CS280 Fall 2021 Assignment 1
Part A

ML Background

September 25, 2021

Name: 袁润洋

Student ID: 48768008

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.

$$KL(p_{emp}\|q) = \int p(x)(\log p(x) - \log q(x))dx$$

$$= \int p(x)\log p(x)dx - \int p(x)\log q(x)dx$$

Since the dataset $D = \{x_1, \cdots, x_n\}$ is given, $p(x)$ is a fixed function.

Then $\int p(x)\log p(x)$ is a constant, named C

$$\Rightarrow KL(p\|q) = C - \int p(x)\log q(x;\hat{\theta})dx$$

$$= C - \int \frac{1}{n}\sum_{i=1}^{n}\delta(x, x_i)\log q(x;\hat{\theta})dx$$

$$= C - \frac{1}{n}\sum_{i=1}^{n}\int \delta(x, x_i)\log q(x;\hat{\theta})dx$$

By the nature of dirac delta function, for any $f(x)$

$$\int \delta(x, a)f(x)dx = f(a)$$

Then $$KL(p\|q) = C - \frac{1}{n}\sum_{i=1}^{n}\int \delta(x, x_i)\log q(x;\hat{\theta})dx$$

$$= C - \frac{1}{n}\sum_{i=1}^{n}\log(x_i;\hat{\theta})$$

$$\operatorname{argmin}_{\hat{\theta}} KL(p\|q(x;\hat{\theta})) = \operatorname{argmin}_{\hat{\theta}} C - \frac{1}{n}\sum_{i=1}^{n}\log(x_i;\hat{\theta})$$

$$= \operatorname{argmax}_{\hat{\theta}} \frac{1}{n}\sum_{i=1}^{n}\log(x_i;\hat{\theta})$$

$$= \hat{\theta}_{MLE}$$

[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 $\sum_k \pi_k = 1$.)

$$\frac{d}{d\mu_k} l(\theta) = \sum_{n=1}^{N} \frac{\frac{dp(x_n|\theta)}{d\mu_k}}{p(x_n|\theta)}$$

$$= \sum_{n=1}^{N} \frac{\frac{dp(x_n|\theta)}{d\mu_k}}{\sum_{k=1}^{K} \pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)}$$

$$\frac{dp(x_n|\theta)}{d\mu_k} = \frac{d\, \pi_k \frac{e^{-\frac{(x_n-\mu_k)^T \Sigma_k^{-1}(x_n-\mu_k)}{2}}}{(\sqrt{2\pi})^d |\Sigma_k|^{\frac{1}{2}}}}{d\mu_k}$$

$$= \frac{d\, -\frac{(x_n-\mu_k)^T \Sigma_k^{-1}(x_n-\mu_k)}{2}}{d\mu_k} \pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)$$

$$= \frac{2\Sigma_k^{-1}(x_n-\mu_k)}{2} \pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)$$

$$= \Sigma_k^{-1}(x_n-\mu_k)\, \pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)$$

Then $\frac{d}{d\mu_k} l(\theta) = \sum_{n=1}^{N} \frac{\Sigma_k^{-1}(x_n - \mu_k)\pi_k N(x_n|\mu_k, \Sigma_k)}{\sum_{k=1}^{K} \pi_k N(x_n|\mu_k, \Sigma_k)}$

Because $r_{nk} = \frac{\pi_k N(x_n|\mu_k, \Sigma_k)}{\sum_{k'=1}^{K} \pi_{k'} N(x|\mu_{k'}, \Sigma_{k'})}$ $\Rightarrow \frac{d}{d\mu_k} l(\theta) = \sum_{n=1}^{N} r_{nk} \Sigma_k^{-1}(x_n - \mu_k)$

2. $\frac{d\, l(\theta)}{d\pi_k} = \sum_{n=1}^{N} \frac{\frac{dp(x_n|\theta)}{d\pi_k}}{p(x_n|\theta)}$

$$= \sum_{n=1}^{N} \frac{\frac{d\pi_k N(x_n|\mu_k, \Sigma_k)}{d\pi_k}}{\sum_{k=1}^{K} \pi_k N(x_n|\mu_k, \Sigma_k)}$$

$$= \sum_{n=1}^{N} \frac{N(x_n|\mu_k, \Sigma_k)}{\sum_{k=1}^{K} \pi_k N(x_n|\mu_{k'}, \Sigma_{k'})}$$

$$= \sum_{n=1}^{N} \frac{r_{nk}}{\pi_k}$$

Bonus: When there is a constraint $\sum_k \pi_k = 1$, the MLE problem is

$$\underset{\theta, \pi}{\text{argmax}} \sum_{n=1}^{N} \log \sum_{k=1}^{K} \pi_k N(x_n|\mu_k, \Sigma_k)$$

$$\text{s.t.} \quad \sum_k \pi_k = 1$$

Applying Lagrange multiplier method, the problem is transformed to

$$\underset{\theta, \pi}{\text{argmax}}\, L(\theta) = \sum_{n=1}^{N} \log \sum_{k=1}^{K} \pi_k N(x_n|\mu_k, \Sigma_k) + \lambda\left(\sum_k \pi_k - 1\right)$$

Then $\frac{d\, L(\theta)}{d\pi_k} = \sum_{n=1}^{N} \frac{r_{nk}}{\pi_k} + \lambda$

MLE requires the gradient at the optimal solution is 0

Then for all $k = 1, \cdots, K$,

$\sum_{n=1}^{N} \frac{r_{nk}}{\pi_k} + \lambda = 0 \Rightarrow \sum_{n=1}^{N} r_{nk} + \pi_k \lambda = 0 \Rightarrow \sum_{k=1}^{K} \sum_{n=1}^{N} r_{nk} + \sum_{k=1}^{K} \pi_k \lambda = 0 \Rightarrow \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} + \sum_{k=1}^{K} \pi_k \lambda = 0$

Because $\sum_{k=1}^{K} r_{nk} = 1$ and $\sum_{k=1}^{K} \pi_k = 1$, the above equation is $\sum_{n=1}^{N} 1 + \lambda = 0 \Rightarrow \lambda = -N$

Bring $\lambda = -N$ into $\sum_{n=1}^{N} \frac{r_{nk}}{\pi_k} + \lambda = 0 \Rightarrow \sum_{n=1}^{N} \frac{r_{nk}}{\pi_k} = N$

Then $\frac{d\, l(\theta)}{d\mu_k} = N$ with constraint $\sum_k \pi_k = 1$

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

#

Using svm loss function

$$L = \sum_{i=1}^{N} max(0, W_i * x_i + b - W_{y_i} * x_i - b) = \sum_{i=1}^{N} max(0, W_i * x_i - W_{y_i} * x_i)$$

$$\frac{dL}{dW_i} = \begin{cases} 0 & L \leq 0 \; or \; other \\ x & if \; i \; is \; the \; wrong \; predicted \; label \\ -x & if \; i \; is \; the \; right \; label \end{cases}$$

$$\frac{dL}{db} = 0$$

In [ ]:

```
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)
```

In [ ]:

```
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]
```

In [ ]:

```
## 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_test.shape)
print(type(y_train))
```

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

In [ ]:

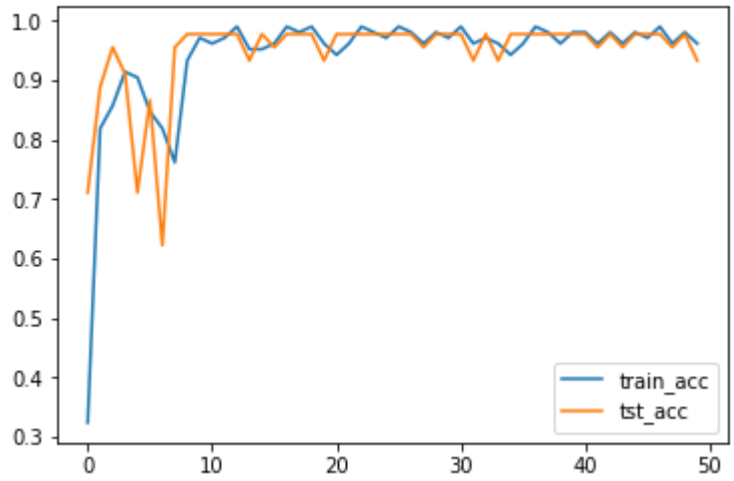
```
class MultiClsPLA(object):

    ## 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_d
imension, 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):
        ########################################
        ##  ToDO: Initialize the weight with   ##
        ##  small std and zero mean gaussian   ##
        ########################################
        weight = np.zeros((num_cls,weight_dimension + 1 ))
        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 = 0

        ###############################################
        ## TODO: to implement the training process  ##
        ## and update the weights                   ##
        ###############################################
        dw = np.zeros_like(self.weight)

        for i in range(X_train.shape[0]):
            scores = np.dot(self.weight, X_train[i].transpose())
            predicted_label = np.argmax(scores)
            if (predicted_label != y_train[i] ):
                dw[predicted_label] += self.lr * X_train[i]
                dw[y_train[i]] -= self.lr * X_train[i]
            else:
                train_acc += 1
        self.weight -= dw
        train_acc /= X_train.shape[0]
        return train_acc

    def test_step(self, X_test, y_test):
```

```
        num_sample = X_test.shape[0]
        test_acc = 0

        ########################################
        ##  ToDO: Evaluate the test set and    ##
        ##  return the test acc                ##
        ########################################
        for i in range(num_sample):
            scores = np.dot(self.weight, X_test[i].transpose())
            predicted_label = np.argmax(scores)
            if (predicted_label == y_test[i]):
                test_acc += 1

        test_acc /= X_test.shape[0]
        return test_acc

    def train(self):

        self.X_train = self.data_preprocessing(data=self.X_train)
        self.X_test = self.data_preprocessing(data=self.X_test)

        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       ###
        ######################################################
        self.X_train = np.concatenate((self.X_train, np.ones((self.X_train.shape
[0],1))), axis=1)
        self.X_test = np.concatenate((self.X_test, np.ones((self.X_test.shape[0
],1))), axis=1)

        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:%.3
f'%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()
```

In [ ]:

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

PLA = MultiClsPLA(X_train, y_train, X_test, y_test, 0.001, 50, 4, 3)

PLA.train()
PLA.vis_acc_curve()

