias into weight.  W = [w, b]

    def __init__(self, X_train, y_train, X_test, y_test, lr, num_epoch,
weight_dimension, num_cls):
        super(MultiClsPLA, self).__init__()
        self.X_train = X_train
        self.y_train = y_train
        self.X_test = X_test
        self.y_test = y_test
        self.weight = self.initial_weight(weight_dimension, num_cls)
        self.sample_mean = np.mean(self.X_train, 0)
        self.sample_std = np.std(self.X_train, 0)
        self.num_epoch = num_epoch
        self.lr = lr
        self.total_acc_train = []
```

```
        self.total_acc_tst = []

    def initial_weight(self, weight_dimension, num_cls):
        weight = None
        ########################################
        ##  ToDO: Initialize the weight with  ##
        ##  small std and zero mean gaussian  ##
        ########################################
        std = 1e-3
        weight = np.random.normal(loc = 0.0, scale = 0.01, size =
(weight_dimension , num_cls)) #scale?
        weight = std * weight

        return weight

    def data_preprocessing(self, data):
        ####################################
        ##  ToDO: Normlize the data       ##
        ####################################
        norm_data = (data-self.sample_mean)/self.sample_std #?
        return norm_data

    def train_step(self, X_train, y_train, shuffle_idx):
        np.random.shuffle(shuffle_idx)
        X_train = X_train[shuffle_idx]
        y_train = y_train[shuffle_idx]
        train_acc = None
        ############################################
        ## TODO: to implement the training process  ##
        ## and update the weights                  ##
        ############################################
        for i in range(X_train.shape[0]):
            y_pred = np.argmax(np.dot(X_train[i], self.weight))
            if y_pred != y_train[i]:
                self.weight[:, y_pred] -= self.lr * X_train[i]
                self.weight[:, y_train[i]] += self.lr * X_train[i]
        train_acc = (np.argmax(np.dot(X_train, self.weight), axis = 1) ==
y_train).mean()
        return train_acc

    def test_step(self, X_test, y_test):


        X_test = self.data_preprocessing(data=X_test)
        num_sample = X_test.shape[0]
        test_acc = None

        ########################################
        ##  ToDO: Evaluate the test set and   ##
        ##  return the test acc               ##
        ########################################
        test_acc = (y_test == np.argmax(np.dot(X_test, self.weight), axis =
1)).mean()
        return test_acc
```

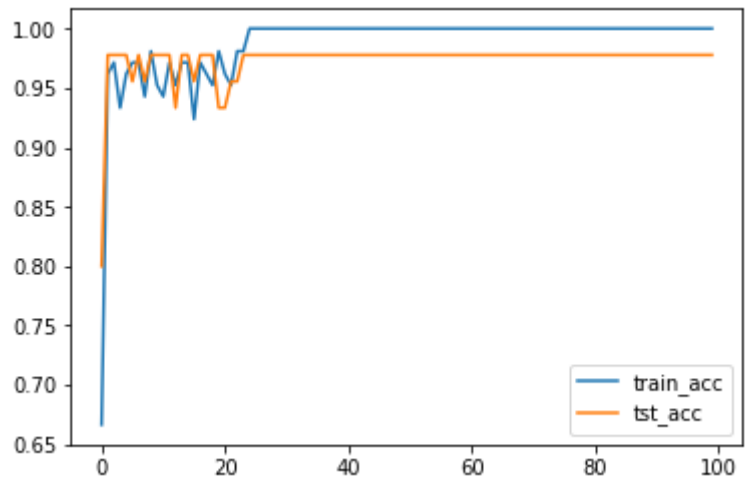
```
    def train(self):

        self.X_train = self.data_preprocessing(data=self.X_train)
        num_sample = self.X_train.shape[0]

        ######################################################
        ### TODO: In order to absorb the bias into weights ###
        ###  we need to modify the input data.             ###
        ###  So You need to transform the input data       ###
        ######################################################

        shuffle_index = np.array(range(0, num_sample))
        for epoch in range(self.num_epoch):
            training_acc = self.train_step(X_train=self.X_train,
y_train=self.y_train, shuffle_idx=shuffle_index)
            tst_acc = self.test_step(X_test=self.X_test,  y_test=self.y_test)
            self.total_acc_train.append(training_acc)
            self.total_acc_tst.append(tst_acc)
            print('epoch:', epoch, 'traing_acc:%.3f'%training_acc,
'tst_acc:%.3f'%tst_acc)

    def vis_acc_curve(self):
        train_acc = np.array(self.total_acc_train)
        tst_acc = np.array(self.total_acc_tst)
        plt.plot(train_acc)
        plt.plot(tst_acc)
        plt.legend(['train_acc', 'tst_acc'])
        plt.show()
```

```
np.random.seed(0)
random.seed(0)
######################################################
### TODO:
### 1. You need to import the model and pass some parameters.
### 2. Then training the model with some epoches.
### 3. Visualize the training acc and test acc verus epoches
epoch = 100
mcp = MultiClsPLA(X_train, y_train, X_test, y_test, 1e-2, epoch, 4, 3)
mcp.train()
mcp.vis_acc_curve()
```

```
epoch: 0 traing_acc:0.667 tst_acc:0.800
epoch: 1 traing_acc:0.962 tst_acc:0.978
epoch: 2 traing_acc:0.971 tst_acc:0.978
epoch: 3 traing_acc:0.933 tst_acc:0.978
epoch: 4 traing_acc:0.962 tst_acc:0.978
epoch: 5 traing_acc:0.971 tst_acc:0.956
epoch: 6 traing_acc:0.971 tst_acc:0.978
epoch: 7 traing_acc:0.943 tst_acc:0.956
epoch: 8 traing_acc:0.981 tst_acc:0.978
epoch: 9 traing_acc:0.952 tst_acc:0.978
epoch: 10 traing_acc:0.943 tst_acc:0.978
```

```
epoch: 11 traing_acc:0.971 tst_acc:0.978
epoch: 12 traing_acc:0.952 tst_acc:0.933
epoch: 13 traing_acc:0.971 tst_acc:0.978
epoch: 14 traing_acc:0.971 tst_acc:0.978
epoch: 15 traing_acc:0.924 tst_acc:0.956
epoch: 16 traing_acc:0.971 tst_acc:0.978
epoch: 17 traing_acc:0.962 tst_acc:0.978
epoch: 18 traing_acc:0.952 tst_acc:0.978
epoch: 19 traing_acc:0.981 tst_acc:0.933
epoch: 20 traing_acc:0.962 tst_acc:0.933
epoch: 21 traing_acc:0.952 tst_acc:0.956
epoch: 22 traing_acc:0.981 tst_acc:0.956
epoch: 23 traing_acc:0.981 tst_acc:0.978
epoch: 24 traing_acc:1.000 tst_acc:0.978
epoch: 25 traing_acc:1.000 tst_acc:0.978
epoch: 26 traing_acc:1.000 tst_acc:0.978
epoch: 27 traing_acc:1.000 tst_acc:0.978
epoch: 28 traing_acc:1.000 tst_acc:0.978
epoch: 29 traing_acc:1.000 tst_acc:0.978
epoch: 30 traing_acc:1.000 tst_acc:0.978
epoch: 31 traing_acc:1.000 tst_acc:0.978
epoch: 32 traing_acc:1.000 tst_acc:0.978
epoch: 33 traing_acc:1.000 tst_acc:0.978
epoch: 34 traing_acc:1.000 tst_acc:0.978
epoch: 35 traing_acc:1.000 tst_acc:0.978
epoch: 36 traing_acc:1.000 tst_acc:0.978
epoch: 37 traing_acc:1.000 tst_acc:0.978
epoch: 38 traing_acc:1.000 tst_acc:0.978
epoch: 39 traing_acc:1.000 tst_acc:0.978
epoch: 40 traing_acc:1.000 tst_acc:0.978
epoch: 41 traing_acc:1.000 tst_acc:0.978
epoch: 42 traing_acc:1.000 tst_acc:0.978
epoch: 43 traing_acc:1.000 tst_acc:0.978
epoch: 44 traing_acc:1.000 tst_acc:0.978
epoch: 45 traing_acc:1.000 tst_acc:0.978
epoch: 46 traing_acc:1.000 tst_acc:0.978
epoch: 47 traing_acc:1.000 tst_acc:0.978
epoch: 48 traing_acc:1.000 tst_acc:0.978
epoch: 49 traing_acc:1.000 tst_acc:0.978
epoch: 50 traing_acc:1.000 tst_acc:0.978
epoch: 51 traing_acc:1.000 tst_acc:0.978
epoch: 52 traing_acc:1.000 tst_acc:0.978
epoch: 53 traing_acc:1.000 tst_acc:0.978
epoch: 54 traing_acc:1.000 tst_acc:0.978
epoch: 55 traing_acc:1.000 tst_acc:0.978
epoch: 56 traing_acc:1.000 tst_acc:0.978
epoch: 57 traing_acc:1.000 tst_acc:0.978
epoch: 58 traing_acc:1.000 tst_acc:0.978
epoch: 59 traing_acc:1.000 tst_acc:0.978
epoch: 60 traing_acc:1.000 tst_acc:0.978
epoch: 61 traing_acc:1.000 tst_acc:0.978
epoch: 62 traing_acc:1.000 tst_acc:0.978
epoch: 63 traing_acc:1.000 tst_acc:0.978
epoch: 64 traing_acc:1.000 tst_acc:0.978
epoch: 65 traing_acc:1.000 tst_acc:0.978
```

```
epoch: 66 traing_acc:1.000 tst_acc:0.978
epoch: 67 traing_acc:1.000 tst_acc:0.978
epoch: 68 traing_acc:1.000 tst_acc:0.978
epoch: 69 traing_acc:1.000 tst_acc:0.978
epoch: 70 traing_acc:1.000 tst_acc:0.978
epoch: 71 traing_acc:1.000 tst_acc:0.978
epoch: 72 traing_acc:1.000 tst_acc:0.978
epoch: 73 traing_acc:1.000 tst_acc:0.978
epoch: 74 traing_acc:1.000 tst_acc:0.978
epoch: 75 traing_acc:1.000 tst_acc:0.978
epoch: 76 traing_acc:1.000 tst_acc:0.978
epoch: 77 traing_acc:1.000 tst_acc:0.978
epoch: 78 traing_acc:1.000 tst_acc:0.978
epoch: 79 traing_acc:1.000 tst_acc:0.978
epoch: 80 traing_acc:1.000 tst_acc:0.978
epoch: 81 traing_acc:1.000 tst_acc:0.978
epoch: 82 traing_acc:1.000 tst_acc:0.978
epoch: 83 traing_acc:1.000 tst_acc:0.978
epoch: 84 traing_acc:1.000 tst_acc:0.978
epoch: 85 traing_acc:1.000 tst_acc:0.978
epoch: 86 traing_acc:1.000 tst_acc:0.978
epoch: 87 traing_acc:1.000 tst_acc:0.978
epoch: 88 traing_acc:1.000 tst_acc:0.978
epoch: 89 traing_acc:1.000 tst_acc:0.978
epoch: 90 traing_acc:1.000 tst_acc:0.978
epoch: 91 traing_acc:1.000 tst_acc:0.978
epoch: 92 traing_acc:1.000 tst_acc:0.978
epoch: 93 traing_acc:1.000 tst_acc:0.978
epoch: 94 traing_acc:1.000 tst_acc:0.978
epoch: 95 traing_acc:1.000 tst_acc:0.978
epoch: 96 traing_acc:1.000 tst_acc:0.978
epoch: 97 traing_acc:1.000 tst_acc:0.978
epoch: 98 traing_acc:1.000 tst_acc:0.978
epoch: 99 traing_acc:1.000 tst_acc:0.978
```



k-Nearest Neighbor (kNN) 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.

The kNN classifier consists of two stages:

- During training, the classifier takes the training data and simply remembers it
- During testing, kNN classifies every test image by comparing to all training images and transfering the labels of the k most similar training examples
- The value of k is cross-validated

In this exercise you will implement these steps and understand the basic Image Classification pipeline, cross-validation, and gain proficiency in writing efficient, vectorized code.

```
# 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
```

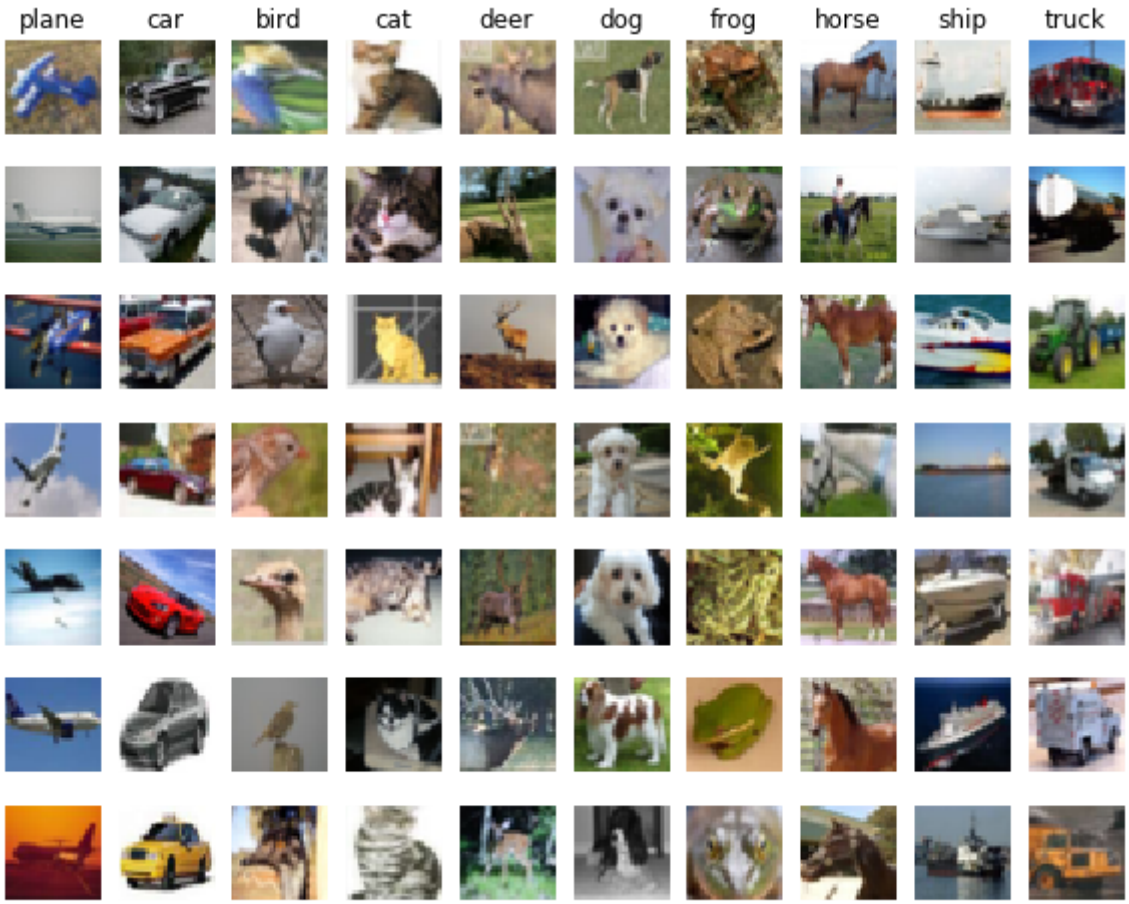
```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

```
# 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 issue)
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)
```

```
Clear previously loaded data.
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

```
# 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', 'truck']
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()
```




```
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

num_test = 500
mask = list(range(num_test))
X_test = X_test[mask]
y_test = y_test[mask]

# Reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
print(X_train.shape, X_test.shape)
```

```
(5000, 3072) (500, 3072)
```

```
from cs231n.classifiers import KNearestNeighbor

# Create a kNN classifier instance.
# Remember that training a kNN classifier is a noop:
# the Classifier simply remembers the data and does no further processing
classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
```

We would now like to classify the test data with the kNN classifier. Recall that we can break down this process into two steps:

1. First we must compute the distances between all test examples and all train examples.
2. Given these distances, for each test example we find the k nearest examples and have them vote for the label

Lets begin with computing the distance matrix between all training and test examples. For example, if there are **Ntr** training examples and **Nte** test examples, this stage should result in a **Nte x Ntr** matrix where each element (i,j) is the distance between the i-th test and j-th train example.

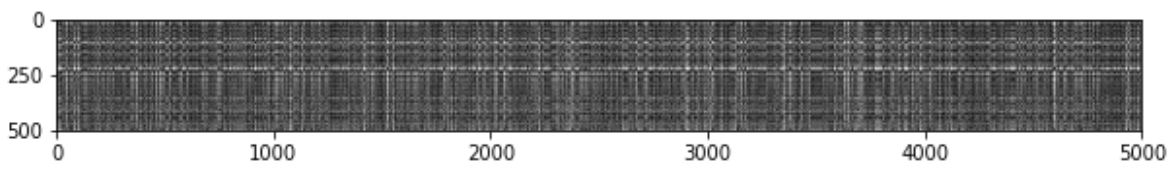
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

First, open `cs231n/classifiers/k_nearest_neighbor.py` and implement the function `compute_distances_two_loops` that uses a (very inefficient) double loop over all pairs of (test, train) examples and computes the distance matrix one element at a time.

```
# Open cs231n/classifiers/k_nearest_neighbor.py and implement
# compute_distances_two_loops.

# Test your implementation:
dists = classifier.compute_distances_two_loops(X_test)
print(dists.shape)
```

```
(500, 5000)
```

```
# We can visualize the distance matrix: each row is a single test example and
# its distances to training examples
plt.imshow(dists, interpolation='none')
plt.show()
```



Inline Question 1

Notice the structured patterns in the distance matrix, where some rows or columns are visible brighter. (Note that with the default color scheme black indicates low distances while white indicates high distances.)

- What in the data is the cause behind the distinctly bright rows?
- What causes the columns?

Your Answer : *1.Because the picture in test is different from most of pictures in train, or it isn't in any type 2.Because the picture in train is different from any other picture in test.*

```
# Now implement the function predict_labels and run the code below:
# We use k = 1 (which is Nearest Neighbor).
y_test_pred = classifier.predict_labels(dists, k=1)

# Compute and print the fraction of correctly predicted examples
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
Got 137 / 500 correct => accuracy: 0.274000
```

You should expect to see approximately `27%` accuracy. Now lets try out a larger `k`, say `k = 5`:

```
y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
Got 139 / 500 correct => accuracy: 0.278000
```

You should expect to see a slightly better performance than with `k = 1`.

Inline Question 2

We can also use other distance metrics such as L1 distance.
For pixel values $p_{ij}^{(k)}$ at location (i, j) of some image I_k,

the mean μ across all pixels over all images is $\mu = \frac{1}{nhw} \sum_{k=1}^{n} \sum_{i=1}^{h} \sum_{j=1}^{w} p_{ij}^{(k)}$
And the pixel-wise mean μ_{ij} across all images is
$\mu_{ij} = \frac{1}{n} \sum_{k=1}^{n} p_{ij}^{(k)}$.
The general standard deviation σ and pixel-wise standard deviation σ_{ij} is defined similarly.

Which of the following preprocessing steps will not change the performance of a Nearest Neighbor classifier that uses L1 distance? Select all that apply.

1. Subtracting the mean μ ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu$.)
2. Subtracting the per pixel mean μ_{ij} ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu_{ij}$.)
3. Subtracting the mean μ and dividing by the standard deviation σ.
4. Subtracting the pixel-wise mean μ_{ij} and dividing by the pixel-wise standard deviation σ_{ij}.
5. Rotating the coordinate axes of the data.

Your Answer : $1, 2, 3$

YourExplanation :
$L1 : d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$

1. Because each pixel has subtracted the same value $|(x_1 - \mu) - (x_2 - \mu)| = |(x_1 - x_2)|$
2. For $|(x_1 - \mu_{ij}) - (x_2 - \mu_{ij})| = |(x_1 - x_2)|$
3. For $|(x_1 - \mu)/\sigma - (x_2 - \mu)/\sigma| = |(x_1 - x_2)|/\sigma$, this won't change closest neighbor.
4. When dividing by the pixel-wise standard deviation, it's scaled different dimensions differently, closest neighbor will change.
5. Not like L2, L1 distance is not invariant to the rotation of the coordinate axes, closest neighbor will change.

```
# Now lets speed up distance matrix computation by using partial vectorization
# with one loop. Implement the function compute_distances_one_loop and run the
# code below:
dists_one = classifier.compute_distances_one_loop(X_test)

# To ensure that our vectorized implementation is correct, we make sure that it
# agrees with the naive implementation. There are many ways to decide whether
# two matrices are similar; one of the simplest is the Frobenius norm. In case
# you haven't seen it before, the Frobenius norm of two matrices is the square
# root of the squared sum of differences of all elements; in other words,
reshape
# the matrices into vectors and compute the Euclidean distance between them.
difference = np.linalg.norm(dists - dists_one, ord='fro')
print('One loop difference was: %f' % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')
```

```
One loop difference was: 0.000000
Good! The distance matrices are the same
```

```
# Now implement the fully vectorized version inside compute_distances_no_loops
# and run the code
dists_two = classifier.compute_distances_no_loops(X_test)

# check that the distance matrix agrees with the one we computed before:
difference = np.linalg.norm(dists - dists_two, ord='fro')
print('No loop difference was: %f' % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')
```

```
No loop difference was: 0.000000
Good! The distance matrices are the same
```

```
# Let's compare how fast the implementations are
def time_function(f, *args):
    """
    Call a function f with args and return the time (in seconds) that it took to
execute.
    """
    import time
    tic = time.time()
    f(*args)
    toc = time.time()
    return toc - tic

two_loop_time = time_function(classifier.compute_distances_two_loops, X_test)
print('Two loop version took %f seconds' % two_loop_time)

one_loop_time = time_function(classifier.compute_distances_one_loop, X_test)
print('One loop version took %f seconds' % one_loop_time)

no_loop_time = time_function(classifier.compute_distances_no_loops, X_test)
print('No loop version took %f seconds' % no_loop_time)

# You should see significantly faster performance with the fully vectorized
implementation!

# NOTE: depending on what machine you're using,
# you might not see a speedup when you go from two loops to one loop,
# and might even see a slow-down.
```

```
Two loop version took 24.256199 seconds
One loop version took 30.744434 seconds
No loop version took 0.141003 seconds
```

Cross-validation

We have implemented the k-Nearest Neighbor classifier but we set the value k = 5 arbitrarily. We will now determine the best value of this hyperparameter with cross-validation.

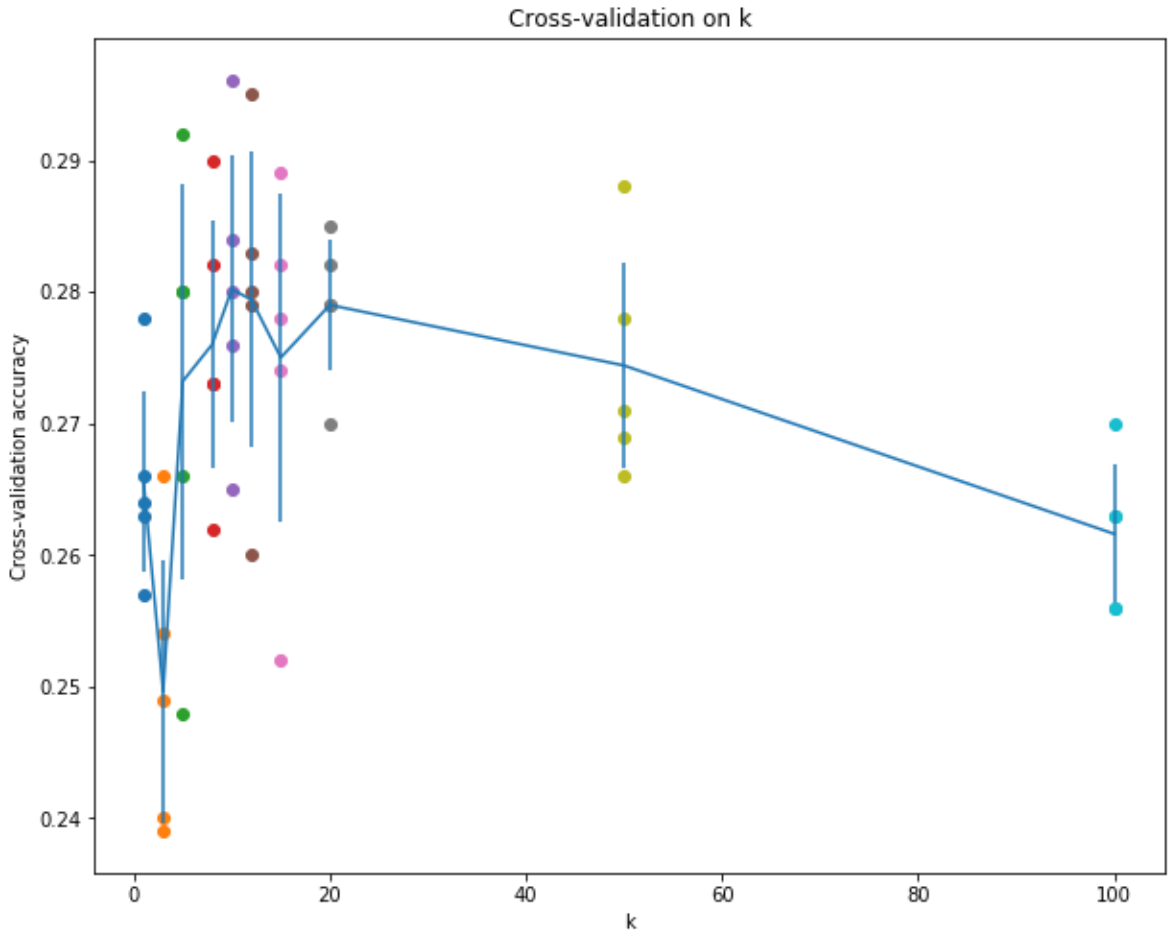
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

X_train_folds = []
y_train_folds = []
################################################################################
# TODO:                                                                        #
# Split up the training data into folds. After splitting, X_train_folds and    #
# y_train_folds should each be lists of length num_folds, where                #
# y_train_folds[i] is the label vector for the points in X_train_folds[i].     #
# Hint: Look up the numpy array_split function.                                #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
X_train_folds = np.array_split(X_train, num_folds)
y_train_folds = np.array_split(y_train, num_folds)

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
# A dictionary holding the accuracies for different values of k that we find
# when running cross-validation. After running cross-validation,
# k_to_accuracies[k] should be a list of length num_folds giving the different
# accuracy values that we found when using that value of k.
k_to_accuracies = {}
################################################################################
# TODO:                                                                        #
# Perform k-fold cross validation to find the best value of k. For each        #
# possible value of k, run the k-nearest-neighbor algorithm num_folds times,   #
# where in each case you use all but one of the folds as training data and the #
# last fold as a validation set. Store the accuracies for all fold and all     #
# values of k in the k_to_accuracies dictionary.                               #
################################################################################
# # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for k in k_choices:
    #classifier = KNearestNeighbor()
    k_to_accuracies[k] = []
    for i in range(num_folds):
        X_t = np.array(X_train_folds[:i] + X_train_folds[i + 1:])
        y_t = np.array(y_train_folds[:i] + y_train_folds[i + 1:])
        X_t = X_t.reshape(-1, X_train.shape[1])
        y_t = y_t.reshape(-1)

        classifier.train(X_t, y_t)
        d = classifier.compute_distances_no_loops(X_train_folds[i])
        y_p = classifier.predict_labels(d , k)
        #num_correct = np.sum(y_p == y_train_folds[i])
        #print(y_p)
        #accuracy = num_correct / y_train_folds[i].shape[0] * 1.0
        accuracy = np.mean(y_p == y_train_folds[i])
        k_to_accuracies[k].append(accuracy)


# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
# Print out the computed accuracies
for k in sorted(k_to_accuracies):
    for accuracy in k_to_accuracies[k]:
        print('k = %d, accuracy = %f' % (k, accuracy))
```

```
k = 1, accuracy = 0.263000
k = 1, accuracy = 0.257000
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
k = 1, accuracy = 0.266000
k = 3, accuracy = 0.239000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.240000
k = 3, accuracy = 0.266000
k = 3, accuracy = 0.254000
k = 5, accuracy = 0.248000
k = 5, accuracy = 0.266000
k = 5, accuracy = 0.280000
k = 5, accuracy = 0.292000
k = 5, accuracy = 0.280000
```

```
k = 8, accuracy = 0.262000
k = 8, accuracy = 0.282000
k = 8, accuracy = 0.273000
k = 8, accuracy = 0.290000
k = 8, accuracy = 0.273000
k = 10, accuracy = 0.265000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.276000
k = 10, accuracy = 0.284000
k = 10, accuracy = 0.280000
k = 12, accuracy = 0.260000
k = 12, accuracy = 0.295000
k = 12, accuracy = 0.279000
k = 12, accuracy = 0.283000
k = 12, accuracy = 0.280000
k = 15, accuracy = 0.252000
k = 15, accuracy = 0.289000
k = 15, accuracy = 0.278000
k = 15, accuracy = 0.282000
k = 15, accuracy = 0.274000
k = 20, accuracy = 0.270000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.282000
k = 20, accuracy = 0.285000
k = 50, accuracy = 0.271000
k = 50, accuracy = 0.288000
k = 50, accuracy = 0.278000
k = 50, accuracy = 0.269000
k = 50, accuracy = 0.266000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.270000
k = 100, accuracy = 0.263000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.263000
```

```
# plot the raw observations
for k in k_choices:
    accuracies = k_to_accuracies[k]
    plt.scatter([k] * len(accuracies), accuracies)

# plot the trend line with error bars that correspond to standard deviation
accuracies_mean = np.array([np.mean(v) for k,v in
sorted(k_to_accuracies.items())])
accuracies_std = np.array([np.std(v) for k,v in
sorted(k_to_accuracies.items())])
plt.errorbar(k_choices, accuracies_mean, yerr=accuracies_std)
plt.title('Cross-validation on k')
plt.xlabel('k')
plt.ylabel('Cross-validation accuracy')
plt.show()
```




```
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 8

classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
y_test_pred = classifier.predict(X_test, k=best_k)

# Compute and display the accuracy
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
Got 137 / 500 correct => accuracy: 0.274000
```

Inline Question 3

Which of the following statements about k-Nearest Neighbor (k-NN) are true in a classification setting, and for all k? Select all that apply.

1. The decision boundary of the k-NN classifier is linear.
2. The training error of a 1-NN will always be lower than that of 5-NN.
3. The test error of a 1-NN will always be lower than that of a 5-NN.