#########################################################
### 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: 0 traing_acc:0.324 tst_acc:0.711
epoch: 1 traing_acc:0.819 tst_acc:0.889
epoch: 2 traing_acc:0.857 tst_acc:0.956
epoch: 3 traing_acc:0.914 tst_acc:0.911
epoch: 4 traing_acc:0.905 tst_acc:0.711
epoch: 5 traing_acc:0.848 tst_acc:0.867
epoch: 6 traing_acc:0.819 tst_acc:0.622
epoch: 7 traing_acc:0.762 tst_acc:0.956
epoch: 8 traing_acc:0.933 tst_acc:0.978
epoch: 9 traing_acc:0.971 tst_acc:0.978
epoch: 10 traing_acc:0.962 tst_acc:0.978
epoch: 11 traing_acc:0.971 tst_acc:0.978
epoch: 12 traing_acc:0.990 tst_acc:0.978
epoch: 13 traing_acc:0.952 tst_acc:0.933
epoch: 14 traing_acc:0.952 tst_acc:0.978
epoch: 15 traing_acc:0.962 tst_acc:0.956
epoch: 16 traing_acc:0.990 tst_acc:0.978
epoch: 17 traing_acc:0.981 tst_acc:0.978
epoch: 18 traing_acc:0.990 tst_acc:0.978
epoch: 19 traing_acc:0.962 tst_acc:0.933
epoch: 20 traing_acc:0.943 tst_acc:0.978
epoch: 21 traing_acc:0.962 tst_acc:0.978
epoch: 22 traing_acc:0.990 tst_acc:0.978
epoch: 23 traing_acc:0.981 tst_acc:0.978
epoch: 24 traing_acc:0.971 tst_acc:0.978
epoch: 25 traing_acc:0.990 tst_acc:0.978
epoch: 26 traing_acc:0.981 tst_acc:0.978
epoch: 27 traing_acc:0.962 tst_acc:0.956
epoch: 28 traing_acc:0.981 tst_acc:0.978
epoch: 29 traing_acc:0.971 tst_acc:0.978
epoch: 30 traing_acc:0.990 tst_acc:0.978
epoch: 31 traing_acc:0.962 tst_acc:0.933
epoch: 32 traing_acc:0.971 tst_acc:0.978
epoch: 33 traing_acc:0.962 tst_acc:0.933
epoch: 34 traing_acc:0.943 tst_acc:0.978
epoch: 35 traing_acc:0.962 tst_acc:0.978
epoch: 36 traing_acc:0.990 tst_acc:0.978
epoch: 37 traing_acc:0.981 tst_acc:0.978
epoch: 38 traing_acc:0.962 tst_acc:0.978
epoch: 39 traing_acc:0.981 tst_acc:0.978
epoch: 40 traing_acc:0.981 tst_acc:0.978
epoch: 41 traing_acc:0.962 tst_acc:0.956
epoch: 42 traing_acc:0.981 tst_acc:0.978
epoch: 43 traing_acc:0.962 tst_acc:0.956
epoch: 44 traing_acc:0.981 tst_acc:0.978
epoch: 45 traing_acc:0.971 tst_acc:0.978
epoch: 46 traing_acc:0.990 tst_acc:0.978
epoch: 47 traing_acc:0.962 tst_acc:0.956
epoch: 48 traing_acc:0.981 tst_acc:0.978
epoch: 49 traing_acc:0.962 tst_acc:0.933
```



In [ ]:

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

In [ ]:

```
# 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 noteboo
k
# 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-ipytho
n
%load_ext autoreload
%autoreload 2
```

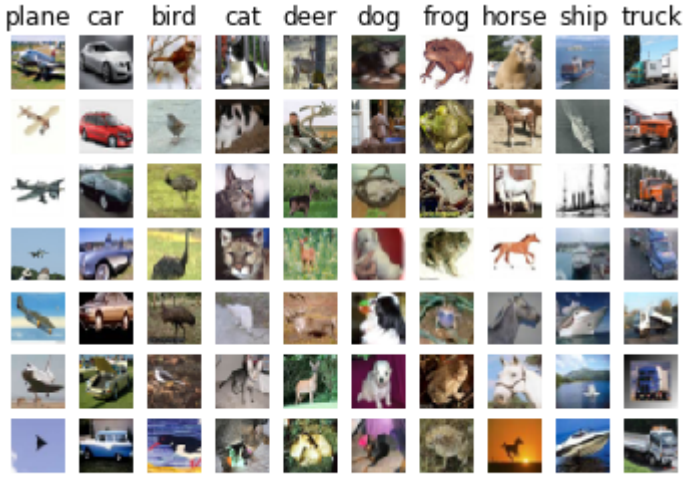
In [ ]:

```
# 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,)
```

In [ ]:

```
# 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()
```



In [ ]:

```
# 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)
```

In [ ]:

```
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.

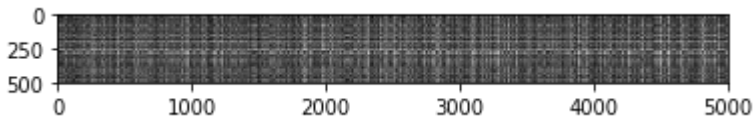
In [ ]:

```
# 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)
```

In [ ]:

```
# 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 :

If one test example is not similar to each train example data, then the row would be distinctly brighter and darker means similar.

If one train example is not similar to each test exampla data, the the column would be distinctly brighter and darker means similar.

In [ ]:

```
# 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`:

In [ ]:

```
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

Your Explanation :

1. Once the training set is fixed, the mean μ_{ij} is fixed which is a constant. When computing the pixel distance $|(p_{ij}^{(k)} - \mu) - (p_{ij}^{(k')} - \mu)| = |p_{ij}^{(k)} - p_{ij}^{(k')}|$, which is unchanged.
2. Because the distance is only computed along the pixel dimension, then pixel at the same location will be subtracted with the same mean, $|(p_{ij}^{(k)} - \mu_{ij}) - (p_{ij}^{(k')} - \mu_{ij})| = |p_{ij}^{(k)} - p_{ij}^{(k')}|$
3. Because the standard deviation is a constant, when all pixel is divided by the same number, the performance would not change.
4. The pixe-wise standart deviation is not the same, which mean the weight of pixel at different location is different, which will influence the performance.
5. Rotating the coordinate axes influence the l1 distance.

In [ ]:

```
# 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, reshap
e
# 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
```

In [ ]:

```
# 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
```

In [ ]:

```
# 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 impl
ementation!

# 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 923.021010 seconds
One loop version took 31.844184 seconds
No loop version took 0.161868 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.

In [ ]:

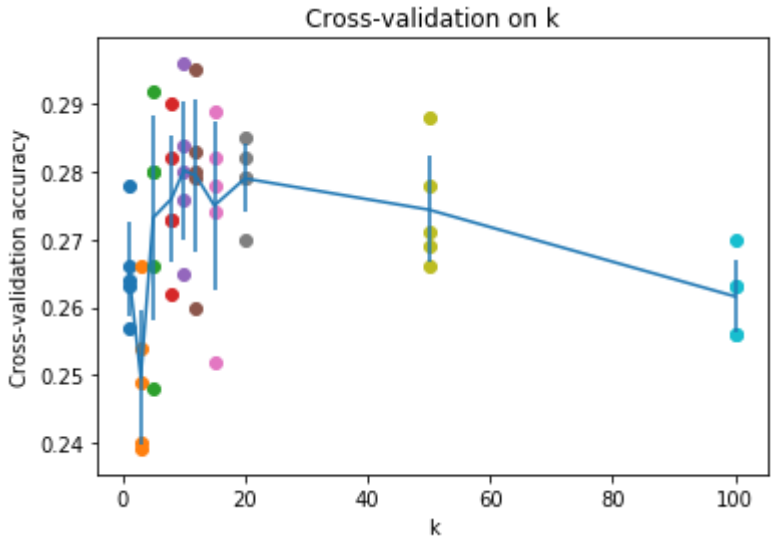
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

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)*****

y_train = y_train.reshape(-1, 1)

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:
    k_to_accuracies[k] = []

for i in range(num_folds):
    classifier = KNearestNeighbor()
    X_vali_train = np.concatenate(X_train_folds[0:i] + X_train_folds[i+1:])
    y_vali_train = np.concatenate(y_train_folds[0:i] + y_train_folds[i+1:])

    y_vali_train = y_vali_train[:, 0]
    classifier.train(X_vali_train, y_vali_train)

    for k in k_choices:
        pred = classifier.predict(X_train_folds[i], k=k)
        accuracy = np.sum(pred == y_train_folds[i][:,0]) / len(y_train_folds[i])
        k_to_accuracies[k].append(accuracy)

y_train = y_train[:,0]

# *****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
```

In [ ]:

```
# 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()
```



In [ ]:

```
# 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 = 10

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 141 / 500 correct => accuracy: 0.282000
```

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.

Your Answer :

2, 4

Your Explanation :

1. We cannot find a hyperplane that separate two different classes in K-NN. Assume we are using 1-NN calssifier, the area belonging to a class is decided by the minimal distance among the close points. The deicision bound seems like a curve that including all the class points.
2. yes, when inputing the training data, 1-NN will always output the correct label which is itself, so the error is 0. But using 5-NN, it may output wrong label, since we use the most label in top 5.
3. Uncertain
4. obviously, when the size of training set grows, one test exaple will compare to more data which causes growing of time.

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 (http://vision.stanford.edu/teaching/cs231n/assignments.html) on the course website.

In this exercise you will:

- implement a fully-vectorized **loss function** for the SVM
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** using numerical gradient
- use a validation set to **tune the learning rate and regularization** strength
- **optimize** the loss function with **SGD**
- **visualize** the final learned weights

In [ ]:

```
# 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-ipytho
n
%load_ext autoreload
%autoreload 2
```

CIFAR-10 Data Loading and Preprocessing

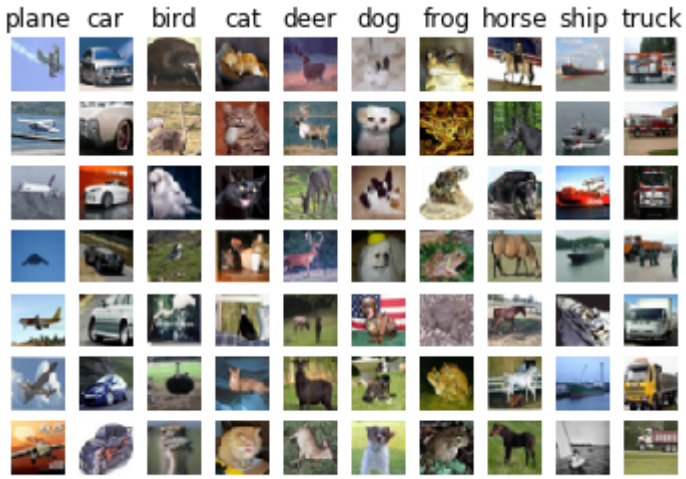
In [ ]:

```
# 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)
```

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

In [ ]:

```
# 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()
```



In [ ]:

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

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

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

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

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

print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

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

In [ ]:

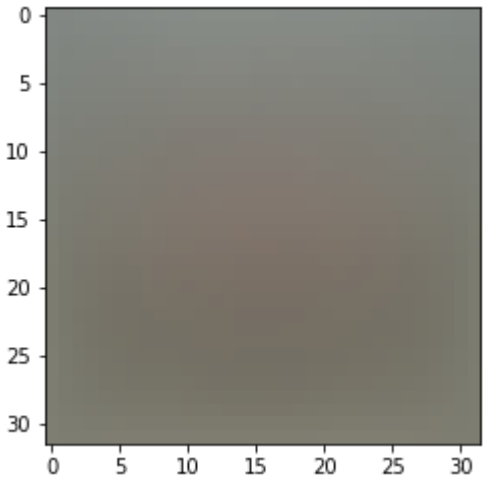
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

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

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

In [ ]:

```
# 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 i
mage
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.

In [ ]:

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

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

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

```
loss: 8.607392
```

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

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

In [ ]:

```
# 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 matc
h
# 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: -8.764384 analytic: -8.715181, relative error: 2.814860e-
03
numerical: 15.051673 analytic: 15.051673, relative error: 3.457964e-
12
numerical: 17.644117 analytic: 17.644117, relative error: 9.269332e-
13
numerical: 5.135621 analytic: 5.135621, relative error: 5.140185e-11
numerical: -45.470843 analytic: -45.470843, relative error: 3.055496
e-12
numerical: 1.531669 analytic: 1.531669, relative error: 3.835962e-10
numerical: 8.170946 analytic: 8.170946, relative error: 1.981267e-12
numerical: 4.379175 analytic: 4.379175, relative error: 2.964195e-12
numerical: -2.366844 analytic: -2.366844, relative error: 7.138572e-
11
numerical: 4.933504 analytic: 4.933504, relative error: 2.953139e-11
numerical: 13.538399 analytic: 13.499303, relative error: 1.445996e-
03
numerical: 16.727553 analytic: 16.696795, relative error: 9.202393e-
04
numerical: -25.199856 analytic: -25.199856, relative error: 1.197969
e-11
numerical: 0.776635 analytic: 0.776635, relative error: 2.952477e-10
numerical: 8.525858 analytic: 8.525858, relative error: 3.122167e-11
numerical: -0.194268 analytic: -0.249790, relative error: 1.250350e-
01
numerical: 19.406481 analytic: 19.406481, relative error: 2.469767e-
11
numerical: -2.258480 analytic: -2.276296, relative error: 3.928725e-
03
numerical: -5.228979 analytic: -5.224056, relative error: 4.709453e-
04
numerical: 3.923384 analytic: 3.923384, relative error: 3.277868e-11
```

Inline Question 1

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

$YourAnswer:$ Yes, The loss function of svm : $max(0, s_j - s_{y_i} + \Delta t)$ is not differentiable at 0. Then at this point there will be different gradient. Given $s_j - s_{y_i} + \Delta t = h/2$, the numerical gradient is $\frac{0+(h/2+h)}{2*h} = \frac{3}{4}$, but the analytic gradient is 1. The method of change the margin affect is that we set a small threshold, any value less than the threshold is treated as 0.