4. The time needed to classify a test example with the k-NN classifier grows with the size of the training set.
5. None of the above.

YourAnswer : 4

YourExplanation :
For 1, it may be a circle. For 2 and 3, the experiments show the wrong of them. For 4, the computation complexity of dist grows with the size of the training set.

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

```
# 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
```

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

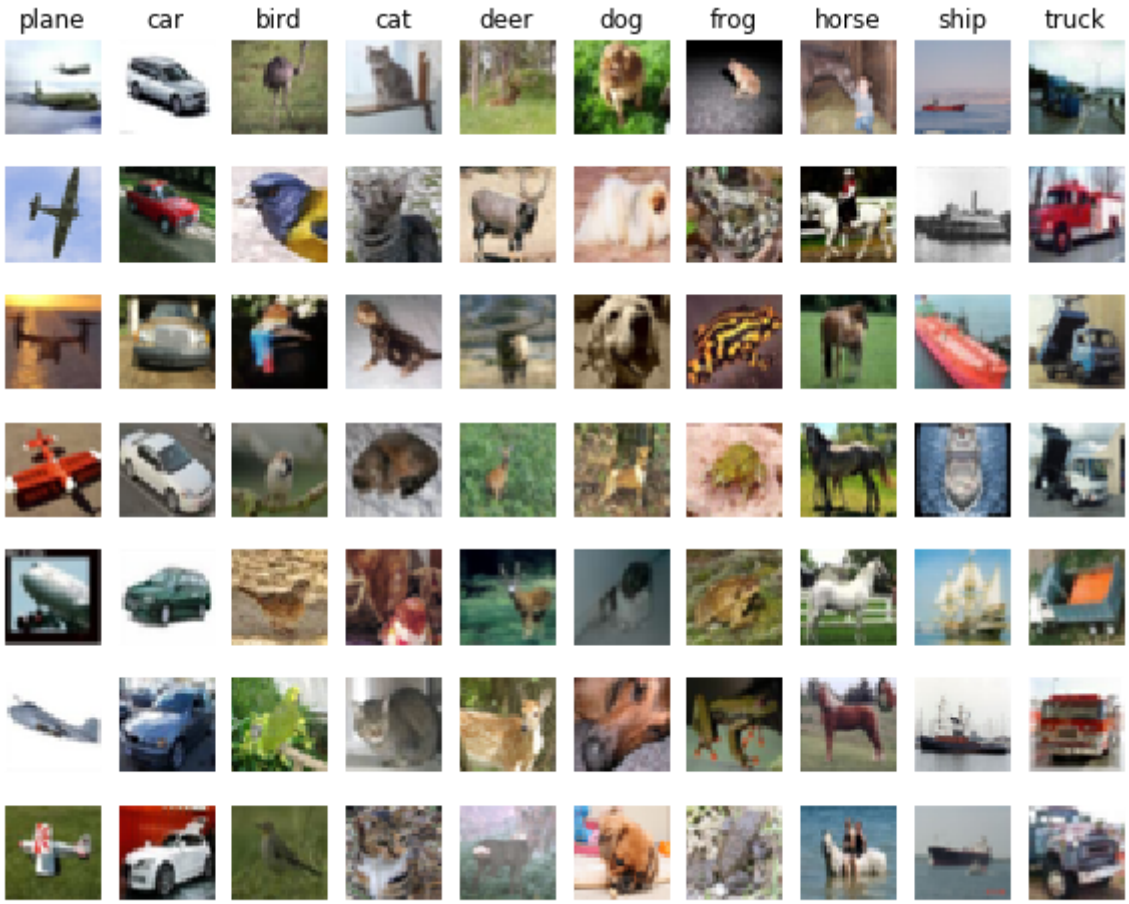
CIFAR-10 Data Loading and Preprocessing

```
# 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 issue)
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)
```

```
Clear previously loaded data.
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

```
# 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', 'truck']
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()
```




```
# 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,)
```

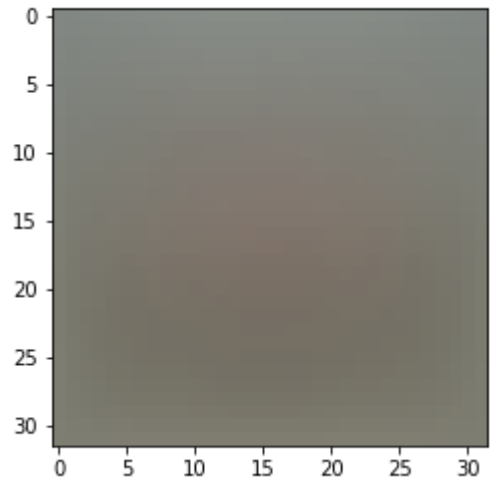
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# 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)
```

```
# 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()

# 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.
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)
```

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 `svm_loss_naive` which uses for loops to evaluate the multiclass SVM loss function.

```
# 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: 9.192991
```

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:

```
# 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: 16.381499 analytic: 16.381499, relative error: 1.393239e-11
numerical: -5.683847 analytic: -5.683847, relative error: 2.261080e-11
numerical: 2.063567 analytic: 2.063567, relative error: 1.330317e-10
numerical: 16.398580 analytic: 16.398580, relative error: 4.026071e-12
numerical: -30.632196 analytic: -30.632196, relative error: 8.924176e-12
numerical: -2.318632 analytic: -2.318632, relative error: 6.092372e-11
numerical: 16.508909 analytic: 16.508909, relative error: 9.353659e-13
numerical: 17.474000 analytic: 17.474000, relative error: 4.270108e-12
numerical: -21.122359 analytic: -21.122359, relative error: 9.170259e-12
numerical: -8.202363 analytic: -8.202363, relative error: 8.304566e-12
numerical: 6.608909 analytic: 6.608909, relative error: 1.872316e-11
numerical: -5.905807 analytic: -5.905807, relative error: 3.280096e-12
numerical: 2.849863 analytic: 2.849863, relative error: 1.959961e-11
numerical: -0.622163 analytic: -0.622163, relative error: 2.774063e-10
numerical: -12.290108 analytic: -12.317102, relative error: 1.096963e-03
numerical: 14.376239 analytic: 14.376239, relative error: 7.440657e-12
numerical: -6.540819 analytic: -6.540819, relative error: 1.287582e-11
numerical: 6.621748 analytic: 6.621748, relative error: 1.953678e-12
numerical: 19.020631 analytic: 19.020631, relative error: 6.956700e-12
numerical: -6.979029 analytic: -6.979029, relative error: 2.523134e-12
```

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*

Your Answer : 1. Because the gradient in the program is computed numerically and in zero max function is continuously non-differentiable. 2. Yes. 3. Similarily, the numerical will fail to check for in zero loss function is not differentiable. 4. Moving the data overall and keeps it away from the marginal point.

```
# 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: 9.192991e+00 computed in 0.291657s
Vectorized loss: 9.192991e+00 computed in 0.002882s
difference: -0.000000
```

```
# 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 0.265734s
Vectorized loss and gradient: computed in 0.002675s
difference: 0.000000
```

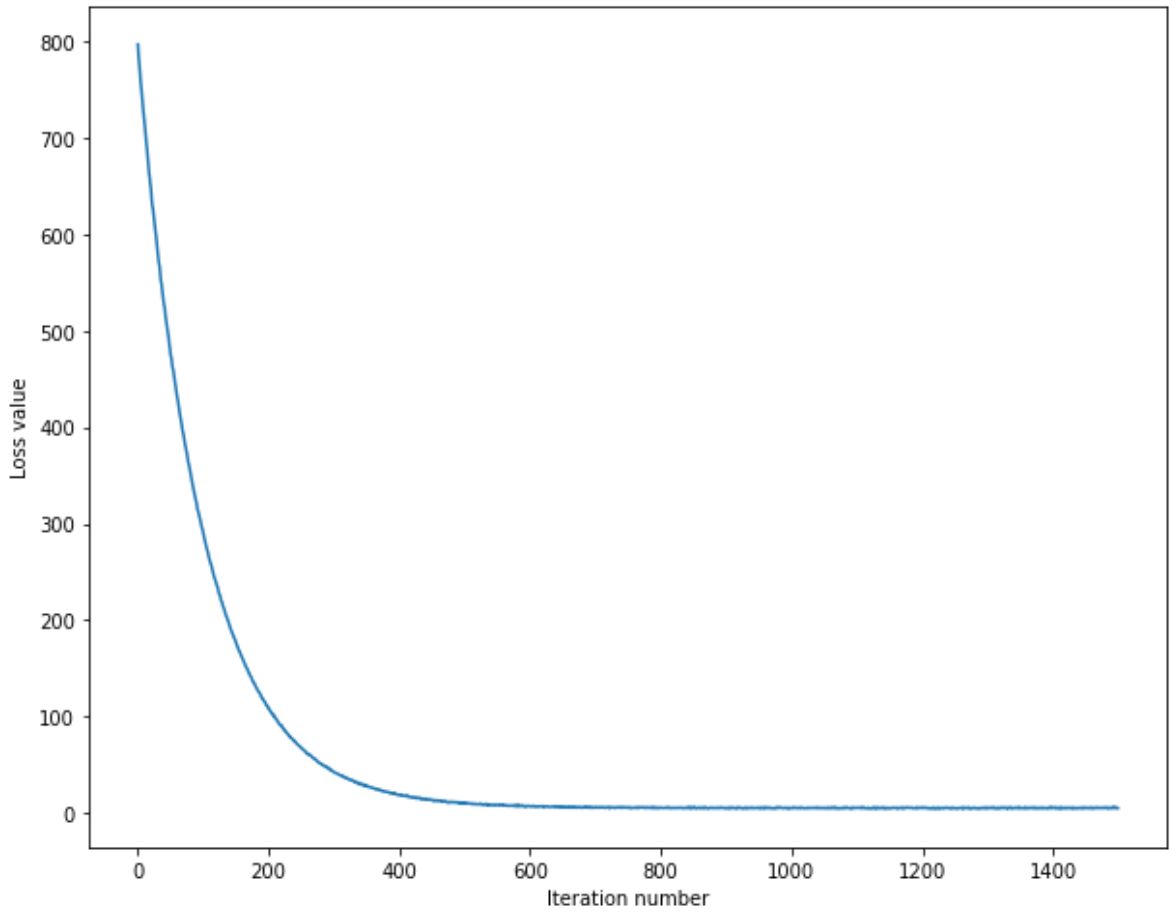
Stochastic Gradient Descent

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. Your code for this part will be written inside `cs231n/classifiers/linear_classifier.py`.

```
# 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 796.756163
iteration 100 / 1500: loss 290.116046
iteration 200 / 1500: loss 107.726910
iteration 300 / 1500: loss 42.488001
iteration 400 / 1500: loss 18.631186
iteration 500 / 1500: loss 10.660977
iteration 600 / 1500: loss 7.155248
iteration 700 / 1500: loss 5.710974
iteration 800 / 1500: loss 5.367862
iteration 900 / 1500: loss 5.166033
iteration 1000 / 1500: loss 5.263973
iteration 1100 / 1500: loss 5.377785
iteration 1200 / 1500: loss 5.602649
iteration 1300 / 1500: loss 4.863869
iteration 1400 / 1500: loss 5.605086
That took 8.018761s
```

```
# 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()
```




```
# 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: 0.365918
validation accuracy: 0.380000
```

```
# 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.

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

# Provided as a reference. You may or may not want to change these
hyperparameters
learning_rates = [1e-7, 5e-5]
regularization_strengths = [2.5e4, 5e4]

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
learning_r = np.linspace(learning_rates[0], learning_rates[1], 10)
reg_s = np.linspace(regularization_strengths[0], regularization_strengths[1],
10)
for lr in learning_r:
    for reg in reg_s:

        svm = LinearSVM()
        svm.train(X_train, y_train, learning_rate=lr, reg=reg, num_iters=1500)
#set 1000 smaller

        yt_pred = svm.predict(X_train)
        yv_pred = svm.predict(X_val)
        acc_t = np.mean(yt_pred == y_train)
        acc_v = np.mean(yv_pred == y_val)
        results[(lr, reg)] = (acc_t, acc_v)

        if acc_v > best_val:
            best_val = acc_v
            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/yangjq/Guests/cs231n/classifiers/linear_svm.py:88: RuntimeWarning: overflow
encountered in double_scalars
  rloss = reg * np.sum(W ** 2)
/home/yangjq/anaconda3/envs/cs231n/lib/python3.8/site-
packages/numpy/core/fromnumeric.py:86: RuntimeWarning: overflow encountered in
reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/home/yangjq/Guests/cs231n/classifiers/linear_svm.py:88: RuntimeWarning: overflow
encountered in square
  rloss = reg * np.sum(W ** 2)
/home/yangjq/Guests/cs231n/classifiers/linear_svm.py:84: RuntimeWarning: overflow
encountered in subtract
  margin = s - s[range(num_train),y].reshape(-1,1) + 1
/home/yangjq/Guests/cs231n/classifiers/linear_svm.py:86: RuntimeWarning: invalid
value encountered in multiply
  margin = (margin > 0) * margin
/home/yangjq/Guests/cs231n/classifiers/linear_svm.py:106: RuntimeWarning:
overflow encountered in multiply
  dW += 2 * reg * W
/home/yangjq/Guests/cs231n/classifiers/linear_svm.py:84: RuntimeWarning: invalid
value encountered in subtract
  margin = s - s[range(num_train),y].reshape(-1,1) + 1
/home/yangjq/Guests/cs231n/classifiers/linear_classifier.py:77: RuntimeWarning:
invalid value encountered in subtract
  self.W = self.W - grad * learning_rate
```

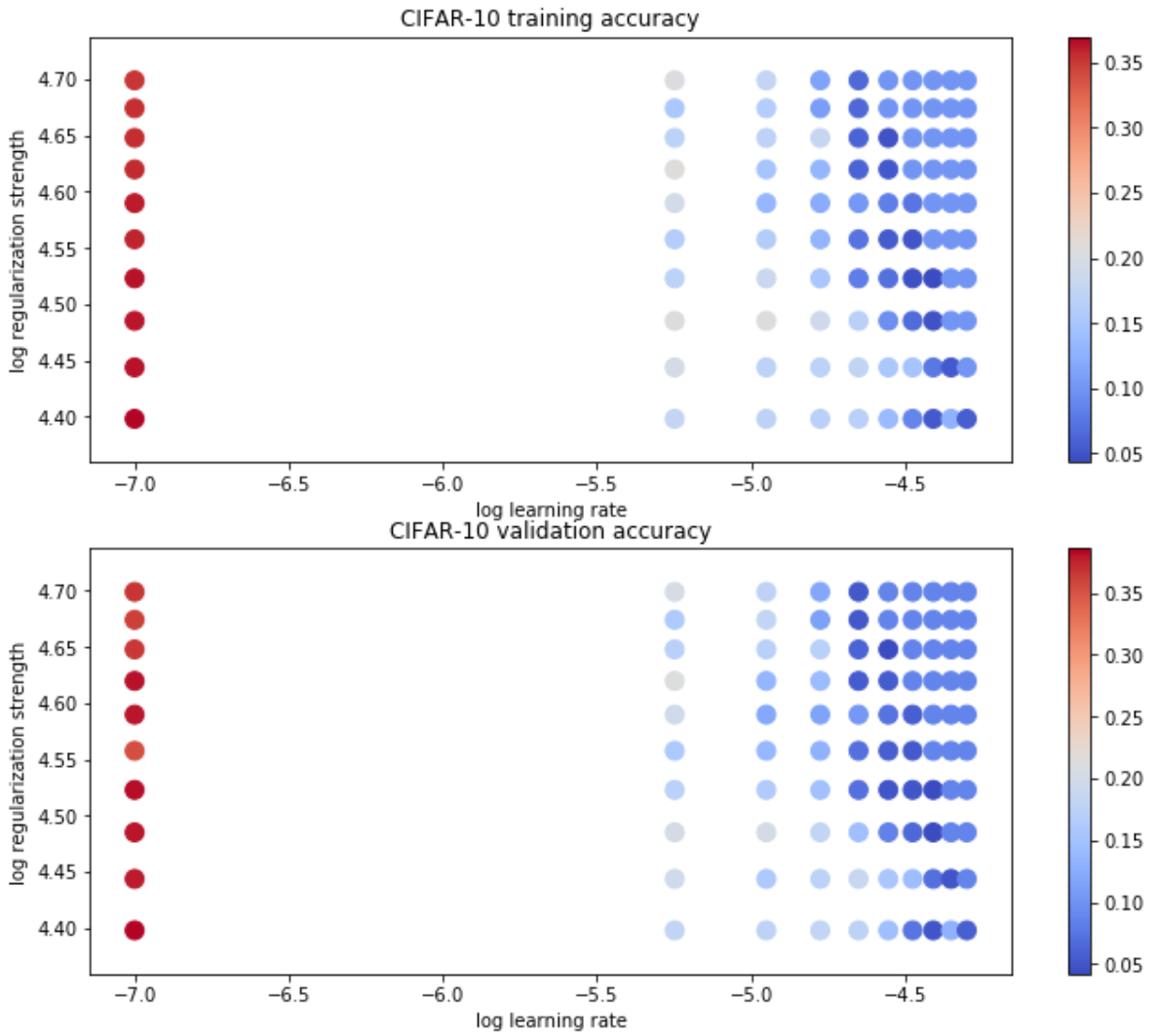
```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.369796 val accuracy: 0.386000
lr 1.000000e-07 reg 2.777778e+04 train accuracy: 0.365347 val accuracy: 0.378000
lr 1.000000e-07 reg 3.055556e+04 train accuracy: 0.363531 val accuracy: 0.380000
lr 1.000000e-07 reg 3.333333e+04 train accuracy: 0.365204 val accuracy: 0.382000
lr 1.000000e-07 reg 3.611111e+04 train accuracy: 0.359796 val accuracy: 0.355000
lr 1.000000e-07 reg 3.888889e+04 train accuracy: 0.362306 val accuracy: 0.380000
lr 1.000000e-07 reg 4.166667e+04 train accuracy: 0.357082 val accuracy: 0.381000
lr 1.000000e-07 reg 4.444444e+04 train accuracy: 0.356673 val accuracy: 0.367000
lr 1.000000e-07 reg 4.722222e+04 train accuracy: 0.355714 val accuracy: 0.362000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.352122 val accuracy: 0.368000
lr 5.644444e-06 reg 2.500000e+04 train accuracy: 0.177327 val accuracy: 0.181000
lr 5.644444e-06 reg 2.777778e+04 train accuracy: 0.195796 val accuracy: 0.197000
lr 5.644444e-06 reg 3.055556e+04 train accuracy: 0.208122 val accuracy: 0.204000
lr 5.644444e-06 reg 3.333333e+04 train accuracy: 0.169469 val accuracy: 0.174000
lr 5.644444e-06 reg 3.611111e+04 train accuracy: 0.162755 val accuracy: 0.161000
lr 5.644444e-06 reg 3.888889e+04 train accuracy: 0.193469 val accuracy: 0.197000
lr 5.644444e-06 reg 4.166667e+04 train accuracy: 0.208776 val accuracy: 0.213000
lr 5.644444e-06 reg 4.444444e+04 train accuracy: 0.169531 val accuracy: 0.173000
lr 5.644444e-06 reg 4.722222e+04 train accuracy: 0.153816 val accuracy: 0.162000
lr 5.644444e-06 reg 5.000000e+04 train accuracy: 0.203245 val accuracy: 0.207000
lr 1.118889e-05 reg 2.500000e+04 train accuracy: 0.170816 val accuracy: 0.177000
```

```
lr 1.118889e-05 reg 2.777778e+04 train accuracy: 0.169959 val accuracy: 0.160000
lr 1.118889e-05 reg 3.055556e+04 train accuracy: 0.204551 val accuracy: 0.204000
lr 1.118889e-05 reg 3.333333e+04 train accuracy: 0.184102 val accuracy: 0.164000
lr 1.118889e-05 reg 3.611111e+04 train accuracy: 0.160490 val accuracy: 0.138000
lr 1.118889e-05 reg 3.888889e+04 train accuracy: 0.132796 val accuracy: 0.120000
lr 1.118889e-05 reg 4.166667e+04 train accuracy: 0.148796 val accuracy: 0.136000
lr 1.118889e-05 reg 4.444444e+04 train accuracy: 0.170980 val accuracy: 0.172000
lr 1.118889e-05 reg 4.722222e+04 train accuracy: 0.162286 val accuracy: 0.182000
lr 1.118889e-05 reg 5.000000e+04 train accuracy: 0.178265 val accuracy: 0.177000
lr 1.673333e-05 reg 2.500000e+04 train accuracy: 0.165878 val accuracy: 0.182000
lr 1.673333e-05 reg 2.777778e+04 train accuracy: 0.170163 val accuracy: 0.176000
lr 1.673333e-05 reg 3.055556e+04 train accuracy: 0.189490 val accuracy: 0.183000
lr 1.673333e-05 reg 3.333333e+04 train accuracy: 0.150490 val accuracy: 0.149000
lr 1.673333e-05 reg 3.611111e+04 train accuracy: 0.130245 val accuracy: 0.132000
lr 1.673333e-05 reg 3.888889e+04 train accuracy: 0.121612 val accuracy: 0.115000
lr 1.673333e-05 reg 4.166667e+04 train accuracy: 0.131122 val accuracy: 0.142000
lr 1.673333e-05 reg 4.444444e+04 train accuracy: 0.179633 val accuracy: 0.172000
lr 1.673333e-05 reg 4.722222e+04 train accuracy: 0.107776 val accuracy: 0.112000
lr 1.673333e-05 reg 5.000000e+04 train accuracy: 0.111980 val accuracy: 0.119000
lr 2.227778e-05 reg 2.500000e+04 train accuracy: 0.167041 val accuracy: 0.177000
lr 2.227778e-05 reg 2.777778e+04 train accuracy: 0.174796 val accuracy: 0.188000
lr 2.227778e-05 reg 3.055556e+04 train accuracy: 0.167551 val accuracy: 0.147000
lr 2.227778e-05 reg 3.333333e+04 train accuracy: 0.080714 val accuracy: 0.071000
lr 2.227778e-05 reg 3.611111e+04 train accuracy: 0.072571 val accuracy: 0.071000
lr 2.227778e-05 reg 3.888889e+04 train accuracy: 0.102633 val accuracy: 0.104000
lr 2.227778e-05 reg 4.166667e+04 train accuracy: 0.061163 val accuracy: 0.055000
lr 2.227778e-05 reg 4.444444e+04 train accuracy: 0.060980 val accuracy: 0.061000
lr 2.227778e-05 reg 4.722222e+04 train accuracy: 0.064306 val accuracy: 0.052000
lr 2.227778e-05 reg 5.000000e+04 train accuracy: 0.064755 val accuracy: 0.052000
lr 2.782222e-05 reg 2.500000e+04 train accuracy: 0.136776 val accuracy: 0.146000
lr 2.782222e-05 reg 2.777778e+04 train accuracy: 0.153061 val accuracy: 0.157000
lr 2.782222e-05 reg 3.055556e+04 train accuracy: 0.095245 val accuracy: 0.086000
lr 2.782222e-05 reg 3.333333e+04 train accuracy: 0.070837 val accuracy: 0.049000
lr 2.782222e-05 reg 3.611111e+04 train accuracy: 0.055327 val accuracy: 0.057000
lr 2.782222e-05 reg 3.888889e+04 train accuracy: 0.081592 val accuracy: 0.074000
lr 2.782222e-05 reg 4.166667e+04 train accuracy: 0.052633 val accuracy: 0.055000
lr 2.782222e-05 reg 4.444444e+04 train accuracy: 0.048898 val accuracy: 0.042000
lr 2.782222e-05 reg 4.722222e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 2.782222e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 3.336667e-05 reg 2.500000e+04 train accuracy: 0.087592 val accuracy: 0.075000
lr 3.336667e-05 reg 2.777778e+04 train accuracy: 0.148673 val accuracy: 0.145000
lr 3.336667e-05 reg 3.055556e+04 train accuracy: 0.066184 val accuracy: 0.064000
lr 3.336667e-05 reg 3.333333e+04 train accuracy: 0.047755 val accuracy: 0.050000
lr 3.336667e-05 reg 3.611111e+04 train accuracy: 0.049735 val accuracy: 0.054000
lr 3.336667e-05 reg 3.888889e+04 train accuracy: 0.074796 val accuracy: 0.058000
lr 3.336667e-05 reg 4.166667e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 3.336667e-05 reg 4.444444e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 3.336667e-05 reg 4.722222e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 3.336667e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 3.891111e-05 reg 2.500000e+04 train accuracy: 0.053306 val accuracy: 0.050000
lr 3.891111e-05 reg 2.777778e+04 train accuracy: 0.077163 val accuracy: 0.071000
lr 3.891111e-05 reg 3.055556e+04 train accuracy: 0.048714 val accuracy: 0.043000
lr 3.891111e-05 reg 3.333333e+04 train accuracy: 0.043041 val accuracy: 0.043000
lr 3.891111e-05 reg 3.611111e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 3.891111e-05 reg 3.888889e+04 train accuracy: 0.100265 val accuracy: 0.087000
```

```
lr 3.891111e-05 reg 4.166667e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 3.891111e-05 reg 4.444444e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 3.891111e-05 reg 4.722222e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 3.891111e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 4.445556e-05 reg 2.500000e+04 train accuracy: 0.128265 val accuracy: 0.132000
lr 4.445556e-05 reg 2.777778e+04 train accuracy: 0.053878 val accuracy: 0.050000
lr 4.445556e-05 reg 3.055556e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 4.445556e-05 reg 3.333333e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 4.445556e-05 reg 3.611111e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 4.445556e-05 reg 3.888889e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 4.445556e-05 reg 4.166667e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 4.445556e-05 reg 4.444444e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 4.445556e-05 reg 4.722222e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 4.445556e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.056041 val accuracy: 0.057000
lr 5.000000e-05 reg 2.777778e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 3.055556e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 3.333333e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 3.611111e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 3.888889e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 4.166667e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 4.444444e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 4.722222e+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: 0.386000
```

```
# Visualize the cross-validation results
import math
import pdb

# pdb.set_trace()

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.tight_layout(pad=3)
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()
```

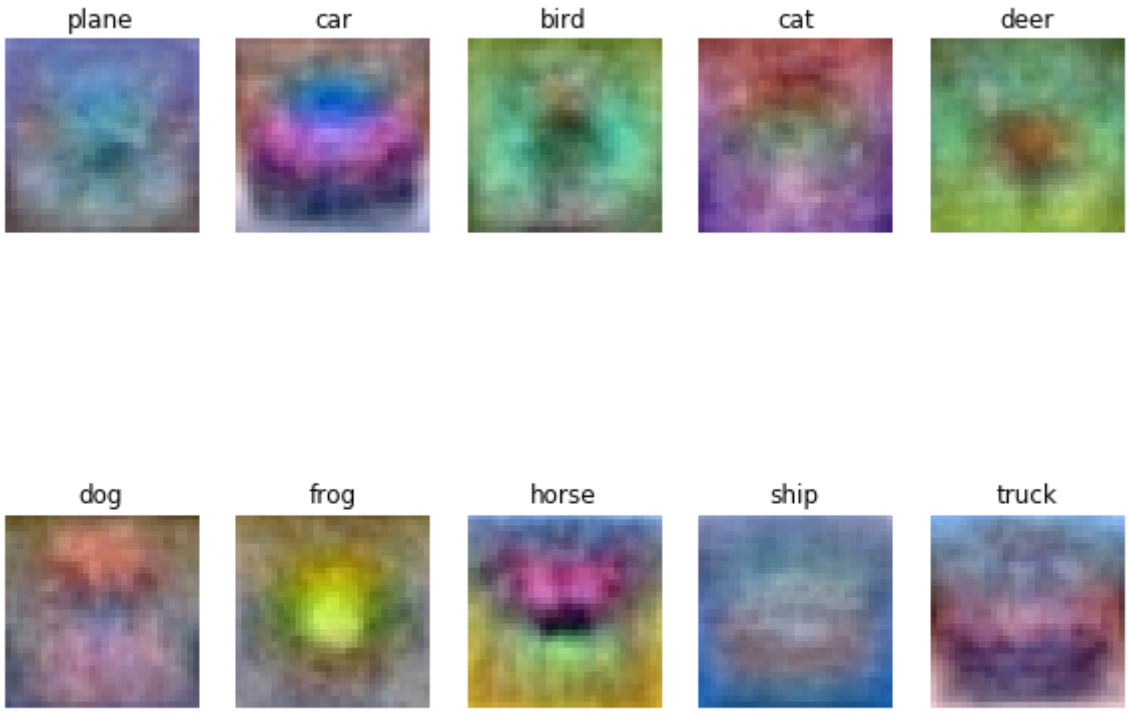



```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.357000
```

```
# 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', '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])
```



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 : *1. They both roughly show the shape of the object and the background color. 2. Because SVM just like taking templates of images from different classes in the training set.*

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

```
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
```

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

```
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'

    # 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

    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


# 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`.