In [ ]:

```
# Next implement the function svm_loss_vectorized; for now only compute the los
s;
# 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 fast
er.
print('difference: %f' % (loss_naive - loss_vectorized))
```

```
Naive loss: 8.607392e+00 computed in 0.330198s
Vectorized loss: 8.607392e+00 computed in 0.003308s
difference: 0.000000
```

In [ ]:

```
# 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 0.396419s
Vectorized loss and gradient: computed in 0.006039s
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` .

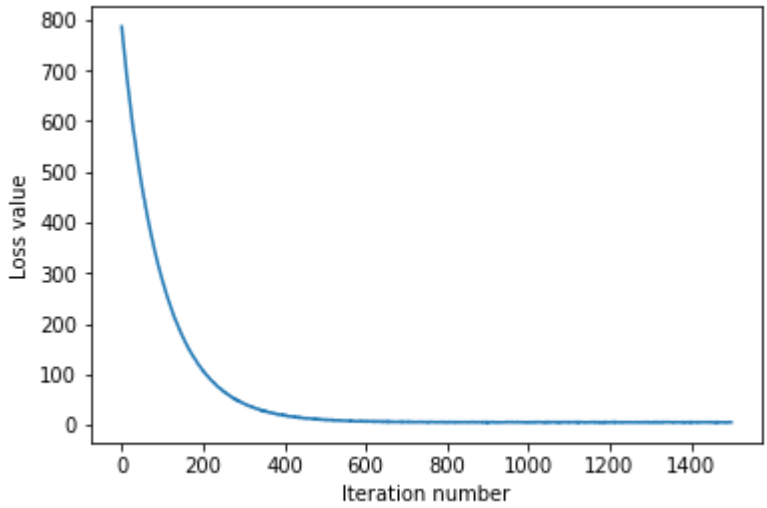
In [ ]:

```
# 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 794.083501
iteration 100 / 1500: loss 291.266574
iteration 200 / 1500: loss 109.617471
iteration 300 / 1500: loss 43.437993
iteration 400 / 1500: loss 19.534533
iteration 500 / 1500: loss 10.190690
iteration 600 / 1500: loss 7.292982
iteration 700 / 1500: loss 5.685529
iteration 800 / 1500: loss 5.604706
iteration 900 / 1500: loss 5.148075
iteration 1000 / 1500: loss 5.278615
iteration 1100 / 1500: loss 5.195671
iteration 1200 / 1500: loss 4.884264
iteration 1300 / 1500: loss 5.464167
iteration 1400 / 1500: loss 5.430266
That took 7.213101s
```

In [ ]:

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



In [ ]:

```
# 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: 0.372918
validation accuracy: 0.380000
```

In [ ]:

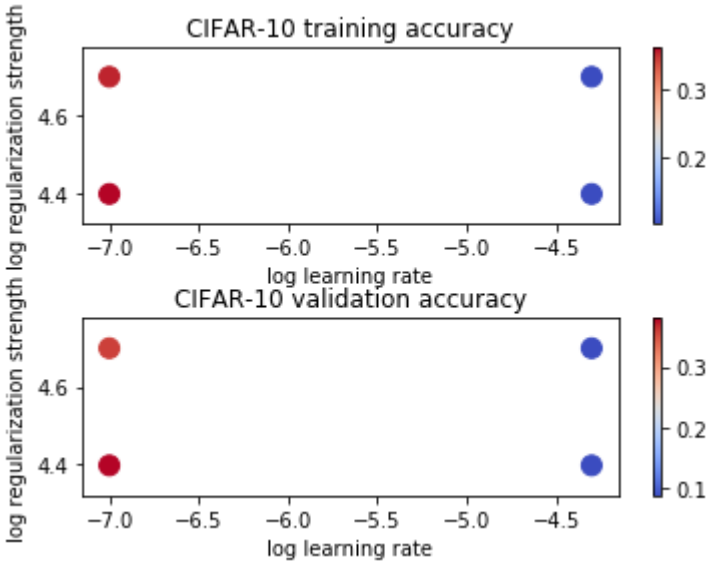
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.39 on the validation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

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

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

for learning_rate in learning_rates:
    for strength in regularization_strengths:
        svm = LinearSVM()
        loss_hist = svm.train(X_train, y_train, learning_rate=learning_rate, reg
=strength, num_iters=2000, verbose=False)

        train_pred = svm.predict(X_train)
        val_pred = svm.predict(X_val)

        train_accuracy = np.sum(y_train == train_pred) / train_pred.shape[0]
        val_accuracy = np.sum(y_val == val_pred) / val_pred.shape[0]
        if (val_accuracy > best_val):
            best_val = val_accuracy
            best_svm = svm

        results[(learning_rate,strength)] = train_accuracy, val_accuracy



# *****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: 0.364714 val accura
cy: 0.381000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.357122 val accura
cy: 0.361000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.100265 val accura
cy: 0.087000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accura
cy: 0.087000
best validation accuracy achieved during cross-validation: 0.381000
```

In [ ]:

```
# 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()
```



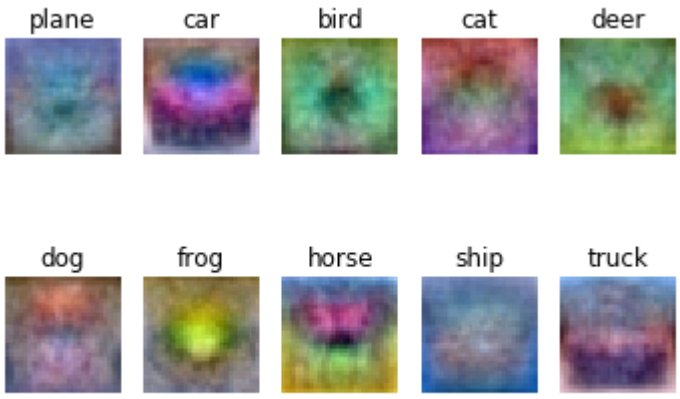
In [ ]:

```
# 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
```

In [ ]:

```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, these m
ay
# 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.

$Your Answer$: The visualized weights look like the average pictures of the training sample. Because the scores are the inner product of weight and the given picture, if they are similar, the score will be generally higher.

Softmax exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the [assignments page (http://vision.stanford.edu/teaching/cs231n/assignments.html)](http://vision.stanford.edu/teaching/cs231n/assignments.html) on the course website.

This exercise is analogous to the SVM exercise. You will:

- implement a fully-vectorized **loss function** for the Softmax classifier
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** with numerical gradient
- use a validation set to **tune the learning rate and regularization** strength
- **optimize** the loss function with **SGD**
- **visualize** the final learned weights

In [ ]:

```
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-ipytho
n
%load_ext autoreload
%autoreload 2
```

In [ ]:

```
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 ca
use 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` .

In [ ]:

```
# 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: 2.370793
sanity check: 2.302585
```

Inline Question 1

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

$Your\ Answer:$

Since the parameter of weight W is uniformly distributed, then probability of predicting the right lable is $\frac{1}{10} = 0.1$. (The number of class is 10). The loss function of softmax is $L = -log(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}) = -log(0.1)$

In [ ]:

```
# 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: -3.584262 analytic: -3.584262, relative error: 1.015706e-
08
numerical: -0.712968 analytic: -0.712968, relative error: 2.469600e-
08
numerical: 1.964334 analytic: 1.964334, relative error: 2.454082e-08
numerical: 0.579612 analytic: 0.579612, relative error: 1.132592e-07
numerical: 2.074250 analytic: 2.074250, relative error: 6.018710e-09
numerical: -0.699314 analytic: -0.699314, relative error: 8.416069e-
08
numerical: 0.915196 analytic: 0.915196, relative error: 9.957081e-08
numerical: -3.078736 analytic: -3.078736, relative error: 4.248062e-
10
numerical: -5.636788 analytic: -5.636788, relative error: 1.017677e-
08
numerical: 0.287363 analytic: 0.287363, relative error: 3.168856e-07
numerical: 0.787721 analytic: 0.787721, relative error: 8.145083e-08
numerical: -2.636386 analytic: -2.636386, relative error: 5.160017e-
09
numerical: 0.468095 analytic: 0.468095, relative error: 4.628773e-08
numerical: 2.650473 analytic: 2.650473, relative error: 2.473295e-08
numerical: 3.169773 analytic: 3.169773, relative error: 2.405508e-08
numerical: 0.984871 analytic: 0.984871, relative error: 1.569300e-08
numerical: -0.866793 analytic: -0.866793, relative error: 4.092280e-
08
numerical: 1.672245 analytic: 1.672245, relative error: 9.048004e-09
numerical: 1.216457 analytic: 1.216457, relative error: 3.640168e-08
numerical: -1.268623 analytic: -1.268623, relative error: 3.700293e-
11
```

In [ ]:

```
# Now that we have a naive implementation of the softmax loss function and its g
radient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version s
hould be
# much faster.
tic = time.time()
loss_naive, grad_naive = softmax_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from cs231n.classifiers.softmax import softmax_loss_vectorized
tic = time.time()
loss_vectorized, grad_vectorized = softmax_loss_vectorized(W, X_dev, y_dev, 0.00
0005)
toc = time.time()
print('vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

# As we did for the SVM, we use the Frobenius norm to compare the two versions
# of the gradient.
grad_difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('Loss difference: %f' % np.abs(loss_naive - loss_vectorized))
print('Gradient difference: %f' % grad_difference)
```

```
naive loss: 2.370793e+00 computed in 0.345840s
vectorized loss: 2.370793e+00 computed in 0.003938s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [ ]:

```
# 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 hyperparamete
rs
learning_rates = [1e-7, 5e-7]
regularization_strengths = [2.5e4, 5e4]

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

for learning_rate in learning_rates:
    for strength in regularization_strengths:
        softmax = Softmax()
        loss_hist = softmax.train(X_train, y_train, learning_rate=learning_rate,
reg=strength, num_iters=2000, verbose=False)

        train_pred = softmax.predict(X_train)
        val_pred = softmax.predict(X_val)

        train_accuracy = np.sum(y_train == train_pred) / train_pred.shape[0]
        val_accuracy = np.sum(y_val == val_pred) / val_pred.shape[0]
        if (val_accuracy > best_val):
            best_val = val_accuracy
            best_softmax = softmax

        results[(learning_rate,strength)] = train_accuracy, val_accuracy

# *****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: 0.328898 val accura
cy: 0.340000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.306673 val accura
cy: 0.319000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.317612 val accura
cy: 0.336000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.299102 val accura
cy: 0.317000
best validation accuracy achieved during cross-validation: 0.340000
```

In [ ]:

```
# 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: 0.351000
```

Inline Question 2 - *True or False*

Suppose the overall training loss is defined as the sum of the per-datapoint loss over all training examples. It is possible to add a new datapoint to a training set that would leave the SVM loss unchanged, but this is not the case with the Softmax classifier loss.

$YourAnswer:$ True

$YourExplanation:$ If we add a data point which is predicted correctly, the loss that this data point bringing in is 0. Then the svm loss won't increase. But in softmax case, there will always be non-zero loss no matter the prediction is right or wrong, according to the loss function.

In [ ]:

```
# 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])
```

In [ ]:

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.

In [ ]:

```
# 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-ipytho
n
%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))))
```

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.

In [ ]:

```
# 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.