```
# 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: 2.370858
sanity check: 2.302585
```

Inline Question 1

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

Your Answer : *Because when error elements in W -> 0, the loss will close to -log(0.1).*

```
# 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: 2.982904 analytic: 2.982904, relative error: 1.677347e-08
numerical: 1.544706 analytic: 1.544706, relative error: 2.894244e-08
numerical: 1.283081 analytic: 1.283081, relative error: 2.200545e-08
numerical: 0.438750 analytic: 0.438750, relative error: 2.050933e-08
numerical: -0.989036 analytic: -0.989036, relative error: 1.531616e-08
numerical: -1.145730 analytic: -1.145730, relative error: 3.611307e-08
numerical: 1.261795 analytic: 1.261795, relative error: 1.505190e-08
numerical: 1.195429 analytic: 1.195429, relative error: 2.567748e-08
numerical: 2.256426 analytic: 2.256426, relative error: 1.268924e-08
numerical: -1.516667 analytic: -1.516667, relative error: 3.488449e-08
numerical: -1.763473 analytic: -1.763473, relative error: 1.496496e-08
numerical: 0.485159 analytic: 0.485159, relative error: 5.516444e-08
numerical: -0.997272 analytic: -0.997272, relative error: 1.066476e-08
numerical: -0.576475 analytic: -0.576475, relative error: 3.291816e-08
numerical: 0.208924 analytic: 0.208924, relative error: 7.472543e-08
numerical: -0.511623 analytic: -0.511622, relative error: 3.096960e-08
numerical: 0.125176 analytic: 0.125176, relative error: 5.166497e-07
numerical: -0.435403 analytic: -0.435403, relative error: 2.247187e-08
numerical: 2.968895 analytic: 2.968894, relative error: 2.261817e-08
numerical: -0.576463 analytic: -0.576463, relative error: 5.393885e-08
```

```
# 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: 2.370858e+00 computed in 0.126608s
vectorized loss: 2.370858e+00 computed in 0.004333s
Loss difference: 0.000000
Gradient difference: 0.000000
```

```
# 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

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

# Provided as a reference. You may or may not want to change these
hyperparameters
learning_rates = [1e-7, 5e-7]
regularization_strengths = [2.5e4, 5e4]

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
learning_r = np.linspace(learning_rates[0], learning_rates[1], 15)
reg_s = np.linspace(regularization_strengths[0], regularization_strengths[1],
15)
for lr in learning_r:
    for reg in reg_s:

        softmax = Softmax()
        softmax.train(X_train, y_train, learning_rate=lr, reg=reg, num_iters =
200) #set smaller

        yt_pred = softmax.predict(X_train)
        yv_pred = softmax.predict(X_val)
        acc_t = np.mean(yt_pred == y_train)
        acc_v = np.mean(yv_pred == y_val)
        results[(lr, reg)] = (acc_t, acc_v)

        if acc_v > best_val:
            best_val = acc_v
            best_softmax = softmax

# *****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 2.500000e+04 train accuracy: 0.199878 val accuracy: 0.203000
lr 1.000000e-07 reg 2.678571e+04 train accuracy: 0.205286 val accuracy: 0.223000
lr 1.000000e-07 reg 2.857143e+04 train accuracy: 0.205204 val accuracy: 0.211000
lr 1.000000e-07 reg 3.035714e+04 train accuracy: 0.198265 val accuracy: 0.197000
lr 1.000000e-07 reg 3.214286e+04 train accuracy: 0.221633 val accuracy: 0.225000
lr 1.000000e-07 reg 3.392857e+04 train accuracy: 0.209102 val accuracy: 0.219000
lr 1.000000e-07 reg 3.571429e+04 train accuracy: 0.215898 val accuracy: 0.212000
lr 1.000000e-07 reg 3.750000e+04 train accuracy: 0.226694 val accuracy: 0.228000
lr 1.000000e-07 reg 3.928571e+04 train accuracy: 0.216898 val accuracy: 0.221000
lr 1.000000e-07 reg 4.107143e+04 train accuracy: 0.236939 val accuracy: 0.241000
lr 1.000000e-07 reg 4.285714e+04 train accuracy: 0.246878 val accuracy: 0.235000
lr 1.000000e-07 reg 4.464286e+04 train accuracy: 0.234918 val accuracy: 0.255000
lr 1.000000e-07 reg 4.642857e+04 train accuracy: 0.247449 val accuracy: 0.241000
lr 1.000000e-07 reg 4.821429e+04 train accuracy: 0.242959 val accuracy: 0.252000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.243714 val accuracy: 0.242000
lr 1.285714e-07 reg 2.500000e+04 train accuracy: 0.235510 val accuracy: 0.238000
lr 1.285714e-07 reg 2.678571e+04 train accuracy: 0.228633 val accuracy: 0.230000
lr 1.285714e-07 reg 2.857143e+04 train accuracy: 0.240510 val accuracy: 0.261000
lr 1.285714e-07 reg 3.035714e+04 train accuracy: 0.235980 val accuracy: 0.240000
lr 1.285714e-07 reg 3.214286e+04 train accuracy: 0.242143 val accuracy: 0.234000
lr 1.285714e-07 reg 3.392857e+04 train accuracy: 0.260204 val accuracy: 0.260000
lr 1.285714e-07 reg 3.571429e+04 train accuracy: 0.267612 val accuracy: 0.254000
lr 1.285714e-07 reg 3.750000e+04 train accuracy: 0.258388 val accuracy: 0.279000
lr 1.285714e-07 reg 3.928571e+04 train accuracy: 0.268163 val accuracy: 0.281000
lr 1.285714e-07 reg 4.107143e+04 train accuracy: 0.265857 val accuracy: 0.284000
lr 1.285714e-07 reg 4.285714e+04 train accuracy: 0.276878 val accuracy: 0.295000
lr 1.285714e-07 reg 4.464286e+04 train accuracy: 0.269837 val accuracy: 0.285000
lr 1.285714e-07 reg 4.642857e+04 train accuracy: 0.268592 val accuracy: 0.278000
lr 1.285714e-07 reg 4.821429e+04 train accuracy: 0.280122 val accuracy: 0.288000
lr 1.285714e-07 reg 5.000000e+04 train accuracy: 0.280551 val accuracy: 0.285000
lr 1.571429e-07 reg 2.500000e+04 train accuracy: 0.237122 val accuracy: 0.224000
lr 1.571429e-07 reg 2.678571e+04 train accuracy: 0.264408 val accuracy: 0.271000
lr 1.571429e-07 reg 2.857143e+04 train accuracy: 0.264837 val accuracy: 0.254000
lr 1.571429e-07 reg 3.035714e+04 train accuracy: 0.264571 val accuracy: 0.278000
lr 1.571429e-07 reg 3.214286e+04 train accuracy: 0.275429 val accuracy: 0.297000
lr 1.571429e-07 reg 3.392857e+04 train accuracy: 0.280408 val accuracy: 0.284000
lr 1.571429e-07 reg 3.571429e+04 train accuracy: 0.286551 val accuracy: 0.300000
lr 1.571429e-07 reg 3.750000e+04 train accuracy: 0.285020 val accuracy: 0.297000
lr 1.571429e-07 reg 3.928571e+04 train accuracy: 0.285918 val accuracy: 0.299000
lr 1.571429e-07 reg 4.107143e+04 train accuracy: 0.289857 val accuracy: 0.314000
lr 1.571429e-07 reg 4.285714e+04 train accuracy: 0.291776 val accuracy: 0.299000
lr 1.571429e-07 reg 4.464286e+04 train accuracy: 0.295347 val accuracy: 0.326000
lr 1.571429e-07 reg 4.642857e+04 train accuracy: 0.300673 val accuracy: 0.309000
lr 1.571429e-07 reg 4.821429e+04 train accuracy: 0.296265 val accuracy: 0.306000
lr 1.571429e-07 reg 5.000000e+04 train accuracy: 0.292571 val accuracy: 0.299000
lr 1.857143e-07 reg 2.500000e+04 train accuracy: 0.272020 val accuracy: 0.309000
lr 1.857143e-07 reg 2.678571e+04 train accuracy: 0.287490 val accuracy: 0.298000
lr 1.857143e-07 reg 2.857143e+04 train accuracy: 0.286592 val accuracy: 0.309000
lr 1.857143e-07 reg 3.035714e+04 train accuracy: 0.288633 val accuracy: 0.324000
lr 1.857143e-07 reg 3.214286e+04 train accuracy: 0.294367 val accuracy: 0.315000
lr 1.857143e-07 reg 3.392857e+04 train accuracy: 0.298490 val accuracy: 0.306000
lr 1.857143e-07 reg 3.571429e+04 train accuracy: 0.297061 val accuracy: 0.307000
lr 1.857143e-07 reg 3.750000e+04 train accuracy: 0.292878 val accuracy: 0.300000
lr 1.857143e-07 reg 3.928571e+04 train accuracy: 0.305735 val accuracy: 0.323000
lr 1.857143e-07 reg 4.107143e+04 train accuracy: 0.299714 val accuracy: 0.314000
```

```
lr 1.857143e-07 reg 4.285714e+04 train accuracy: 0.301898 val accuracy: 0.320000
lr 1.857143e-07 reg 4.464286e+04 train accuracy: 0.303204 val accuracy: 0.334000
lr 1.857143e-07 reg 4.642857e+04 train accuracy: 0.298510 val accuracy: 0.322000
lr 1.857143e-07 reg 4.821429e+04 train accuracy: 0.305347 val accuracy: 0.312000
lr 1.857143e-07 reg 5.000000e+04 train accuracy: 0.299755 val accuracy: 0.312000
lr 2.142857e-07 reg 2.500000e+04 train accuracy: 0.292571 val accuracy: 0.299000
lr 2.142857e-07 reg 2.678571e+04 train accuracy: 0.299490 val accuracy: 0.306000
lr 2.142857e-07 reg 2.857143e+04 train accuracy: 0.302388 val accuracy: 0.323000
lr 2.142857e-07 reg 3.035714e+04 train accuracy: 0.305735 val accuracy: 0.328000
lr 2.142857e-07 reg 3.214286e+04 train accuracy: 0.308673 val accuracy: 0.326000
lr 2.142857e-07 reg 3.392857e+04 train accuracy: 0.305959 val accuracy: 0.324000
lr 2.142857e-07 reg 3.571429e+04 train accuracy: 0.305796 val accuracy: 0.330000
lr 2.142857e-07 reg 3.750000e+04 train accuracy: 0.311735 val accuracy: 0.317000
lr 2.142857e-07 reg 3.928571e+04 train accuracy: 0.303878 val accuracy: 0.310000
lr 2.142857e-07 reg 4.107143e+04 train accuracy: 0.312551 val accuracy: 0.315000
lr 2.142857e-07 reg 4.285714e+04 train accuracy: 0.301980 val accuracy: 0.312000
lr 2.142857e-07 reg 4.464286e+04 train accuracy: 0.306837 val accuracy: 0.315000
lr 2.142857e-07 reg 4.642857e+04 train accuracy: 0.309204 val accuracy: 0.329000
lr 2.142857e-07 reg 4.821429e+04 train accuracy: 0.306020 val accuracy: 0.322000
lr 2.142857e-07 reg 5.000000e+04 train accuracy: 0.309388 val accuracy: 0.321000
lr 2.428571e-07 reg 2.500000e+04 train accuracy: 0.302918 val accuracy: 0.310000
lr 2.428571e-07 reg 2.678571e+04 train accuracy: 0.312041 val accuracy: 0.327000
lr 2.428571e-07 reg 2.857143e+04 train accuracy: 0.306000 val accuracy: 0.327000
lr 2.428571e-07 reg 3.035714e+04 train accuracy: 0.310959 val accuracy: 0.334000
lr 2.428571e-07 reg 3.214286e+04 train accuracy: 0.310408 val accuracy: 0.317000
lr 2.428571e-07 reg 3.392857e+04 train accuracy: 0.313592 val accuracy: 0.333000
lr 2.428571e-07 reg 3.571429e+04 train accuracy: 0.316837 val accuracy: 0.324000
lr 2.428571e-07 reg 3.750000e+04 train accuracy: 0.308673 val accuracy: 0.328000
lr 2.428571e-07 reg 3.928571e+04 train accuracy: 0.308041 val accuracy: 0.311000
lr 2.428571e-07 reg 4.107143e+04 train accuracy: 0.305980 val accuracy: 0.315000
lr 2.428571e-07 reg 4.285714e+04 train accuracy: 0.312714 val accuracy: 0.326000
lr 2.428571e-07 reg 4.464286e+04 train accuracy: 0.314449 val accuracy: 0.331000
lr 2.428571e-07 reg 4.642857e+04 train accuracy: 0.304939 val accuracy: 0.320000
lr 2.428571e-07 reg 4.821429e+04 train accuracy: 0.316878 val accuracy: 0.336000
lr 2.428571e-07 reg 5.000000e+04 train accuracy: 0.300490 val accuracy: 0.319000
lr 2.714286e-07 reg 2.500000e+04 train accuracy: 0.323510 val accuracy: 0.348000
lr 2.714286e-07 reg 2.678571e+04 train accuracy: 0.315714 val accuracy: 0.334000
lr 2.714286e-07 reg 2.857143e+04 train accuracy: 0.316980 val accuracy: 0.330000
lr 2.714286e-07 reg 3.035714e+04 train accuracy: 0.310612 val accuracy: 0.322000
lr 2.714286e-07 reg 3.214286e+04 train accuracy: 0.314163 val accuracy: 0.336000
lr 2.714286e-07 reg 3.392857e+04 train accuracy: 0.316816 val accuracy: 0.324000
lr 2.714286e-07 reg 3.571429e+04 train accuracy: 0.314224 val accuracy: 0.334000
lr 2.714286e-07 reg 3.750000e+04 train accuracy: 0.310816 val accuracy: 0.338000
lr 2.714286e-07 reg 3.928571e+04 train accuracy: 0.309714 val accuracy: 0.333000
lr 2.714286e-07 reg 4.107143e+04 train accuracy: 0.310816 val accuracy: 0.321000
lr 2.714286e-07 reg 4.285714e+04 train accuracy: 0.314735 val accuracy: 0.319000
lr 2.714286e-07 reg 4.464286e+04 train accuracy: 0.308796 val accuracy: 0.323000
lr 2.714286e-07 reg 4.642857e+04 train accuracy: 0.314551 val accuracy: 0.325000
lr 2.714286e-07 reg 4.821429e+04 train accuracy: 0.300735 val accuracy: 0.318000
lr 2.714286e-07 reg 5.000000e+04 train accuracy: 0.307408 val accuracy: 0.327000
lr 3.000000e-07 reg 2.500000e+04 train accuracy: 0.322531 val accuracy: 0.331000
lr 3.000000e-07 reg 2.678571e+04 train accuracy: 0.323306 val accuracy: 0.336000
lr 3.000000e-07 reg 2.857143e+04 train accuracy: 0.320490 val accuracy: 0.333000
lr 3.000000e-07 reg 3.035714e+04 train accuracy: 0.318878 val accuracy: 0.331000
lr 3.000000e-07 reg 3.214286e+04 train accuracy: 0.315327 val accuracy: 0.327000
```

```
lr 3.000000e-07 reg 3.392857e+04 train accuracy: 0.313959 val accuracy: 0.330000
lr 3.000000e-07 reg 3.571429e+04 train accuracy: 0.311327 val accuracy: 0.321000
lr 3.000000e-07 reg 3.750000e+04 train accuracy: 0.305163 val accuracy: 0.321000
lr 3.000000e-07 reg 3.928571e+04 train accuracy: 0.302857 val accuracy: 0.325000
lr 3.000000e-07 reg 4.107143e+04 train accuracy: 0.316041 val accuracy: 0.339000
lr 3.000000e-07 reg 4.285714e+04 train accuracy: 0.313735 val accuracy: 0.330000
lr 3.000000e-07 reg 4.464286e+04 train accuracy: 0.302796 val accuracy: 0.321000
lr 3.000000e-07 reg 4.642857e+04 train accuracy: 0.292653 val accuracy: 0.314000
lr 3.000000e-07 reg 4.821429e+04 train accuracy: 0.302531 val accuracy: 0.318000
lr 3.000000e-07 reg 5.000000e+04 train accuracy: 0.304224 val accuracy: 0.305000
lr 3.285714e-07 reg 2.500000e+04 train accuracy: 0.322980 val accuracy: 0.336000
lr 3.285714e-07 reg 2.678571e+04 train accuracy: 0.322980 val accuracy: 0.349000
lr 3.285714e-07 reg 2.857143e+04 train accuracy: 0.318714 val accuracy: 0.337000
lr 3.285714e-07 reg 3.035714e+04 train accuracy: 0.321347 val accuracy: 0.337000
lr 3.285714e-07 reg 3.214286e+04 train accuracy: 0.319694 val accuracy: 0.331000
lr 3.285714e-07 reg 3.392857e+04 train accuracy: 0.310653 val accuracy: 0.314000
lr 3.285714e-07 reg 3.571429e+04 train accuracy: 0.315143 val accuracy: 0.332000
lr 3.285714e-07 reg 3.750000e+04 train accuracy: 0.319306 val accuracy: 0.325000
lr 3.285714e-07 reg 3.928571e+04 train accuracy: 0.306490 val accuracy: 0.322000
lr 3.285714e-07 reg 4.107143e+04 train accuracy: 0.317224 val accuracy: 0.336000
lr 3.285714e-07 reg 4.285714e+04 train accuracy: 0.306796 val accuracy: 0.326000
lr 3.285714e-07 reg 4.464286e+04 train accuracy: 0.300592 val accuracy: 0.311000
lr 3.285714e-07 reg 4.642857e+04 train accuracy: 0.311531 val accuracy: 0.310000
lr 3.285714e-07 reg 4.821429e+04 train accuracy: 0.296714 val accuracy: 0.307000
lr 3.285714e-07 reg 5.000000e+04 train accuracy: 0.300898 val accuracy: 0.312000
lr 3.571429e-07 reg 2.500000e+04 train accuracy: 0.322020 val accuracy: 0.334000
lr 3.571429e-07 reg 2.678571e+04 train accuracy: 0.325694 val accuracy: 0.345000
lr 3.571429e-07 reg 2.857143e+04 train accuracy: 0.333857 val accuracy: 0.342000
lr 3.571429e-07 reg 3.035714e+04 train accuracy: 0.335245 val accuracy: 0.352000
lr 3.571429e-07 reg 3.214286e+04 train accuracy: 0.320571 val accuracy: 0.335000
lr 3.571429e-07 reg 3.392857e+04 train accuracy: 0.306449 val accuracy: 0.320000
lr 3.571429e-07 reg 3.571429e+04 train accuracy: 0.318102 val accuracy: 0.331000
lr 3.571429e-07 reg 3.750000e+04 train accuracy: 0.311204 val accuracy: 0.317000
lr 3.571429e-07 reg 3.928571e+04 train accuracy: 0.317408 val accuracy: 0.318000
lr 3.571429e-07 reg 4.107143e+04 train accuracy: 0.315633 val accuracy: 0.324000
lr 3.571429e-07 reg 4.285714e+04 train accuracy: 0.311918 val accuracy: 0.323000
lr 3.571429e-07 reg 4.464286e+04 train accuracy: 0.310429 val accuracy: 0.321000
lr 3.571429e-07 reg 4.642857e+04 train accuracy: 0.307306 val accuracy: 0.328000
lr 3.571429e-07 reg 4.821429e+04 train accuracy: 0.304184 val accuracy: 0.309000
lr 3.571429e-07 reg 5.000000e+04 train accuracy: 0.300898 val accuracy: 0.314000
lr 3.857143e-07 reg 2.500000e+04 train accuracy: 0.321878 val accuracy: 0.339000
lr 3.857143e-07 reg 2.678571e+04 train accuracy: 0.328306 val accuracy: 0.345000
lr 3.857143e-07 reg 2.857143e+04 train accuracy: 0.324735 val accuracy: 0.338000
lr 3.857143e-07 reg 3.035714e+04 train accuracy: 0.326633 val accuracy: 0.339000
lr 3.857143e-07 reg 3.214286e+04 train accuracy: 0.322571 val accuracy: 0.341000
lr 3.857143e-07 reg 3.392857e+04 train accuracy: 0.315510 val accuracy: 0.333000
lr 3.857143e-07 reg 3.571429e+04 train accuracy: 0.319041 val accuracy: 0.329000
lr 3.857143e-07 reg 3.750000e+04 train accuracy: 0.321490 val accuracy: 0.328000
lr 3.857143e-07 reg 3.928571e+04 train accuracy: 0.315449 val accuracy: 0.324000
lr 3.857143e-07 reg 4.107143e+04 train accuracy: 0.300755 val accuracy: 0.321000
lr 3.857143e-07 reg 4.285714e+04 train accuracy: 0.298245 val accuracy: 0.315000
lr 3.857143e-07 reg 4.464286e+04 train accuracy: 0.314755 val accuracy: 0.315000
lr 3.857143e-07 reg 4.642857e+04 train accuracy: 0.304673 val accuracy: 0.324000
lr 3.857143e-07 reg 4.821429e+04 train accuracy: 0.298265 val accuracy: 0.315000
lr 3.857143e-07 reg 5.000000e+04 train accuracy: 0.294592 val accuracy: 0.308000
```

```
lr 4.142857e-07 reg 2.500000e+04 train accuracy: 0.319041 val accuracy: 0.339000
lr 4.142857e-07 reg 2.678571e+04 train accuracy: 0.320959 val accuracy: 0.341000
lr 4.142857e-07 reg 2.857143e+04 train accuracy: 0.307735 val accuracy: 0.321000
lr 4.142857e-07 reg 3.035714e+04 train accuracy: 0.315531 val accuracy: 0.321000
lr 4.142857e-07 reg 3.214286e+04 train accuracy: 0.318531 val accuracy: 0.325000
lr 4.142857e-07 reg 3.392857e+04 train accuracy: 0.312673 val accuracy: 0.336000
lr 4.142857e-07 reg 3.571429e+04 train accuracy: 0.310265 val accuracy: 0.330000
lr 4.142857e-07 reg 3.750000e+04 train accuracy: 0.318327 val accuracy: 0.310000
lr 4.142857e-07 reg 3.928571e+04 train accuracy: 0.314286 val accuracy: 0.325000
lr 4.142857e-07 reg 4.107143e+04 train accuracy: 0.301265 val accuracy: 0.310000
lr 4.142857e-07 reg 4.285714e+04 train accuracy: 0.313061 val accuracy: 0.323000
lr 4.142857e-07 reg 4.464286e+04 train accuracy: 0.310347 val accuracy: 0.328000
lr 4.142857e-07 reg 4.642857e+04 train accuracy: 0.313327 val accuracy: 0.336000
lr 4.142857e-07 reg 4.821429e+04 train accuracy: 0.295878 val accuracy: 0.315000
lr 4.142857e-07 reg 5.000000e+04 train accuracy: 0.305510 val accuracy: 0.316000
lr 4.428571e-07 reg 2.500000e+04 train accuracy: 0.333286 val accuracy: 0.345000
lr 4.428571e-07 reg 2.678571e+04 train accuracy: 0.316816 val accuracy: 0.324000
lr 4.428571e-07 reg 2.857143e+04 train accuracy: 0.320490 val accuracy: 0.334000
lr 4.428571e-07 reg 3.035714e+04 train accuracy: 0.320776 val accuracy: 0.344000
lr 4.428571e-07 reg 3.214286e+04 train accuracy: 0.315449 val accuracy: 0.318000
lr 4.428571e-07 reg 3.392857e+04 train accuracy: 0.318592 val accuracy: 0.330000
lr 4.428571e-07 reg 3.571429e+04 train accuracy: 0.316857 val accuracy: 0.329000
lr 4.428571e-07 reg 3.750000e+04 train accuracy: 0.304959 val accuracy: 0.316000
lr 4.428571e-07 reg 3.928571e+04 train accuracy: 0.313306 val accuracy: 0.320000
lr 4.428571e-07 reg 4.107143e+04 train accuracy: 0.305612 val accuracy: 0.317000
lr 4.428571e-07 reg 4.285714e+04 train accuracy: 0.296980 val accuracy: 0.314000
lr 4.428571e-07 reg 4.464286e+04 train accuracy: 0.300796 val accuracy: 0.314000
lr 4.428571e-07 reg 4.642857e+04 train accuracy: 0.310429 val accuracy: 0.323000
lr 4.428571e-07 reg 4.821429e+04 train accuracy: 0.298020 val accuracy: 0.316000
lr 4.428571e-07 reg 5.000000e+04 train accuracy: 0.304510 val accuracy: 0.314000
lr 4.714286e-07 reg 2.500000e+04 train accuracy: 0.330143 val accuracy: 0.347000
lr 4.714286e-07 reg 2.678571e+04 train accuracy: 0.326245 val accuracy: 0.335000
lr 4.714286e-07 reg 2.857143e+04 train accuracy: 0.319143 val accuracy: 0.333000
lr 4.714286e-07 reg 3.035714e+04 train accuracy: 0.321102 val accuracy: 0.329000
lr 4.714286e-07 reg 3.214286e+04 train accuracy: 0.316816 val accuracy: 0.328000
lr 4.714286e-07 reg 3.392857e+04 train accuracy: 0.310673 val accuracy: 0.332000
lr 4.714286e-07 reg 3.571429e+04 train accuracy: 0.318490 val accuracy: 0.326000
lr 4.714286e-07 reg 3.750000e+04 train accuracy: 0.306735 val accuracy: 0.304000
lr 4.714286e-07 reg 3.928571e+04 train accuracy: 0.311633 val accuracy: 0.324000
lr 4.714286e-07 reg 4.107143e+04 train accuracy: 0.306327 val accuracy: 0.309000
lr 4.714286e-07 reg 4.285714e+04 train accuracy: 0.307020 val accuracy: 0.316000
lr 4.714286e-07 reg 4.464286e+04 train accuracy: 0.310020 val accuracy: 0.329000
lr 4.714286e-07 reg 4.642857e+04 train accuracy: 0.295694 val accuracy: 0.316000
lr 4.714286e-07 reg 4.821429e+04 train accuracy: 0.296306 val accuracy: 0.314000
lr 4.714286e-07 reg 5.000000e+04 train accuracy: 0.294714 val accuracy: 0.308000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.331388 val accuracy: 0.340000
lr 5.000000e-07 reg 2.678571e+04 train accuracy: 0.316245 val accuracy: 0.332000
lr 5.000000e-07 reg 2.857143e+04 train accuracy: 0.321633 val accuracy: 0.334000
lr 5.000000e-07 reg 3.035714e+04 train accuracy: 0.319653 val accuracy: 0.334000
lr 5.000000e-07 reg 3.214286e+04 train accuracy: 0.321224 val accuracy: 0.337000
lr 5.000000e-07 reg 3.392857e+04 train accuracy: 0.322429 val accuracy: 0.333000
lr 5.000000e-07 reg 3.571429e+04 train accuracy: 0.311204 val accuracy: 0.325000
lr 5.000000e-07 reg 3.750000e+04 train accuracy: 0.301408 val accuracy: 0.315000
lr 5.000000e-07 reg 3.928571e+04 train accuracy: 0.305286 val accuracy: 0.320000
lr 5.000000e-07 reg 4.107143e+04 train accuracy: 0.319429 val accuracy: 0.336000
```

```
lr 5.000000e-07 reg 4.285714e+04 train accuracy: 0.304490 val accuracy: 0.310000
lr 5.000000e-07 reg 4.464286e+04 train accuracy: 0.299959 val accuracy: 0.304000
lr 5.000000e-07 reg 4.642857e+04 train accuracy: 0.309592 val accuracy: 0.323000
lr 5.000000e-07 reg 4.821429e+04 train accuracy: 0.308531 val accuracy: 0.317000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.310041 val accuracy: 0.319000
best validation accuracy achieved during cross-validation: 0.352000
```

```
# 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: 0.348000
```

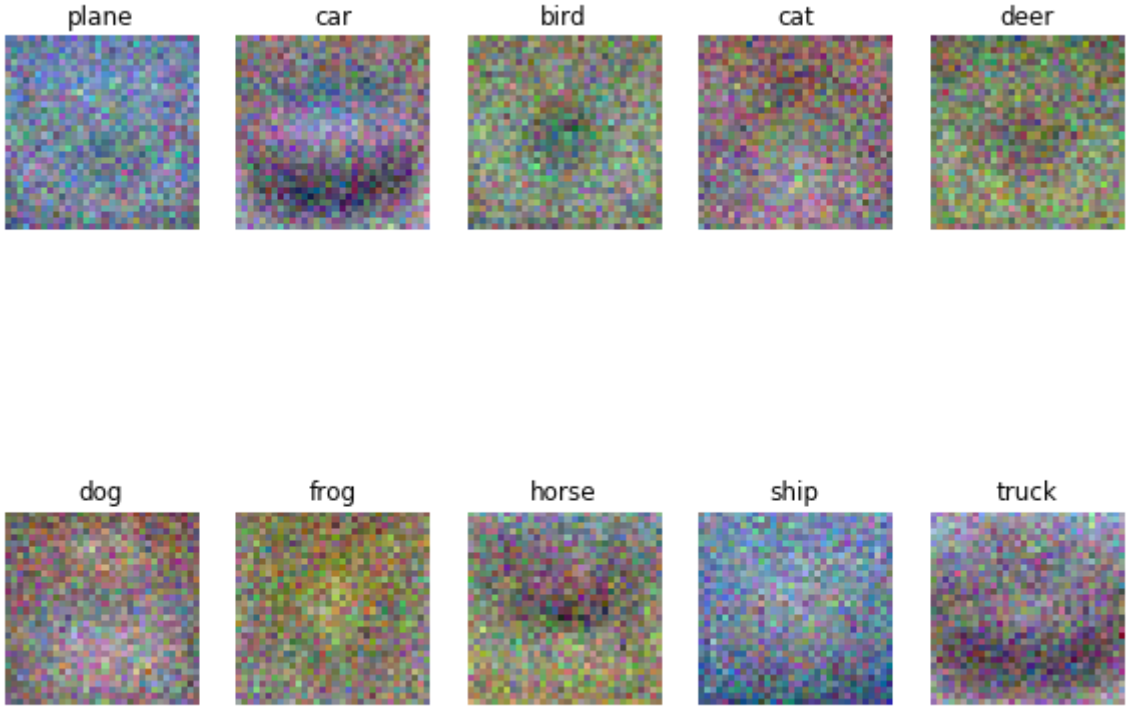
Inline Question 2 - *True or False*

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.

$\mathit{Your Answer}$: True.

$\mathit{Your Explanation}$: For loss in SVM can be zero when $(s_j - s_{yi} + \triangle) < 0$. But loss in softmax will never be zero.

```
# 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

Implementing a Neural Network

In this exercise we will develop a neural network with fully-connected layers to perform classification, and test it out on the CIFAR-10 dataset.

```
# A bit of setup

import numpy as np
import matplotlib.pyplot as plt

from cs231n.classifiers.neural_net import TwoLayerNet

%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 external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-
ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

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

We will use the class `TwoLayerNet` in the file `cs231n/classifiers/neural_net.py` to represent instances of our network. The network parameters are stored in the instance variable `self.params` where keys are string parameter names and values are numpy arrays. Below, we initialize toy data and a toy model that we will use to develop your implementation.

```
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.

input_size = 4
hidden_size = 10
num_classes = 3
num_inputs = 5

def init_toy_model():
    np.random.seed(0)
    return TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)

def init_toy_data():
    np.random.seed(1)
    X = 10 * np.random.randn(num_inputs, input_size)
```

```
    y = np.array([0, 1, 2, 2, 1])
    return X, y

net = init_toy_model()
X, y = init_toy_data()
```

Forward pass: compute scores

Open the file `cs231n/classifiers/neural_net.py` and look at the method `TwoLayerNet.loss`. This function is very similar to the loss functions you have written for the SVM and Softmax exercises: It takes the data and weights and computes the class scores, the loss, and the gradients on the parameters.

Implement the first part of the forward pass which uses the weights and biases to compute the scores for all inputs.

```
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
  [-0.81233741, -1.27654624, -0.70335995],
  [-0.17129677, -1.18803311, -0.47310444],
  [-0.51590475, -1.01354314, -0.8504215 ],
  [-0.15419291, -0.48629638, -0.52901952],
  [-0.00618733, -0.12435261, -0.15226949]])
print(correct_scores)
print()

# The difference should be very small. We get < 1e-7
print('Difference between your scores and correct scores:')
print(np.sum(np.abs(scores - correct_scores)))
```

```
Your scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

correct scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

Difference between your scores and correct scores:
3.6802720745909845e-08
```

Forward pass: compute loss

In the same function, implement the second part that computes the data and regularization loss.