In [ ]:

```
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.

In [ ]:

```
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:

In [ ]:

```
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])))
```

```
W2 max relative error: 3.440708e-09
b2 max relative error: 4.447656e-11
W1 max relative error: 3.561318e-09
b1 max relative error: 2.738421e-09
```

Train the network

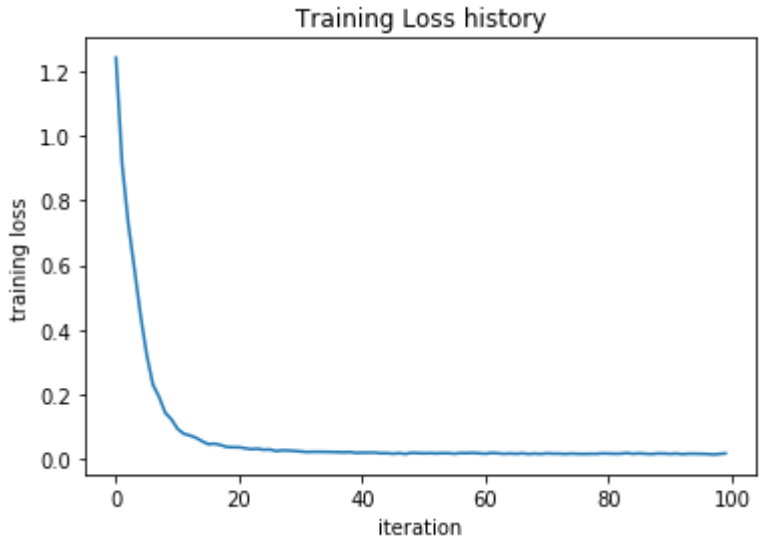
To train the network we will use stochastic gradient descent (SGD), similar to the SVM and Softmax classifiers. Look at the function `TwoLayerNet.train` and fill in the missing sections to implement the training procedure. This should be very similar to the training procedure you used for the SVM and 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.

In [ ]:

```
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:  0.01714960793873208
```



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.

In [ ]:

```
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 ca
use 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.

In [ ]:

```
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 2.302550
iteration 200 / 1000: loss 2.297648
iteration 300 / 1000: loss 2.259602
iteration 400 / 1000: loss 2.204170
iteration 500 / 1000: loss 2.118565
iteration 600 / 1000: loss 2.051535
iteration 700 / 1000: loss 1.988466
iteration 800 / 1000: loss 2.006591
iteration 900 / 1000: loss 1.951473
Validation accuracy:  0.287
```

Debug the training

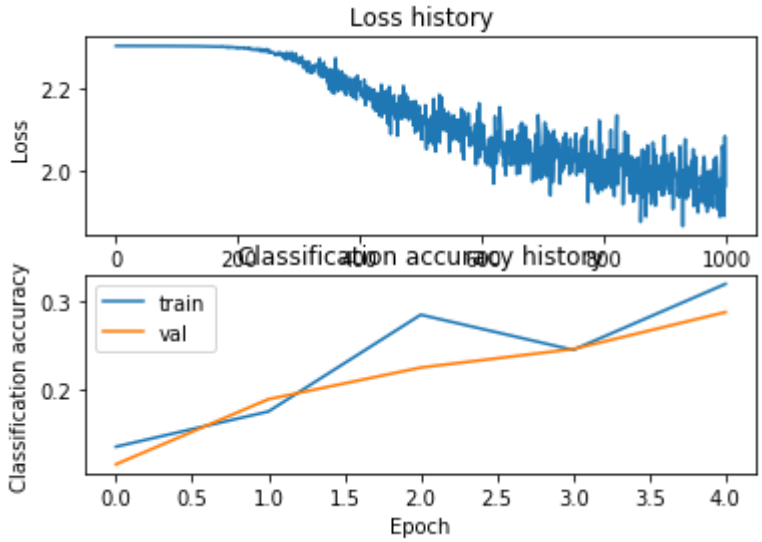
With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

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.

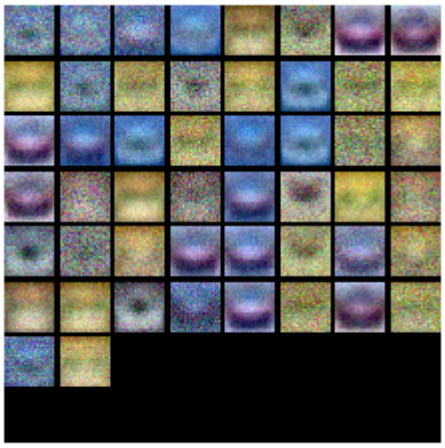
In [ ]:

```
# 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()
```



In [ ]:

```
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.

$YourAnswer:$ list a combination of hyperparameter, use them to train the network and test on the validation set. Choose the hyperparameters with the best validation accuracy

In [ ]:

```
best_net = None # store the best model into this
best_accuracy = 0.0

################################################################################
#
# 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)*****

input_size = 32 * 32 * 3
num_classes = 10

hidden_sizes = [75, 100,300, 500]
regs = [0.001, 0.01, 0.5, 1, 3]
lrs = [1e-4, 4e-4, 9e-4, 2e-3]


for hidden_size in hidden_sizes:
    for reg in regs:
        for lr in lrs:
            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=300,
                        learning_rate=lr, learning_rate_decay=0.95,
                        reg=reg, verbose=False)

            # Predict on the validation set
            train_acc = (net.predict(X_train) == y_train).mean()
            val_acc = (net.predict(X_val) == y_val).mean()

            print("hidden_size:", hidden_size, "reg:", reg, "lr:", lr, "val_ac
c:", val_acc, "train_acc:", train_acc)
            if (val_acc > best_accuracy):
                best_net = net
                best_accuracy = val_acc

print("best accuracy:", best_accuracy)
```

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

```
hidden_size: 75 reg: 0.001 lr: 0.0001 val_acc: 0.306 train_acc: 0.30
159183673469386
hidden_size: 75 reg: 0.001 lr: 0.0004 val_acc: 0.455 train_acc: 0.44
918367346938776
hidden_size: 75 reg: 0.001 lr: 0.0009 val_acc: 0.477 train_acc: 0.50
01836734693877
hidden_size: 75 reg: 0.001 lr: 0.002 val_acc: 0.487 train_acc: 0.505
0816326530613
hidden_size: 75 reg: 0.01 lr: 0.0001 val_acc: 0.303 train_acc: 0.307
6938775510204
hidden_size: 75 reg: 0.01 lr: 0.0004 val_acc: 0.454 train_acc: 0.444
18367346938775
hidden_size: 75 reg: 0.01 lr: 0.0009 val_acc: 0.495 train_acc: 0.505
5714285714286
hidden_size: 75 reg: 0.01 lr: 0.002 val_acc: 0.457 train_acc: 0.5046
530612244898
hidden_size: 75 reg: 0.5 lr: 0.0001 val_acc: 0.306 train_acc: 0.3042
0408163265306
hidden_size: 75 reg: 0.5 lr: 0.0004 val_acc: 0.446 train_acc: 0.4388
1632653061226
hidden_size: 75 reg: 0.5 lr: 0.0009 val_acc: 0.462 train_acc: 0.4911
632653061225
hidden_size: 75 reg: 0.5 lr: 0.002 val_acc: 0.453 train_acc: 0.47155
102040816327
hidden_size: 75 reg: 1 lr: 0.0001 val_acc: 0.296 train_acc: 0.299693
8775510204
hidden_size: 75 reg: 1 lr: 0.0004 val_acc: 0.426 train_acc: 0.432081
6326530612
hidden_size: 75 reg: 1 lr: 0.0009 val_acc: 0.483 train_acc: 0.479795
9183673469
hidden_size: 75 reg: 1 lr: 0.002 val_acc: 0.469 train_acc: 0.4798571
4285714287
hidden_size: 75 reg: 3 lr: 0.0001 val_acc: 0.279 train_acc: 0.281857
14285714286
hidden_size: 75 reg: 3 lr: 0.0004 val_acc: 0.419 train_acc: 0.398979
5918367347
hidden_size: 75 reg: 3 lr: 0.0009 val_acc: 0.442 train_acc: 0.425122
44897959184
hidden_size: 75 reg: 3 lr: 0.002 val_acc: 0.438 train_acc: 0.4170408
163265306
hidden_size: 100 reg: 0.001 lr: 0.0001 val_acc: 0.306 train_acc: 0.3
0873469387755104
hidden_size: 100 reg: 0.001 lr: 0.0004 val_acc: 0.452 train_acc: 0.4
5010204081632654
hidden_size: 100 reg: 0.001 lr: 0.0009 val_acc: 0.47 train_acc: 0.51
75918367346939
hidden_size: 100 reg: 0.001 lr: 0.002 val_acc: 0.488 train_acc: 0.52
16938775510204
hidden_size: 100 reg: 0.01 lr: 0.0001 val_acc: 0.303 train_acc: 0.30
60816326530612
hidden_size: 100 reg: 0.01 lr: 0.0004 val_acc: 0.453 train_acc: 0.45
026530612244897
hidden_size: 100 reg: 0.01 lr: 0.0009 val_acc: 0.471 train_acc: 0.50
43877551020408
hidden_size: 100 reg: 0.01 lr: 0.002 val_acc: 0.48 train_acc: 0.5349
183673469388
hidden_size: 100 reg: 0.5 lr: 0.0001 val_acc: 0.305 train_acc: 0.307
83673469387757
hidden_size: 100 reg: 0.5 lr: 0.0004 val_acc: 0.451 train_acc: 0.444
42857142857145
hidden_size: 100 reg: 0.5 lr: 0.0009 val_acc: 0.48 train_acc: 0.4942
```

```
244897959184
hidden_size: 100 reg: 0.5 lr: 0.002 val_acc: 0.488 train_acc: 0.5037
755102040816
hidden_size: 100 reg: 1 lr: 0.0001 val_acc: 0.309 train_acc: 0.30444
897959183675
hidden_size: 100 reg: 1 lr: 0.0004 val_acc: 0.452 train_acc: 0.43716
326530612243
hidden_size: 100 reg: 1 lr: 0.0009 val_acc: 0.479 train_acc: 0.47722
448979591836
hidden_size: 100 reg: 1 lr: 0.002 val_acc: 0.47 train_acc: 0.4824693
8775510206
hidden_size: 100 reg: 3 lr: 0.0001 val_acc: 0.283 train_acc: 0.28940
81632653061
hidden_size: 100 reg: 3 lr: 0.0004 val_acc: 0.41 train_acc: 0.400612
24489795916
hidden_size: 100 reg: 3 lr: 0.0009 val_acc: 0.429 train_acc: 0.42806
122448979594
hidden_size: 100 reg: 3 lr: 0.002 val_acc: 0.427 train_acc: 0.414367
3469387755
hidden_size: 300 reg: 0.001 lr: 0.0001 val_acc: 0.316 train_acc: 0.3
2555102040816325
hidden_size: 300 reg: 0.001 lr: 0.0004 val_acc: 0.457 train_acc: 0.4
586938775510204
hidden_size: 300 reg: 0.001 lr: 0.0009 val_acc: 0.495 train_acc: 0.5
169795918367347
hidden_size: 300 reg: 0.001 lr: 0.002 val_acc: 0.422 train_acc: 0.45
63265306122449
hidden_size: 300 reg: 0.01 lr: 0.0001 val_acc: 0.313 train_acc: 0.32
49183673469388
hidden_size: 300 reg: 0.01 lr: 0.0004 val_acc: 0.462 train_acc: 0.45
74285714285714
hidden_size: 300 reg: 0.01 lr: 0.0009 val_acc: 0.487 train_acc: 0.52
71632653061225
hidden_size: 300 reg: 0.01 lr: 0.002 val_acc: 0.499 train_acc: 0.534
6122448979592
hidden_size: 300 reg: 0.5 lr: 0.0001 val_acc: 0.315 train_acc: 0.321
3061224489796
hidden_size: 300 reg: 0.5 lr: 0.0004 val_acc: 0.466 train_acc: 0.451
3469387755102
hidden_size: 300 reg: 0.5 lr: 0.0009 val_acc: 0.489 train_acc: 0.505
7755102040816
hidden_size: 300 reg: 0.5 lr: 0.002 val_acc: 0.484 train_acc: 0.5071
428571428571
hidden_size: 300 reg: 1 lr: 0.0001 val_acc: 0.315 train_acc: 0.31763
26530612245
hidden_size: 300 reg: 1 lr: 0.0004 val_acc: 0.449 train_acc: 0.43791
83673469388
hidden_size: 300 reg: 1 lr: 0.0009 val_acc: 0.469 train_acc: 0.47557
14285714286
hidden_size: 300 reg: 1 lr: 0.002 val_acc: 0.458 train_acc: 0.474836
73469387755
hidden_size: 300 reg: 3 lr: 0.0001 val_acc: 0.312 train_acc: 0.30646
9387755102
hidden_size: 300 reg: 3 lr: 0.0004 val_acc: 0.416 train_acc: 0.41108
163265306125
hidden_size: 300 reg: 3 lr: 0.0009 val_acc: 0.444 train_acc: 0.43075
510204081635
hidden_size: 300 reg: 3 lr: 0.002 val_acc: 0.425 train_acc: 0.423387
7551020408
hidden_size: 500 reg: 0.001 lr: 0.0001 val_acc: 0.323 train_acc: 0.3
296938775510204
```

```
hidden_size: 500 reg: 0.001 lr: 0.0004 val_acc: 0.465 train_acc: 0.4
6285714285714286
hidden_size: 500 reg: 0.001 lr: 0.0009 val_acc: 0.5 train_acc: 0.529
6326530612245
hidden_size: 500 reg: 0.001 lr: 0.002 val_acc: 0.501 train_acc: 0.55
92244897959183
hidden_size: 500 reg: 0.01 lr: 0.0001 val_acc: 0.319 train_acc: 0.32
95918367346939
hidden_size: 500 reg: 0.01 lr: 0.0004 val_acc: 0.467 train_acc: 0.46
344897959183673
hidden_size: 500 reg: 0.01 lr: 0.0009 val_acc: 0.492 train_acc: 0.53
2734693877551
hidden_size: 500 reg: 0.01 lr: 0.002 val_acc: 0.487 train_acc: 0.544
2040816326531
hidden_size: 500 reg: 0.5 lr: 0.0001 val_acc: 0.323 train_acc: 0.328
7551020408163
hidden_size: 500 reg: 0.5 lr: 0.0004 val_acc: 0.453 train_acc: 0.450
5918367346939
hidden_size: 500 reg: 0.5 lr: 0.0009 val_acc: 0.479 train_acc: 0.513
0408163265306
hidden_size: 500 reg: 0.5 lr: 0.002 val_acc: 0.498 train_acc: 0.5222
653061224489
hidden_size: 500 reg: 1 lr: 0.0001 val_acc: 0.319 train_acc: 0.32536
73469387755
hidden_size: 500 reg: 1 lr: 0.0004 val_acc: 0.452 train_acc: 0.44428
57142857143
hidden_size: 500 reg: 1 lr: 0.0009 val_acc: 0.461 train_acc: 0.48836
734693877554
hidden_size: 500 reg: 1 lr: 0.002 val_acc: 0.445 train_acc: 0.457
hidden_size: 500 reg: 3 lr: 0.0001 val_acc: 0.317 train_acc: 0.31381
632653061226
hidden_size: 500 reg: 3 lr: 0.0004 val_acc: 0.426 train_acc: 0.41197
95918367347
hidden_size: 500 reg: 3 lr: 0.0009 val_acc: 0.442 train_acc: 0.43861
22448979592
hidden_size: 500 reg: 3 lr: 0.002 val_acc: 0.442 train_acc: 0.425122
44897959184
best accuracy: 0.501
```

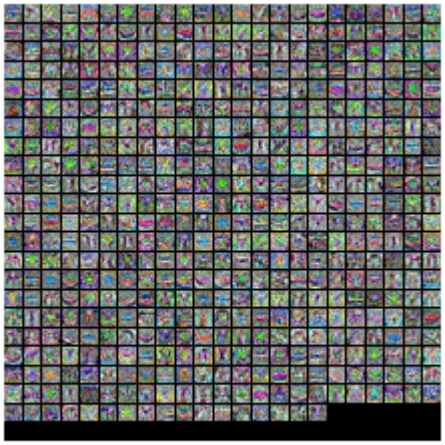
In [ ]:

```
# 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:  0.501
```

In [ ]:

```
# 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%.

In [ ]:

```
# 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:  0.497
```

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 $Your Explanation:$

The testing accuracy is lower than the training accuracy indicates that the model is overfitting in the dataset.

1. Training on a larger dataset can increase the diversity of the data, which may reduce the overfitting.
2. Too complex model may increase the overfitting.
3. Increasing the regularization strength avoid some weight with large magnitude, which avoid overfitting.

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

In [ ]:

```
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-ipytho
n
%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.

In [ ]:

```
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 ca
use 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.

In [ ]:

```
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.

In [ ]:

```
# 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 reg in regularization_strengths:
    for lr in learning_rates:
        svm = LinearSVM()
        # Train the svm
        loss_hist = svm.train(X_train_feats, y_train, lr, reg, num_iters=5000)

        # Predict on the validation set
        train_acc = (svm.predict(X_train_feats) == y_train).mean()
        val_acc = (svm.predict(X_val_feats) == y_val).mean()

        if (val_acc > best_val):
            best_svm = svm
            best_val = val_acc

# *****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: 0.419000
```

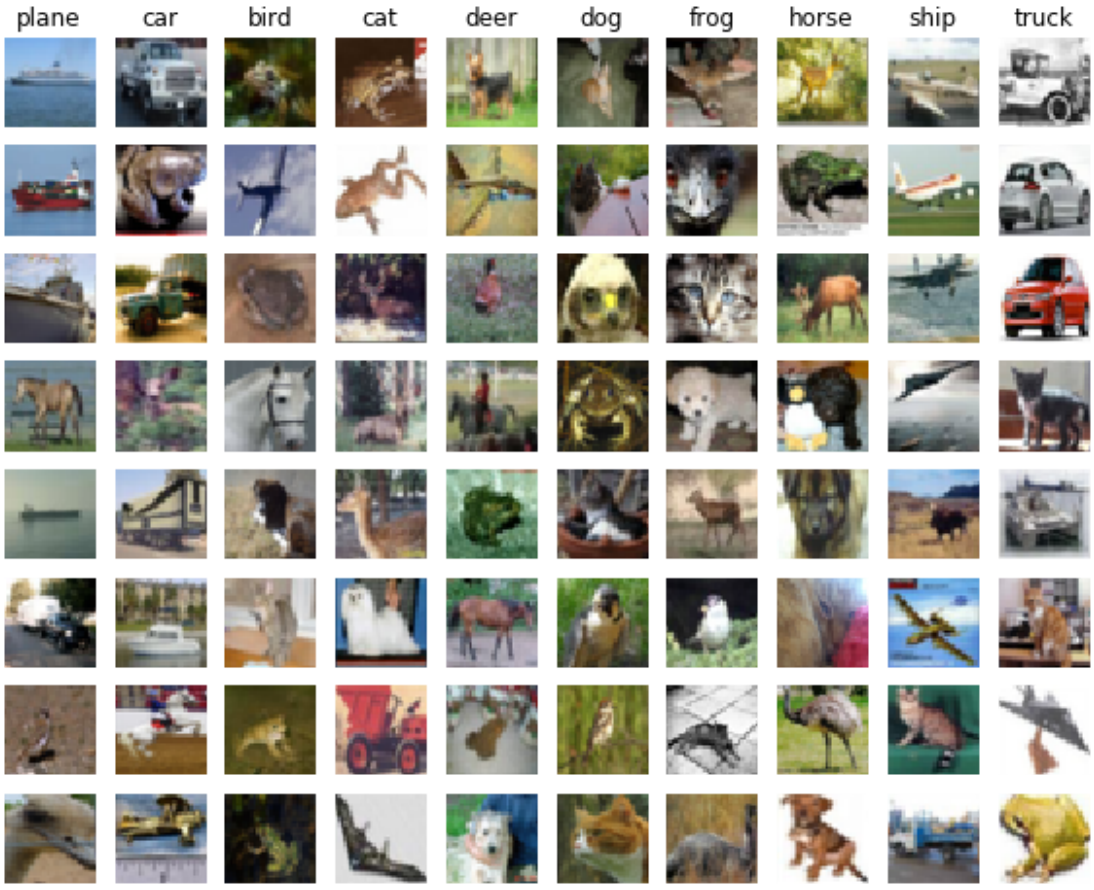
In [ ]:

```
# 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)
```

```
0.419
```

In [ ]:

```
# 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?

$Your Answer:$

Take first column for example, the pictures are mostly realtively bluer, which suits the environment of the plane. There is a horse which is misclassfied into the plane class, and the background of the horse image is blue, which may be the reasom. Since the weights of svm is the 'average' of the training dataset, svm would always misclassified the picture with similar shape or environment.

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.

In [ ]:

```
# 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)
```

In [ ]:

```
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
best_accuracy = 0.0

################################################################################
# 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)*****

regs = [0.001, 0.01, 0.5]
lrs = [0.09, 0.1, 0.2, 0.3, 0.5]
learning_rate_decays = [0.99, 0.95, 0.75, 0.5]
batch_sizes = [100, 200, 400, 600]

for reg in regs:
    for lr in lrs:
        for lr_decay in learning_rate_decays:
            for batch_size in batch_sizes:
                net = TwoLayerNet(input_dim, hidden_dim, num_classes)

                # Train the network
                stats = net.train(X_train_feats, y_train, X_val_feats, y_val,
                            num_iters=1000, batch_size=batch_size,
                            learning_rate=lr, learning_rate_decay=lr_decay,
                            reg=reg, verbose=False)

                # Predict on the validation set
                train_acc = (net.predict(X_train_feats) == y_train).mean()
                val_acc = (net.predict(X_val_feats) == y_val).mean()

                print("batch_size:", batch_size, "reg:", reg, "lr:", lr, "lr_dec
ay:", lr_decay, "val_acc:", val_acc, "train_acc:", train_acc)
                if (val_acc > best_accuracy):
                    best_net = net
                    best_accuracy = val_acc

print("best accuracy:", best_accuracy)