```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.30378789133

# should be very small, we get < 1e-12
print('Difference between your loss and correct loss:')
print(np.sum(np.abs(loss - correct_loss)))
```

```
Difference between your loss and correct loss:
1.7985612998927536e-13
```

Backward pass

Implement the rest of the function. This will compute the gradient of the loss with respect to the variables `w1`, `b1`, `w2`, and `b2`. Now that you (hopefully!) have a correctly implemented forward pass, you can debug your backward pass using a numeric gradient check:

```
from cs231n.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward
pass.
# If your implementation is correct, the difference between the numeric and
# analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b2.

loss, grads = net.loss(X, y, reg=0.05)

# these should all be less than 1e-8 or so
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.05)[0]
    param_grad_num = eval_numerical_gradient(f, net.params[param_name],
verbose=False)
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num,
grads[param_name])))
```

```
b2 max relative error: 1.276034e-10
W2 max relative error: 1.000000e+00
b1 max relative error: 2.738421e-09
W1 max relative error: 3.561318e-09
```

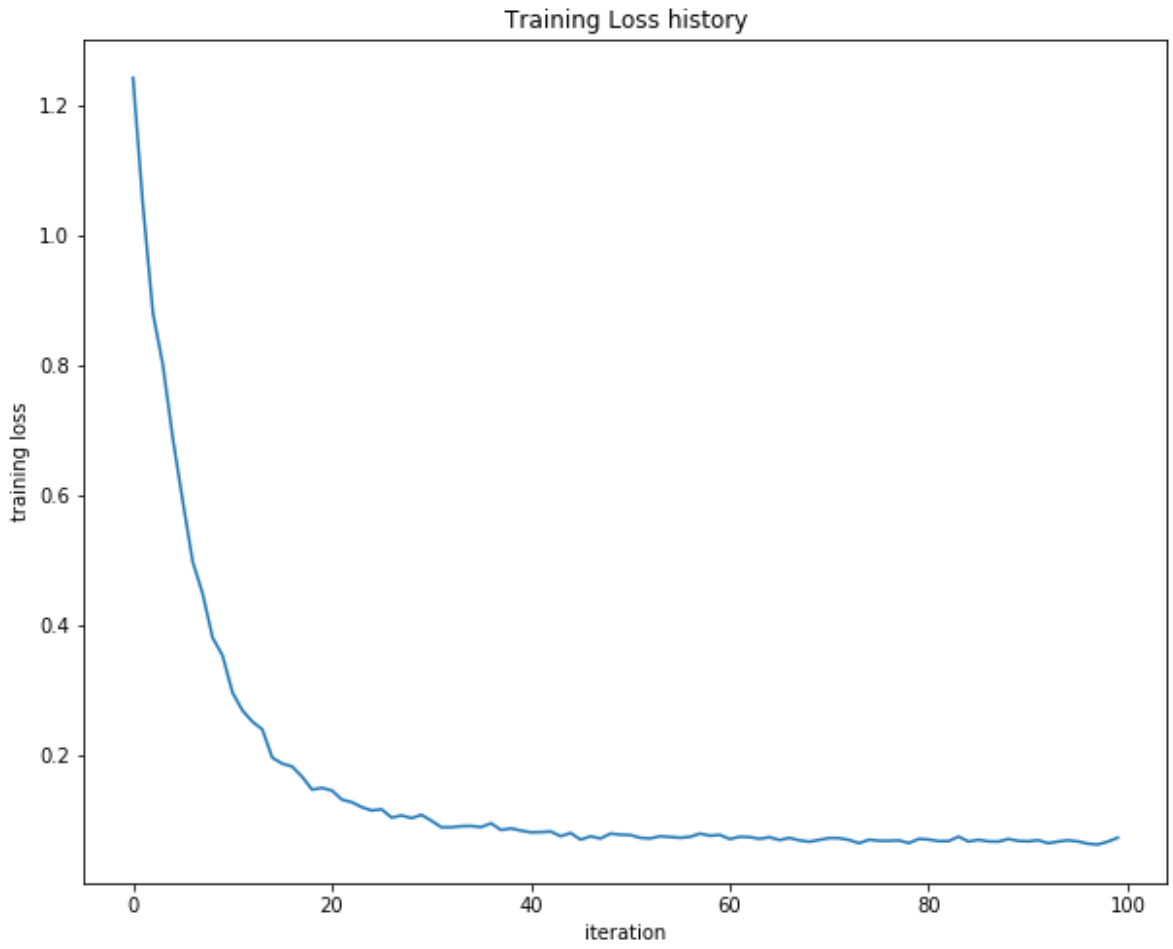
Train the network

To train the network we will use stochastic gradient descent (SGD), similar to the SVM and Softmax classifiers. Look at the function `TwoLayerNet.train` and fill in the missing sections to implement the training procedure. This should be very similar to the training procedure you used for the SVM and Softmax classifiers. You will also have to implement `TwoLayerNet.predict`, as the training process periodically performs prediction to keep track of accuracy over time while the network trains.

Once you have implemented the method, run the code below to train a two-layer network on toy data. You should achieve a training loss less than 0.02.

```
net = init_toy_model()
stats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, verbose=False)

print('Final training loss: ', stats['loss_history'][-1])

# plot the loss history
plt.plot(stats['loss_history'])
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.title('Training Loss history')
plt.show()
```

```
Final training loss:  0.07350642483108832
```



Load the data

Now that you have implemented a two-layer network that passes gradient checks and works on toy data, it's time to load up our favorite CIFAR-10 data so we can use it to train a classifier on a real dataset.

```
from cs231n.data_utils import load_CIFAR10
```

```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the two-layer neural net 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'

    # 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

    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]

    # 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

    # Reshape data to rows
    X_train = X_train.reshape(num_training, -1)
    X_val = X_val.reshape(num_validation, -1)
    X_test = X_test.reshape(num_test, -1)

    return X_train, y_train, X_val, y_val, X_test, y_test


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test = 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)
```

```
Train data shape:  (49000, 3072)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3072)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3072)
Test labels shape:  (1000,)
```

Train a network

To train our network we will use SGD. In addition, we will adjust the learning rate with an exponential learning rate schedule as optimization proceeds; after each epoch, we will reduce the learning rate by multiplying it by a decay rate.

```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=1000, batch_size=200,
            learning_rate=1e-4, learning_rate_decay=0.95,
            reg=0.25, verbose=True)

# Predict on the validation set
val_acc = (net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
iteration 0 / 1000: loss 2.302954
iteration 100 / 1000: loss 2.302069
iteration 200 / 1000: loss 2.295918
iteration 300 / 1000: loss 2.282984
iteration 400 / 1000: loss 2.263426
iteration 500 / 1000: loss 2.232446
iteration 600 / 1000: loss 2.192886
iteration 700 / 1000: loss 2.176924
iteration 800 / 1000: loss 2.164193
iteration 900 / 1000: loss 2.147833
Validation accuracy:  0.21
```

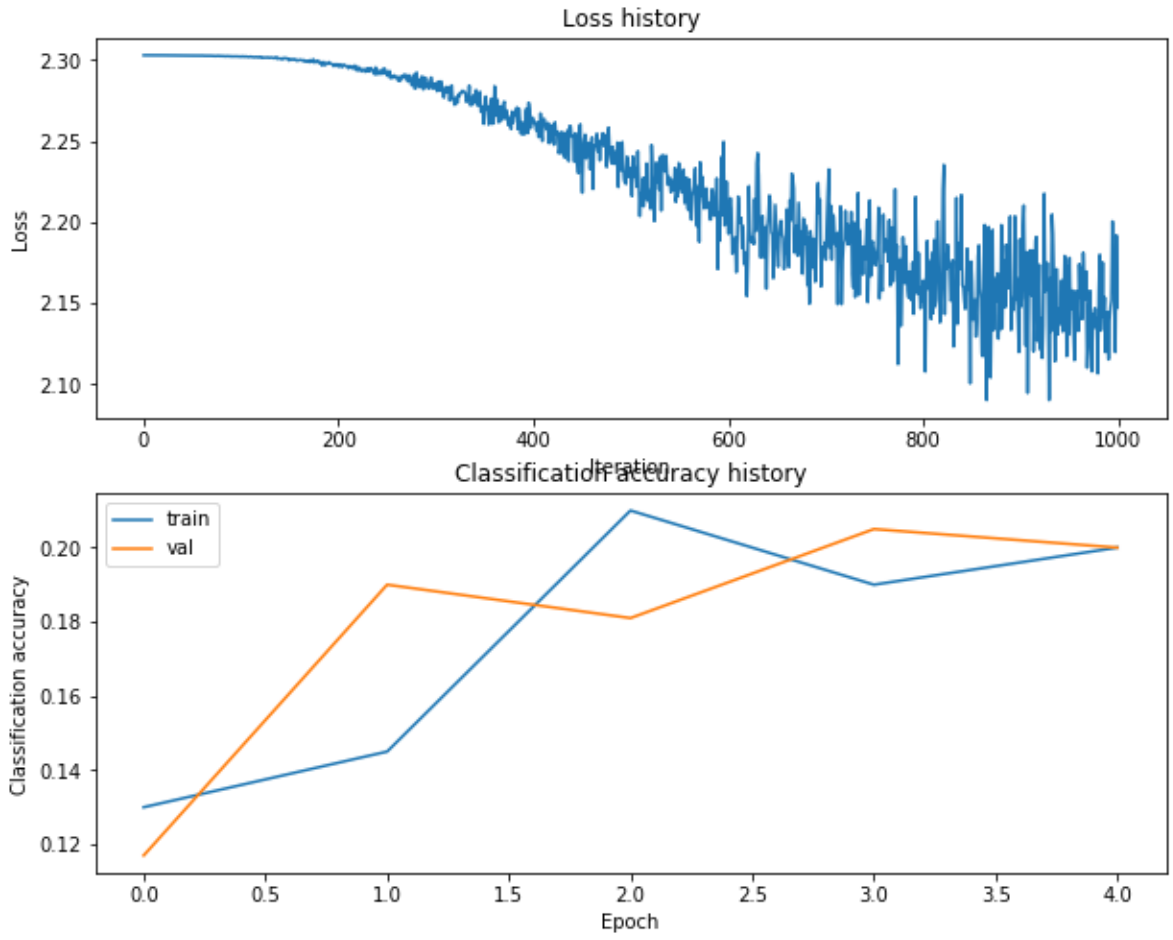
Debug the training

With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

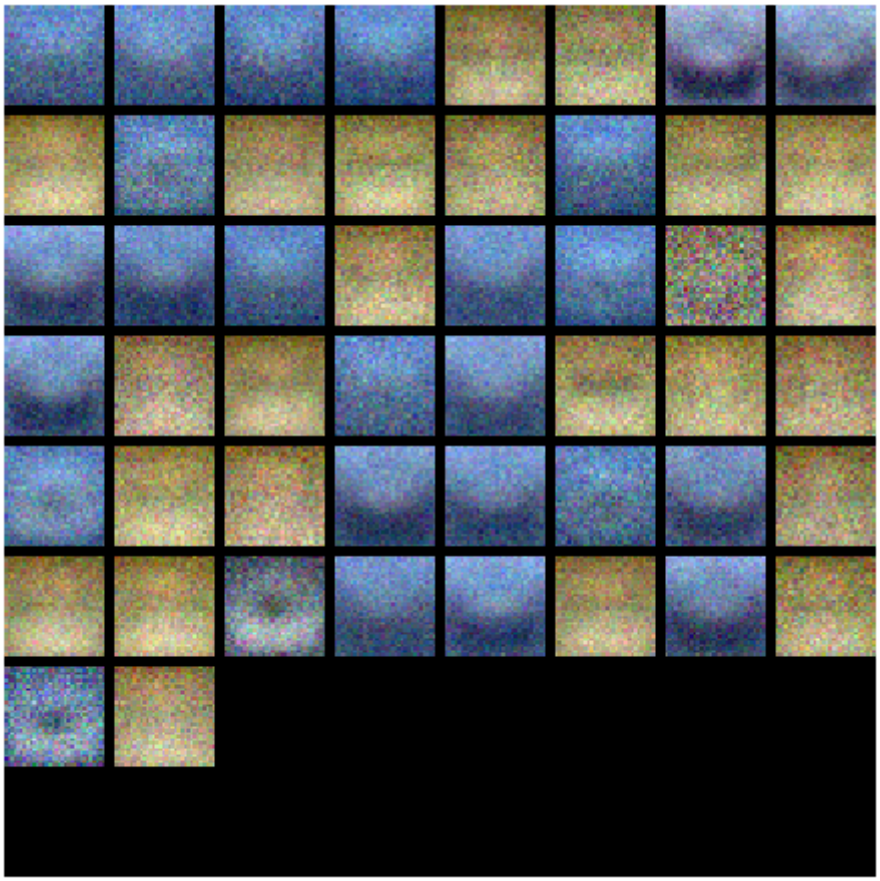
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

```
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
plt.plot(stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(2, 1, 2)
plt.plot(stats['train_acc_history'], label='train')
plt.plot(stats['val_acc_history'], label='val')
plt.title('Classification accuracy history')
plt.xlabel('Epoch')
plt.ylabel('Classification accuracy')
plt.legend()
plt.show()
```



```
from cs231n.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.reshape(32, 32, 3, -1).transpose(3, 0, 1, 2)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()

show_net_weights(net)
```



Tune your hyperparameters

What's wrong?. Looking at the visualizations above, we see that the loss is decreasing more or less linearly, which seems to suggest that the learning rate may be too low. Moreover, there is no gap between the training and validation accuracy, suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

Tuning. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, numer of training epochs, and regularization strength. You might also consider tuning the learning rate decay, but you should be able to get good performance using the default value.

Approximate results. You should be aim to achieve a classification accuracy of greater than 48% on the validation set. Our best network gets over 52% on the validation set.

Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can (52% could serve as a reference), with a fully-connected Neural Network. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

Explain your hyperparameter tuning process below.

Your Answer :

Hidden size: I have tried hidden size for 60,80,100,128,150,200. When hidden size is too small, the accuracy gets decreased. When it was set too large, the gap between train and validation accuracy became big. So I choose 128 finally.

Learning rate: From the initial learning rate, I set it as median and expand the range to search, such as 1.0e-4 -> [0.01e-4, 1.0e-4, 2.0e-4]. Then if I find 2.0e-4 works better, I will set it as a new median. Repeat until the accuracy did't have a big change.

Regularizations: The same as learning rate process.

Num iters: If it is too small may cause unconverge of loss function, but it will train slowly when it is too large. I try 1000,1500 and 2000. Finally I choose 1500.

Batch size: The same as above process.