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

```
batch_size: 100 reg: 0.001 lr: 0.09 lr_decay: 0.99 val_acc: 0.525 tr
ain_acc: 0.5195102040816326
batch_size: 200 reg: 0.001 lr: 0.09 lr_decay: 0.99 val_acc: 0.519 tr
ain_acc: 0.5238367346938776
batch_size: 400 reg: 0.001 lr: 0.09 lr_decay: 0.99 val_acc: 0.512 tr
ain_acc: 0.5245102040816326
batch_size: 600 reg: 0.001 lr: 0.09 lr_decay: 0.99 val_acc: 0.512 tr
ain_acc: 0.5238979591836734
batch_size: 100 reg: 0.001 lr: 0.09 lr_decay: 0.95 val_acc: 0.507 tr
ain_acc: 0.5156938775510204
batch_size: 200 reg: 0.001 lr: 0.09 lr_decay: 0.95 val_acc: 0.525 tr
ain_acc: 0.5164081632653061
batch_size: 400 reg: 0.001 lr: 0.09 lr_decay: 0.95 val_acc: 0.509 tr
ain_acc: 0.5167551020408163
batch_size: 600 reg: 0.001 lr: 0.09 lr_decay: 0.95 val_acc: 0.512 tr
ain_acc: 0.5221020408163265
batch_size: 100 reg: 0.001 lr: 0.09 lr_decay: 0.75 val_acc: 0.491 tr
ain_acc: 0.48828571428571427
batch_size: 200 reg: 0.001 lr: 0.09 lr_decay: 0.75 val_acc: 0.447 tr
ain_acc: 0.4588979591836735
batch_size: 400 reg: 0.001 lr: 0.09 lr_decay: 0.75 val_acc: 0.448 tr
ain_acc: 0.4599591836734694
batch_size: 600 reg: 0.001 lr: 0.09 lr_decay: 0.75 val_acc: 0.492 tr
ain_acc: 0.502265306122449
batch_size: 100 reg: 0.001 lr: 0.09 lr_decay: 0.5 val_acc: 0.392 tra
in_acc: 0.39155102040816325
batch_size: 200 reg: 0.001 lr: 0.09 lr_decay: 0.5 val_acc: 0.259 tra
in_acc: 0.2433061224489796
batch_size: 400 reg: 0.001 lr: 0.09 lr_decay: 0.5 val_acc: 0.267 tra
in_acc: 0.25479591836734694
batch_size: 600 reg: 0.001 lr: 0.09 lr_decay: 0.5 val_acc: 0.446 tra
in_acc: 0.45991836734693875
batch_size: 100 reg: 0.001 lr: 0.1 lr_decay: 0.99 val_acc: 0.504 tra
in_acc: 0.519734693877551
batch_size: 200 reg: 0.001 lr: 0.1 lr_decay: 0.99 val_acc: 0.517 tra
in_acc: 0.525734693877551
batch_size: 400 reg: 0.001 lr: 0.1 lr_decay: 0.99 val_acc: 0.507 tra
in_acc: 0.5293877551020408
batch_size: 600 reg: 0.001 lr: 0.1 lr_decay: 0.99 val_acc: 0.521 tra
in_acc: 0.5299387755102041
batch_size: 100 reg: 0.001 lr: 0.1 lr_decay: 0.95 val_acc: 0.504 tra
in_acc: 0.5166326530612245
batch_size: 200 reg: 0.001 lr: 0.1 lr_decay: 0.95 val_acc: 0.515 tra
in_acc: 0.5240408163265307
batch_size: 400 reg: 0.001 lr: 0.1 lr_decay: 0.95 val_acc: 0.52 trai
n_acc: 0.5240816326530612
batch_size: 600 reg: 0.001 lr: 0.1 lr_decay: 0.95 val_acc: 0.519 tra
in_acc: 0.5289387755102041
batch_size: 100 reg: 0.001 lr: 0.1 lr_decay: 0.75 val_acc: 0.497 tra
in_acc: 0.49883673469387757
batch_size: 200 reg: 0.001 lr: 0.1 lr_decay: 0.75 val_acc: 0.477 tra
in_acc: 0.4741020408163265
batch_size: 400 reg: 0.001 lr: 0.1 lr_decay: 0.75 val_acc: 0.467 tra
in_acc: 0.4744897959183674
batch_size: 600 reg: 0.001 lr: 0.1 lr_decay: 0.75 val_acc: 0.511 tra
in_acc: 0.5134285714285715
batch_size: 100 reg: 0.001 lr: 0.1 lr_decay: 0.5 val_acc: 0.404 trai
n_acc: 0.41475510204081634
batch_size: 200 reg: 0.001 lr: 0.1 lr_decay: 0.5 val_acc: 0.276 trai
n_acc: 0.2664489795918367
batch_size: 400 reg: 0.001 lr: 0.1 lr_decay: 0.5 val_acc: 0.287 trai
```

```
n_acc: 0.2693877551020408
batch_size: 600 reg: 0.001 lr: 0.1 lr_decay: 0.5 val_acc: 0.466 trai
n_acc: 0.4717959183673469
batch_size: 100 reg: 0.001 lr: 0.2 lr_decay: 0.99 val_acc: 0.529 tra
in_acc: 0.5546326530612244
batch_size: 200 reg: 0.001 lr: 0.2 lr_decay: 0.99 val_acc: 0.53 trai
n_acc: 0.5605918367346939
batch_size: 400 reg: 0.001 lr: 0.2 lr_decay: 0.99 val_acc: 0.544 tra
in_acc: 0.5627551020408164
batch_size: 600 reg: 0.001 lr: 0.2 lr_decay: 0.99 val_acc: 0.555 tra
in_acc: 0.5688367346938775
batch_size: 100 reg: 0.001 lr: 0.2 lr_decay: 0.95 val_acc: 0.532 tra
in_acc: 0.5545510204081633
batch_size: 200 reg: 0.001 lr: 0.2 lr_decay: 0.95 val_acc: 0.52 trai
n_acc: 0.5549795918367347
batch_size: 400 reg: 0.001 lr: 0.2 lr_decay: 0.95 val_acc: 0.537 tra
in_acc: 0.5592040816326531
batch_size: 600 reg: 0.001 lr: 0.2 lr_decay: 0.95 val_acc: 0.553 tra
in_acc: 0.5668163265306122
batch_size: 100 reg: 0.001 lr: 0.2 lr_decay: 0.75 val_acc: 0.516 tra
in_acc: 0.5316938775510204
batch_size: 200 reg: 0.001 lr: 0.2 lr_decay: 0.75 val_acc: 0.519 tra
in_acc: 0.5289387755102041
batch_size: 400 reg: 0.001 lr: 0.2 lr_decay: 0.75 val_acc: 0.512 tra
in_acc: 0.5314285714285715
batch_size: 600 reg: 0.001 lr: 0.2 lr_decay: 0.75 val_acc: 0.536 tra
in_acc: 0.549
batch_size: 100 reg: 0.001 lr: 0.2 lr_decay: 0.5 val_acc: 0.499 trai
n_acc: 0.5114897959183673
batch_size: 200 reg: 0.001 lr: 0.2 lr_decay: 0.5 val_acc: 0.451 trai
n_acc: 0.4625714285714286
batch_size: 400 reg: 0.001 lr: 0.2 lr_decay: 0.5 val_acc: 0.451 trai
n_acc: 0.4632857142857143
batch_size: 600 reg: 0.001 lr: 0.2 lr_decay: 0.5 val_acc: 0.508 trai
n_acc: 0.530795918367347
batch_size: 100 reg: 0.001 lr: 0.3 lr_decay: 0.99 val_acc: 0.533 tra
in_acc: 0.5755714285714286
batch_size: 200 reg: 0.001 lr: 0.3 lr_decay: 0.99 val_acc: 0.55 trai
n_acc: 0.5879795918367346
batch_size: 400 reg: 0.001 lr: 0.3 lr_decay: 0.99 val_acc: 0.59 trai
n_acc: 0.5990816326530612
batch_size: 600 reg: 0.001 lr: 0.3 lr_decay: 0.99 val_acc: 0.576 tra
in_acc: 0.6054693877551021
batch_size: 100 reg: 0.001 lr: 0.3 lr_decay: 0.95 val_acc: 0.544 tra
in_acc: 0.5675102040816327
batch_size: 200 reg: 0.001 lr: 0.3 lr_decay: 0.95 val_acc: 0.544 tra
in_acc: 0.5807551020408164
batch_size: 400 reg: 0.001 lr: 0.3 lr_decay: 0.95 val_acc: 0.558 tra
in_acc: 0.5902448979591837
batch_size: 600 reg: 0.001 lr: 0.3 lr_decay: 0.95 val_acc: 0.576 tra
in_acc: 0.599
batch_size: 100 reg: 0.001 lr: 0.3 lr_decay: 0.75 val_acc: 0.519 tra
in_acc: 0.5542448979591836
batch_size: 200 reg: 0.001 lr: 0.3 lr_decay: 0.75 val_acc: 0.532 tra
in_acc: 0.5475714285714286
batch_size: 400 reg: 0.001 lr: 0.3 lr_decay: 0.75 val_acc: 0.53 trai
n_acc: 0.5513877551020409
batch_size: 600 reg: 0.001 lr: 0.3 lr_decay: 0.75 val_acc: 0.554 tra
in_acc: 0.5770612244897959
batch_size: 100 reg: 0.001 lr: 0.3 lr_decay: 0.5 val_acc: 0.525 trai
n_acc: 0.5272653061224489
```

```
batch_size: 200 reg: 0.001 lr: 0.3 lr_decay: 0.5 val_acc: 0.502 trai
n_acc: 0.5075510204081632
batch_size: 400 reg: 0.001 lr: 0.3 lr_decay: 0.5 val_acc: 0.492 trai
n_acc: 0.5054081632653061
batch_size: 600 reg: 0.001 lr: 0.3 lr_decay: 0.5 val_acc: 0.521 trai
n_acc: 0.5471020408163265
batch_size: 100 reg: 0.001 lr: 0.5 lr_decay: 0.99 val_acc: 0.555 tra
in_acc: 0.5933877551020408
batch_size: 200 reg: 0.001 lr: 0.5 lr_decay: 0.99 val_acc: 0.569 tra
in_acc: 0.6205102040816326
batch_size: 400 reg: 0.001 lr: 0.5 lr_decay: 0.99 val_acc: 0.592 tra
in_acc: 0.6394489795918368
batch_size: 600 reg: 0.001 lr: 0.5 lr_decay: 0.99 val_acc: 0.594 tra
in_acc: 0.6524897959183673
batch_size: 100 reg: 0.001 lr: 0.5 lr_decay: 0.95 val_acc: 0.553 tra
in_acc: 0.5903469387755103
batch_size: 200 reg: 0.001 lr: 0.5 lr_decay: 0.95 val_acc: 0.562 tra
in_acc: 0.6181224489795918
batch_size: 400 reg: 0.001 lr: 0.5 lr_decay: 0.95 val_acc: 0.568 tra
in_acc: 0.632265306122449
batch_size: 600 reg: 0.001 lr: 0.5 lr_decay: 0.95 val_acc: 0.591 tra
in_acc: 0.647061224489796
batch_size: 100 reg: 0.001 lr: 0.5 lr_decay: 0.75 val_acc: 0.561 tra
in_acc: 0.5851632653061225
batch_size: 200 reg: 0.001 lr: 0.5 lr_decay: 0.75 val_acc: 0.571 tra
in_acc: 0.5844897959183674
batch_size: 400 reg: 0.001 lr: 0.5 lr_decay: 0.75 val_acc: 0.567 tra
in_acc: 0.5901020408163266
batch_size: 600 reg: 0.001 lr: 0.5 lr_decay: 0.75 val_acc: 0.573 tra
in_acc: 0.6231836734693877
batch_size: 100 reg: 0.001 lr: 0.5 lr_decay: 0.5 val_acc: 0.524 trai
n_acc: 0.5554489795918367
batch_size: 200 reg: 0.001 lr: 0.5 lr_decay: 0.5 val_acc: 0.522 trai
n_acc: 0.5371020408163265
batch_size: 400 reg: 0.001 lr: 0.5 lr_decay: 0.5 val_acc: 0.513 trai
n_acc: 0.5368979591836734
batch_size: 600 reg: 0.001 lr: 0.5 lr_decay: 0.5 val_acc: 0.56 train
_acc: 0.5891836734693877
batch_size: 100 reg: 0.01 lr: 0.09 lr_decay: 0.99 val_acc: 0.501 tra
in_acc: 0.5057551020408163
batch_size: 200 reg: 0.01 lr: 0.09 lr_decay: 0.99 val_acc: 0.49 trai
n_acc: 0.5018571428571429
batch_size: 400 reg: 0.01 lr: 0.09 lr_decay: 0.99 val_acc: 0.496 tra
in_acc: 0.5029591836734694
batch_size: 600 reg: 0.01 lr: 0.09 lr_decay: 0.99 val_acc: 0.502 tra
in_acc: 0.505061224489796
batch_size: 100 reg: 0.01 lr: 0.09 lr_decay: 0.95 val_acc: 0.491 tra
in_acc: 0.4903469387755102
batch_size: 200 reg: 0.01 lr: 0.09 lr_decay: 0.95 val_acc: 0.482 tra
in_acc: 0.49489795918367346
batch_size: 400 reg: 0.01 lr: 0.09 lr_decay: 0.95 val_acc: 0.481 tra
in_acc: 0.49742857142857144
batch_size: 600 reg: 0.01 lr: 0.09 lr_decay: 0.95 val_acc: 0.492 tra
in_acc: 0.504265306122449
batch_size: 100 reg: 0.01 lr: 0.09 lr_decay: 0.75 val_acc: 0.462 tra
in_acc: 0.4767959183673469
batch_size: 200 reg: 0.01 lr: 0.09 lr_decay: 0.75 val_acc: 0.426 tra
in_acc: 0.438469387755102
batch_size: 400 reg: 0.01 lr: 0.09 lr_decay: 0.75 val_acc: 0.423 tra
in_acc: 0.4403877551020408
batch_size: 600 reg: 0.01 lr: 0.09 lr_decay: 0.75 val_acc: 0.482 tra
```

```
in_acc: 0.48412244897959184
batch_size: 100 reg: 0.01 lr: 0.09 lr_decay: 0.5 val_acc: 0.367 trai
n_acc: 0.36253061224489797
batch_size: 200 reg: 0.01 lr: 0.09 lr_decay: 0.5 val_acc: 0.246 trai
n_acc: 0.23087755102040816
batch_size: 400 reg: 0.01 lr: 0.09 lr_decay: 0.5 val_acc: 0.243 trai
n_acc: 0.2296326530612245
batch_size: 600 reg: 0.01 lr: 0.09 lr_decay: 0.5 val_acc: 0.421 trai
n_acc: 0.4376938775510204
batch_size: 100 reg: 0.01 lr: 0.1 lr_decay: 0.99 val_acc: 0.517 trai
n_acc: 0.505734693877551
batch_size: 200 reg: 0.01 lr: 0.1 lr_decay: 0.99 val_acc: 0.504 trai
n_acc: 0.5118979591836734
batch_size: 400 reg: 0.01 lr: 0.1 lr_decay: 0.99 val_acc: 0.497 trai
n_acc: 0.5108163265306123
batch_size: 600 reg: 0.01 lr: 0.1 lr_decay: 0.99 val_acc: 0.5 train_
acc: 0.5133877551020408
batch_size: 100 reg: 0.01 lr: 0.1 lr_decay: 0.95 val_acc: 0.507 trai
n_acc: 0.502061224489796
batch_size: 200 reg: 0.01 lr: 0.1 lr_decay: 0.95 val_acc: 0.494 trai
n_acc: 0.5013265306122449
batch_size: 400 reg: 0.01 lr: 0.1 lr_decay: 0.95 val_acc: 0.502 trai
n_acc: 0.5061224489795918
batch_size: 600 reg: 0.01 lr: 0.1 lr_decay: 0.95 val_acc: 0.51 train
_acc: 0.5123877551020408
batch_size: 100 reg: 0.01 lr: 0.1 lr_decay: 0.75 val_acc: 0.473 trai
n_acc: 0.4850408163265306
batch_size: 200 reg: 0.01 lr: 0.1 lr_decay: 0.75 val_acc: 0.439 trai
n_acc: 0.45671428571428574
batch_size: 400 reg: 0.01 lr: 0.1 lr_decay: 0.75 val_acc: 0.442 trai
n_acc: 0.4571224489795918
batch_size: 600 reg: 0.01 lr: 0.1 lr_decay: 0.75 val_acc: 0.484 trai
n_acc: 0.49448979591836734
batch_size: 100 reg: 0.01 lr: 0.1 lr_decay: 0.5 val_acc: 0.388 train
_acc: 0.39691836734693875
batch_size: 200 reg: 0.01 lr: 0.1 lr_decay: 0.5 val_acc: 0.276 train
_acc: 0.2589183673469388
batch_size: 400 reg: 0.01 lr: 0.1 lr_decay: 0.5 val_acc: 0.277 train
_acc: 0.26285714285714284
batch_size: 600 reg: 0.01 lr: 0.1 lr_decay: 0.5 val_acc: 0.445 train
_acc: 0.4545102040816327
batch_size: 100 reg: 0.01 lr: 0.2 lr_decay: 0.99 val_acc: 0.513 trai
n_acc: 0.5083877551020408
batch_size: 200 reg: 0.01 lr: 0.2 lr_decay: 0.99 val_acc: 0.516 trai
n_acc: 0.5149591836734694
batch_size: 400 reg: 0.01 lr: 0.2 lr_decay: 0.99 val_acc: 0.5 train_
acc: 0.5189795918367347
batch_size: 600 reg: 0.01 lr: 0.2 lr_decay: 0.99 val_acc: 0.515 trai
n_acc: 0.5216326530612245
batch_size: 100 reg: 0.01 lr: 0.2 lr_decay: 0.95 val_acc: 0.504 trai
n_acc: 0.5108979591836734
batch_size: 200 reg: 0.01 lr: 0.2 lr_decay: 0.95 val_acc: 0.496 trai
n_acc: 0.5175102040816326
batch_size: 400 reg: 0.01 lr: 0.2 lr_decay: 0.95 val_acc: 0.512 trai
n_acc: 0.5235510204081633
batch_size: 600 reg: 0.01 lr: 0.2 lr_decay: 0.95 val_acc: 0.514 trai
n_acc: 0.5246938775510204
batch_size: 100 reg: 0.01 lr: 0.2 lr_decay: 0.75 val_acc: 0.52 train
_acc: 0.5177755102040816
batch_size: 200 reg: 0.01 lr: 0.2 lr_decay: 0.75 val_acc: 0.502 trai
n_acc: 0.515061224489796
```

```
batch_size: 400 reg: 0.01 lr: 0.2 lr_decay: 0.75 val_acc: 0.51 train
_acc: 0.5159387755102041
batch_size: 600 reg: 0.01 lr: 0.2 lr_decay: 0.75 val_acc: 0.504 trai
n_acc: 0.5228979591836734
batch_size: 100 reg: 0.01 lr: 0.2 lr_decay: 0.5 val_acc: 0.487 train
_acc: 0.4919387755102041
batch_size: 200 reg: 0.01 lr: 0.2 lr_decay: 0.5 val_acc: 0.427 train
_acc: 0.4417142857142857
batch_size: 400 reg: 0.01 lr: 0.2 lr_decay: 0.5 val_acc: 0.429 train
_acc: 0.43942857142857145
batch_size: 600 reg: 0.01 lr: 0.2 lr_decay: 0.5 val_acc: 0.5 train_a
cc: 0.5111632653061224
batch_size: 100 reg: 0.01 lr: 0.3 lr_decay: 0.99 val_acc: 0.499 trai
n_acc: 0.5078775510204082
batch_size: 200 reg: 0.01 lr: 0.3 lr_decay: 0.99 val_acc: 0.51 train
_acc: 0.5130408163265306
batch_size: 400 reg: 0.01 lr: 0.3 lr_decay: 0.99 val_acc: 0.507 trai
n_acc: 0.5219183673469387
batch_size: 600 reg: 0.01 lr: 0.3 lr_decay: 0.99 val_acc: 0.521 trai
n_acc: 0.5266530612244898
batch_size: 100 reg: 0.01 lr: 0.3 lr_decay: 0.95 val_acc: 0.503 trai
n_acc: 0.5020816326530613
batch_size: 200 reg: 0.01 lr: 0.3 lr_decay: 0.95 val_acc: 0.514 trai
n_acc: 0.5186122448979592
batch_size: 400 reg: 0.01 lr: 0.3 lr_decay: 0.95 val_acc: 0.508 trai
n_acc: 0.5218979591836734
batch_size: 600 reg: 0.01 lr: 0.3 lr_decay: 0.95 val_acc: 0.521 trai
n_acc: 0.5245510204081633
batch_size: 100 reg: 0.01 lr: 0.3 lr_decay: 0.75 val_acc: 0.501 trai
n_acc: 0.5119387755102041
batch_size: 200 reg: 0.01 lr: 0.3 lr_decay: 0.75 val_acc: 0.506 trai
n_acc: 0.5219795918367347
batch_size: 400 reg: 0.01 lr: 0.3 lr_decay: 0.75 val_acc: 0.508 trai
n_acc: 0.5234285714285715
batch_size: 600 reg: 0.01 lr: 0.3 lr_decay: 0.75 val_acc: 0.512 trai
n_acc: 0.521673469387755
batch_size: 100 reg: 0.01 lr: 0.3 lr_decay: 0.5 val_acc: 0.498 train
_acc: 0.5145306122448979
batch_size: 200 reg: 0.01 lr: 0.3 lr_decay: 0.5 val_acc: 0.483 train
_acc: 0.4913265306122449
batch_size: 400 reg: 0.01 lr: 0.3 lr_decay: 0.5 val_acc: 0.494 train
_acc: 0.4887755102040816
batch_size: 600 reg: 0.01 lr: 0.3 lr_decay: 0.5 val_acc: 0.506 train
_acc: 0.5208775510204081
batch_size: 100 reg: 0.01 lr: 0.5 lr_decay: 0.99 val_acc: 0.484 trai
n_acc: 0.4850816326530612
batch_size: 200 reg: 0.01 lr: 0.5 lr_decay: 0.99 val_acc: 0.5 train_
acc: 0.5131224489795918
batch_size: 400 reg: 0.01 lr: 0.5 lr_decay: 0.99 val_acc: 0.518 trai
n_acc: 0.5212448979591837
batch_size: 600 reg: 0.01 lr: 0.5 lr_decay: 0.99 val_acc: 0.509 trai
n_acc: 0.528795918367347
batch_size: 100 reg: 0.01 lr: 0.5 lr_decay: 0.95 val_acc: 0.467 trai
n_acc: 0.4887959183673469
batch_size: 200 reg: 0.01 lr: 0.5 lr_decay: 0.95 val_acc: 0.49 train
_acc: 0.5015510204081632
batch_size: 400 reg: 0.01 lr: 0.5 lr_decay: 0.95 val_acc: 0.504 trai
n_acc: 0.5165102040816326
batch_size: 600 reg: 0.01 lr: 0.5 lr_decay: 0.95 val_acc: 0.518 trai
n_acc: 0.5275510204081633
batch_size: 100 reg: 0.01 lr: 0.5 lr_decay: 0.75 val_acc: 0.495 trai
```

```
n_acc: 0.5086122448979592
batch_size: 200 reg: 0.01 lr: 0.5 lr_decay: 0.75 val_acc: 0.494 trai
n_acc: 0.5245714285714286
batch_size: 400 reg: 0.01 lr: 0.5 lr_decay: 0.75 val_acc: 0.522 trai
n_acc: 0.5271632653061225
batch_size: 600 reg: 0.01 lr: 0.5 lr_decay: 0.75 val_acc: 0.517 trai
n_acc: 0.5281020408163265
batch_size: 100 reg: 0.01 lr: 0.5 lr_decay: 0.5 val_acc: 0.5 train_a