```
best_net = None # store the best model into this

################################################################################
#
# TODO: Tune hyperparameters using the validation set. Store your best trained
#
# model in best_net.
#
#
#
# To help debug your network, it may help to use visualizations similar to the
#
# ones we used above; these visualizations will have significant qualitative
#
# differences from the ones we saw above for the poorly tuned network.
#
#
#
# Tweaking hyperparameters by hand can be fun, but you might find it useful to
#
# write code to sweep through possible combinations of hyperparameters
#
```

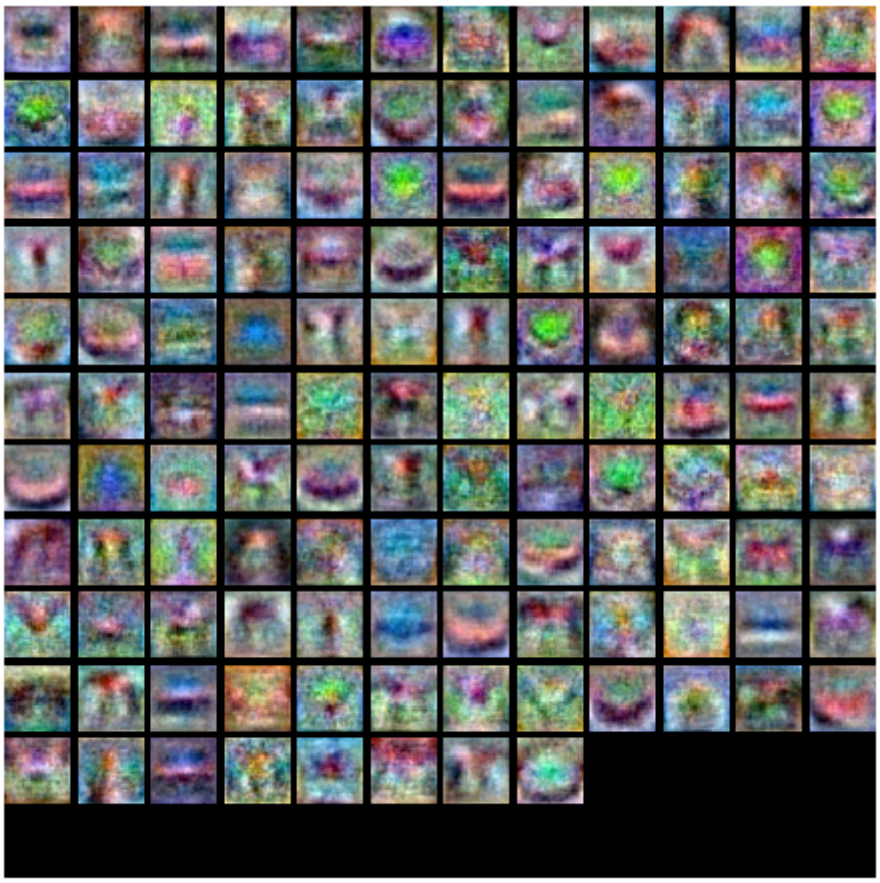
```
# automatically like we did on the previous exercises.
#
################################################################################
#
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
results = {}
best_val_acc = 0

best_val = -1
input_size = 32 * 32 * 3
hidden_size = 128 #hidden_size = [150x, 100 *, 128x, gap when more than 90]
num_classes = 10

net = TwoLayerNet(input_size, hidden_size, num_classes)
for lr in [3.5e-3, 4.8e-3, 4.5e-3, 5.0e-3]:
    for reg in [0.05, 0.1, 0.15]:
        stats = net.train(X_train, y_train, X_val, y_val,
                          num_iters = 1500, batch_size = 200, #1500 batchsize
200 300*??
                          learning_rate=lr, learning_rate_decay=0.95,
                          reg=reg, verbose=False)
        val_acc = (net.predict(X_val) == y_val).mean()
        if val_acc > best_val:
            best_val = val_acc
            best_net = net
        print ("learing_rates: ",lr, "regularizations: ", reg, "val accuracy:",
val_acc)
print ("Best validation accuracy: ", best_val)

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
learing_rates:  0.0035 regularizations:  0.05 val accuracy: 0.461
learing_rates:  0.0035 regularizations:  0.1 val accuracy: 0.482
learing_rates:  0.0035 regularizations:  0.15 val accuracy: 0.47
learing_rates:  0.0048 regularizations:  0.05 val accuracy: 0.5
learing_rates:  0.0048 regularizations:  0.1 val accuracy: 0.484
learing_rates:  0.0048 regularizations:  0.15 val accuracy: 0.482
learing_rates:  0.0045 regularizations:  0.05 val accuracy: 0.494
learing_rates:  0.0045 regularizations:  0.1 val accuracy: 0.496
learing_rates:  0.0045 regularizations:  0.15 val accuracy: 0.475
learing_rates:  0.005 regularizations:  0.05 val accuracy: 0.493
learing_rates:  0.005 regularizations:  0.1 val accuracy: 0.468
learing_rates:  0.005 regularizations:  0.15 val accuracy: 0.48
Best validation accuracy:  0.5
```

```
# Print your validation accuracy: this should be above 48%
val_acc = (best_net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
Validation accuracy:  0.48
```

```
# Visualize the weights of the best network
show_net_weights(best_net)
```



Run on the test set

When you are done experimenting, you should evaluate your final trained network on the test set; you should get above 48%.

```
# Print your test accuracy: this should be above 48%
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.483
```

Inline Question

Now that you have trained a Neural Network classifier, you may find that your testing accuracy is much lower than the training accuracy. In what ways can we decrease this gap? Select all that apply.

1. Train on a larger dataset.
2. Add more hidden units.
3. Increase the regularization strength.

4. None of the above.

$Your Answer$: 1,3

$Your Explanation$: When testing accuracy is much lower than the training accuracy, I think overfitting has occurred. For 1 can increase diversity of dataset. For 3 can reduce overfitting too. But too much hidden units may cause overfitting(2).

Image features 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.

We have seen that we can achieve reasonable performance on an image classification task by training a linear classifier on the pixels of the input image. In this exercise we will show that we can improve our classification performance by training linear classifiers not on raw pixels but on features that are computed from the raw pixels.

All of your work for this exercise will be done in this notebook.

```
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
```

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

Load data

Similar to previous exercises, we will load CIFAR-10 data from disk.

```
from cs231n.features import color_histogram_hsv, hog_feature

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    # 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 issue)
    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)

    # 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]

    return X_train, y_train, X_val, y_val, X_test, y_test

X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
```

Extract Features

For each image we will compute a Histogram of Oriented Gradients (HOG) as well as a color histogram using the hue channel in HSV color space. We form our final feature vector for each image by concatenating the HOG and color histogram feature vectors.

Roughly speaking, HOG should capture the texture of the image while ignoring color information, and the color histogram represents the color of the input image while ignoring texture. As a result, we expect that using both together ought to work better than using either alone. Verifying this assumption would be a good thing to try for your own interest.

The `hog_feature` and `color_histogram_hsv` functions both operate on a single image and return a feature vector for that image. The extract_features function takes a set of images and a list of feature functions and evaluates each feature function on each image, storing the results in a matrix where each column is the concatenation of all feature vectors for a single image.

```
from cs231n.features import *

num_color_bins = 10 # Number of bins in the color histogram
feature_fns = [hog_feature, lambda img: color_histogram_hsv(img,
nbin=num_color_bins)]
X_train_feats = extract_features(X_train, feature_fns, verbose=True)
X_val_feats = extract_features(X_val, feature_fns)
X_test_feats = extract_features(X_test, feature_fns)

# Preprocessing: Subtract the mean feature
mean_feat = np.mean(X_train_feats, axis=0, keepdims=True)
X_train_feats -= mean_feat
X_val_feats -= mean_feat
X_test_feats -= mean_feat

# Preprocessing: Divide by standard deviation. This ensures that each feature
# has roughly the same scale.
```

```
std_feat = np.std(X_train_feats, axis=0, keepdims=True)
X_train_feats /= std_feat
X_val_feats /= std_feat
X_test_feats /= std_feat

# Preprocessing: Add a bias dimension
X_train_feats = np.hstack([X_train_feats, np.ones((X_train_feats.shape[0], 1))])
X_val_feats = np.hstack([X_val_feats, np.ones((X_val_feats.shape[0], 1))])
X_test_feats = np.hstack([X_test_feats, np.ones((X_test_feats.shape[0], 1))])
```

```
Done extracting features for 1000 / 49000 images
Done extracting features for 2000 / 49000 images
Done extracting features for 3000 / 49000 images
Done extracting features for 4000 / 49000 images
Done extracting features for 5000 / 49000 images
Done extracting features for 6000 / 49000 images
Done extracting features for 7000 / 49000 images
Done extracting features for 8000 / 49000 images
Done extracting features for 9000 / 49000 images
Done extracting features for 10000 / 49000 images
Done extracting features for 11000 / 49000 images
Done extracting features for 12000 / 49000 images
Done extracting features for 13000 / 49000 images
Done extracting features for 14000 / 49000 images
Done extracting features for 15000 / 49000 images
Done extracting features for 16000 / 49000 images
Done extracting features for 17000 / 49000 images
Done extracting features for 18000 / 49000 images
Done extracting features for 19000 / 49000 images
Done extracting features for 20000 / 49000 images
Done extracting features for 21000 / 49000 images
Done extracting features for 22000 / 49000 images
Done extracting features for 23000 / 49000 images
Done extracting features for 24000 / 49000 images
Done extracting features for 25000 / 49000 images
Done extracting features for 26000 / 49000 images
Done extracting features for 27000 / 49000 images
Done extracting features for 28000 / 49000 images
Done extracting features for 29000 / 49000 images
Done extracting features for 30000 / 49000 images
Done extracting features for 31000 / 49000 images
Done extracting features for 32000 / 49000 images
Done extracting features for 33000 / 49000 images
Done extracting features for 34000 / 49000 images
Done extracting features for 35000 / 49000 images
Done extracting features for 36000 / 49000 images
Done extracting features for 37000 / 49000 images
Done extracting features for 38000 / 49000 images
Done extracting features for 39000 / 49000 images
Done extracting features for 40000 / 49000 images
Done extracting features for 41000 / 49000 images
Done extracting features for 42000 / 49000 images
Done extracting features for 43000 / 49000 images
Done extracting features for 44000 / 49000 images
Done extracting features for 45000 / 49000 images
```

```
Done extracting features for 46000 / 49000 images
Done extracting features for 47000 / 49000 images
Done extracting features for 48000 / 49000 images
Done extracting features for 49000 / 49000 images
```

Train SVM on features

Using the multiclass SVM code developed earlier in the assignment, train SVMs on top of the features extracted above; this should achieve better results than training SVMs directly on top of raw pixels.

```
# Use the validation set to tune the learning rate and regularization strength

from cs231n.classifiers.linear_classifier import LinearSVM

learning_rates = [1e-9, 1e-8, 1e-7]
regularization_strengths = [5e4, 5e5, 5e6]

results = {}
best_val = -1
best_svm = None

################################################################################
# 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 classifer in best_svm. You might also want to play          #
# with different numbers of bins in the color histogram. If you are careful    #
# you should be able to get accuracy of near 0.44 on the validation set.       #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for lr in learning_rates:
    for reg in regularization_strengths:
        svm = LinearSVM()
        svm.train(X_train_feats, y_train, learning_rate = lr, reg = reg,
num_iters = 1500)
        pred = svm.predict(X_val_feats)
        acc = np.mean(pred == y_val)
        if acc > best_val:
            best_val = acc
            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)
```

```
best validation accuracy achieved during cross-validation: 0.428000
```

```
# Evaluate your trained SVM on the test set: you should be able to get at least
0.40
y_test_pred = best_svm.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print(test_accuracy)
```

```
0.422
```

```
# An important way to gain intuition about how an algorithm works is to
# visualize the mistakes that it makes. In this visualization, we show examples
# of images that are misclassified by our current system. The first column
# shows images that our system labeled as "plane" but whose true label is
# something other than "plane".

examples_per_class = 8
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
'ship', 'truck']
for cls, cls_name in enumerate(classes):
    idxs = np.where((y_test != cls) & (y_test_pred == cls))[0]
    idxs = np.random.choice(idxs, examples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt.subplot(examples_per_class, len(classes), i * len(classes) + cls +
1)
        plt.imshow(X_test[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls_name)
plt.show()
```

Inline question 1:

Describe the misclassification results that you see. Do they make sense?

Your Answer : There are many misclassifications. Yes, because car, cat, dog, horse and so on, they all have four supports underground(leg, wheel), which may be seen as a feature.

Neural Network on image features

Earlier in this assigment we saw that training a two-layer neural network on raw pixels achieved better classification performance than linear classifiers on raw pixels. In this notebook we have seen that linear classifiers on image features outperform linear classifiers on raw pixels.

For completeness, we should also try training a neural network on image features. This approach should outperform all previous approaches: you should easily be able to achieve over 55% classification accuracy on the test set; our best model achieves about 60% classification accuracy.

```
# Preprocessing: Remove the bias dimension
# Make sure to run this cell only ONCE
print(X_train_feats.shape)
X_train_feats = X_train_feats[:, :-1]
X_val_feats = X_val_feats[:, :-1]
X_test_feats = X_test_feats[:, :-1]

print(X_train_feats.shape)
```

```
(49000, 155)
(49000, 154)
```

```
from cs231n.classifiers.neural_net import TwoLayerNet

input_dim = X_train_feats.shape[1]
hidden_dim = 500
num_classes = 10

net = TwoLayerNet(input_dim, hidden_dim, num_classes)
best_net = None

################################################################################
# TODO: Train a two-layer neural network on image features. You may want to    #
# cross-validate various parameters as in previous sections. Store your best   #
# model in the best_net variable.                                              #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
# net = TwoLayerNet(input_size, hidden_size, num_classes)
# for lr in [3.5e-3, 4.8e-3, 4.5e-3, 5.0e-3]:
#     for reg in [0.05, 0.1, 0.15]:
#         stats = net.train(X_train, y_train, X_val, y_val,
#                           num_iters = 1500, batch_size = 200, #1500 batchsize
200 300*??
#                           learning_rate=lr, learning_rate_decay=0.95,
#                           reg=reg, verbose=False)
#         val_acc = (net.predict(X_val) == y_val).mean()
#         if val_acc > best_val:
#             best_val = val_acc
#             best_net = net
#         print ("learing_rates: ",lr, "regularizations: ", reg, "val accuracy:",
val_acc)

net.train(X_train_feats, y_train, X_val_feats, y_val,
          learning_rate=0.7, learning_rate_decay=0.95,
          reg=2.5e-4, num_iters=2000, batch_size=200, verbose=True)

acc = (net.predict(X_val_feats) == y_val).mean()
print('Best validation accuracy: ', acc)
best_net = net
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
iteration 0 / 2000: loss 2.302585
iteration 100 / 2000: loss 1.520260
iteration 200 / 2000: loss 1.292056
iteration 300 / 2000: loss 1.297558
iteration 400 / 2000: loss 1.207266
iteration 500 / 2000: loss 1.193181
iteration 600 / 2000: loss 1.241948
iteration 700 / 2000: loss 1.191374
iteration 800 / 2000: loss 1.031217
iteration 900 / 2000: loss 0.991403
iteration 1000 / 2000: loss 1.110569
iteration 1100 / 2000: loss 0.992712
iteration 1200 / 2000: loss 1.059855
iteration 1300 / 2000: loss 0.875875
iteration 1400 / 2000: loss 0.883443
iteration 1500 / 2000: loss 0.934303
iteration 1600 / 2000: loss 1.031493
iteration 1700 / 2000: loss 0.789615
iteration 1800 / 2000: loss 1.045857
iteration 1900 / 2000: loss 0.770237
Best validation accuracy:  0.577
```

```
# Run your best neural net classifier on the test set. You should be able
# to get more than 55% accuracy.

test_acc = (best_net.predict(X_test_feats) == y_test).mean()
print(test_acc)
```

```
0.566
```