cc: 0.5186326530612245
batch_size: 200 reg: 0.01 lr: 0.5 lr_decay: 0.5 val_acc: 0.516 train
_acc: 0.5191632653061224
batch_size: 400 reg: 0.01 lr: 0.5 lr_decay: 0.5 val_acc: 0.513 train
_acc: 0.5204081632653061
batch_size: 600 reg: 0.01 lr: 0.5 lr_decay: 0.5 val_acc: 0.537 train
_acc: 0.5238367346938776
batch_size: 100 reg: 0.5 lr: 0.09 lr_decay: 0.99 val_acc: 0.087 trai
n_acc: 0.10026530612244898
batch_size: 200 reg: 0.5 lr: 0.09 lr_decay: 0.99 val_acc: 0.087 trai
n_acc: 0.10026530612244898
batch_size: 400 reg: 0.5 lr: 0.09 lr_decay: 0.99 val_acc: 0.119 trai
n_acc: 0.09961224489795918
batch_size: 600 reg: 0.5 lr: 0.09 lr_decay: 0.99 val_acc: 0.079 trai
n_acc: 0.10042857142857142
batch_size: 100 reg: 0.5 lr: 0.09 lr_decay: 0.95 val_acc: 0.113 trai
n_acc: 0.09973469387755102
batch_size: 200 reg: 0.5 lr: 0.09 lr_decay: 0.95 val_acc: 0.105 trai
n_acc: 0.09989795918367347
batch_size: 400 reg: 0.5 lr: 0.09 lr_decay: 0.95 val_acc: 0.119 trai
n_acc: 0.09961224489795918
batch_size: 600 reg: 0.5 lr: 0.09 lr_decay: 0.95 val_acc: 0.078 trai
n_acc: 0.10044897959183674
batch_size: 100 reg: 0.5 lr: 0.09 lr_decay: 0.75 val_acc: 0.079 trai
n_acc: 0.10042857142857142
batch_size: 200 reg: 0.5 lr: 0.09 lr_decay: 0.75 val_acc: 0.078 trai
n_acc: 0.10044897959183674
batch_size: 400 reg: 0.5 lr: 0.09 lr_decay: 0.75 val_acc: 0.098 trai
n_acc: 0.10004081632653061
batch_size: 600 reg: 0.5 lr: 0.09 lr_decay: 0.75 val_acc: 0.098 trai
n_acc: 0.10004081632653061
batch_size: 100 reg: 0.5 lr: 0.09 lr_decay: 0.5 val_acc: 0.102 train
_acc: 0.09995918367346938
batch_size: 200 reg: 0.5 lr: 0.09 lr_decay: 0.5 val_acc: 0.078 train
_acc: 0.10044897959183674
batch_size: 400 reg: 0.5 lr: 0.09 lr_decay: 0.5 val_acc: 0.078 train
_acc: 0.10044897959183674
batch_size: 600 reg: 0.5 lr: 0.09 lr_decay: 0.5 val_acc: 0.087 train
_acc: 0.10026530612244898
batch_size: 100 reg: 0.5 lr: 0.1 lr_decay: 0.99 val_acc: 0.087 train
_acc: 0.10026530612244898
batch_size: 200 reg: 0.5 lr: 0.1 lr_decay: 0.99 val_acc: 0.102 train
_acc: 0.09995918367346938
batch_size: 400 reg: 0.5 lr: 0.1 lr_decay: 0.99 val_acc: 0.078 train
_acc: 0.10044897959183674
batch_size: 600 reg: 0.5 lr: 0.1 lr_decay: 0.99 val_acc: 0.078 train
_acc: 0.10044897959183674
batch_size: 100 reg: 0.5 lr: 0.1 lr_decay: 0.95 val_acc: 0.113 train
_acc: 0.09973469387755102
batch_size: 200 reg: 0.5 lr: 0.1 lr_decay: 0.95 val_acc: 0.102 train
_acc: 0.09995918367346938
batch_size: 400 reg: 0.5 lr: 0.1 lr_decay: 0.95 val_acc: 0.119 train
_acc: 0.09961224489795918
```

```
batch_size: 600 reg: 0.5 lr: 0.1 lr_decay: 0.95 val_acc: 0.079 train
_acc: 0.10042857142857142
batch_size: 100 reg: 0.5 lr: 0.1 lr_decay: 0.75 val_acc: 0.112 train
_acc: 0.09975510204081632
batch_size: 200 reg: 0.5 lr: 0.1 lr_decay: 0.75 val_acc: 0.078 train
_acc: 0.10044897959183674
batch_size: 400 reg: 0.5 lr: 0.1 lr_decay: 0.75 val_acc: 0.078 train
_acc: 0.10044897959183674
batch_size: 600 reg: 0.5 lr: 0.1 lr_decay: 0.75 val_acc: 0.098 train
_acc: 0.10004081632653061
batch_size: 100 reg: 0.5 lr: 0.1 lr_decay: 0.5 val_acc: 0.078 train_
acc: 0.10044897959183674
batch_size: 200 reg: 0.5 lr: 0.1 lr_decay: 0.5 val_acc: 0.078 train_
acc: 0.10044897959183674
batch_size: 400 reg: 0.5 lr: 0.1 lr_decay: 0.5 val_acc: 0.078 train_
acc: 0.10044897959183674
batch_size: 600 reg: 0.5 lr: 0.1 lr_decay: 0.5 val_acc: 0.087 train_
acc: 0.10026530612244898
batch_size: 100 reg: 0.5 lr: 0.2 lr_decay: 0.99 val_acc: 0.113 train
_acc: 0.09973469387755102
batch_size: 200 reg: 0.5 lr: 0.2 lr_decay: 0.99 val_acc: 0.107 train
_acc: 0.09985714285714285
batch_size: 400 reg: 0.5 lr: 0.2 lr_decay: 0.99 val_acc: 0.105 train
_acc: 0.09989795918367347
batch_size: 600 reg: 0.5 lr: 0.2 lr_decay: 0.99 val_acc: 0.098 train
_acc: 0.10004081632653061
batch_size: 100 reg: 0.5 lr: 0.2 lr_decay: 0.95 val_acc: 0.112 train
_acc: 0.09975510204081632
batch_size: 200 reg: 0.5 lr: 0.2 lr_decay: 0.95 val_acc: 0.087 train
_acc: 0.10026530612244898
batch_size: 400 reg: 0.5 lr: 0.2 lr_decay: 0.95 val_acc: 0.078 train
_acc: 0.10044897959183674
batch_size: 600 reg: 0.5 lr: 0.2 lr_decay: 0.95 val_acc: 0.078 train
_acc: 0.10044897959183674
batch_size: 100 reg: 0.5 lr: 0.2 lr_decay: 0.75 val_acc: 0.119 train
_acc: 0.09961224489795918
batch_size: 200 reg: 0.5 lr: 0.2 lr_decay: 0.75 val_acc: 0.079 train
_acc: 0.10042857142857142
batch_size: 400 reg: 0.5 lr: 0.2 lr_decay: 0.75 val_acc: 0.087 train
_acc: 0.10026530612244898
batch_size: 600 reg: 0.5 lr: 0.2 lr_decay: 0.75 val_acc: 0.119 train
_acc: 0.09961224489795918
batch_size: 100 reg: 0.5 lr: 0.2 lr_decay: 0.5 val_acc: 0.098 train_
acc: 0.10004081632653061
batch_size: 200 reg: 0.5 lr: 0.2 lr_decay: 0.5 val_acc: 0.087 train_
acc: 0.10026530612244898
batch_size: 400 reg: 0.5 lr: 0.2 lr_decay: 0.5 val_acc: 0.107 train_
acc: 0.09985714285714285
batch_size: 600 reg: 0.5 lr: 0.2 lr_decay: 0.5 val_acc: 0.102 train_
acc: 0.09995918367346938
batch_size: 100 reg: 0.5 lr: 0.3 lr_decay: 0.99 val_acc: 0.098 train
_acc: 0.10004081632653061
batch_size: 200 reg: 0.5 lr: 0.3 lr_decay: 0.99 val_acc: 0.119 train
_acc: 0.09961224489795918
batch_size: 400 reg: 0.5 lr: 0.3 lr_decay: 0.99 val_acc: 0.112 train
_acc: 0.09975510204081632
batch_size: 600 reg: 0.5 lr: 0.3 lr_decay: 0.99 val_acc: 0.107 train
_acc: 0.09985714285714285
batch_size: 100 reg: 0.5 lr: 0.3 lr_decay: 0.95 val_acc: 0.079 train
_acc: 0.10042857142857142
batch_size: 200 reg: 0.5 lr: 0.3 lr_decay: 0.95 val_acc: 0.102 train
```

```
_acc: 0.09995918367346938
batch_size: 400 reg: 0.5 lr: 0.3 lr_decay: 0.95 val_acc: 0.079 train
_acc: 0.10042857142857142
batch_size: 600 reg: 0.5 lr: 0.3 lr_decay: 0.95 val_acc: 0.078 train
_acc: 0.10044897959183674
batch_size: 100 reg: 0.5 lr: 0.3 lr_decay: 0.75 val_acc: 0.078 train
_acc: 0.10044897959183674
batch_size: 200 reg: 0.5 lr: 0.3 lr_decay: 0.75 val_acc: 0.098 train
_acc: 0.10004081632653061
batch_size: 400 reg: 0.5 lr: 0.3 lr_decay: 0.75 val_acc: 0.079 train
_acc: 0.10042857142857142
batch_size: 600 reg: 0.5 lr: 0.3 lr_decay: 0.75 val_acc: 0.087 train
_acc: 0.10026530612244898
batch_size: 100 reg: 0.5 lr: 0.3 lr_decay: 0.5 val_acc: 0.078 train_
acc: 0.10044897959183674
batch_size: 200 reg: 0.5 lr: 0.3 lr_decay: 0.5 val_acc: 0.087 train_
acc: 0.10026530612244898
batch_size: 400 reg: 0.5 lr: 0.3 lr_decay: 0.5 val_acc: 0.079 train_
acc: 0.10042857142857142
batch_size: 600 reg: 0.5 lr: 0.3 lr_decay: 0.5 val_acc: 0.079 train_
acc: 0.10042857142857142
batch_size: 100 reg: 0.5 lr: 0.5 lr_decay: 0.99 val_acc: 0.078 train
_acc: 0.10044897959183674
batch_size: 200 reg: 0.5 lr: 0.5 lr_decay: 0.99 val_acc: 0.113 train
_acc: 0.09973469387755102
batch_size: 400 reg: 0.5 lr: 0.5 lr_decay: 0.99 val_acc: 0.105 train
_acc: 0.09989795918367347
batch_size: 600 reg: 0.5 lr: 0.5 lr_decay: 0.99 val_acc: 0.087 train
_acc: 0.10026530612244898
batch_size: 100 reg: 0.5 lr: 0.5 lr_decay: 0.95 val_acc: 0.113 train
_acc: 0.09973469387755102
batch_size: 200 reg: 0.5 lr: 0.5 lr_decay: 0.95 val_acc: 0.105 train
_acc: 0.09989795918367347
batch_size: 400 reg: 0.5 lr: 0.5 lr_decay: 0.95 val_acc: 0.107 train
_acc: 0.09985714285714285
batch_size: 600 reg: 0.5 lr: 0.5 lr_decay: 0.95 val_acc: 0.078 train
_acc: 0.10044897959183674
batch_size: 100 reg: 0.5 lr: 0.5 lr_decay: 0.75 val_acc: 0.078 train
_acc: 0.10044897959183674
batch_size: 200 reg: 0.5 lr: 0.5 lr_decay: 0.75 val_acc: 0.113 train
_acc: 0.09973469387755102
batch_size: 400 reg: 0.5 lr: 0.5 lr_decay: 0.75 val_acc: 0.107 train
_acc: 0.09985714285714285
batch_size: 600 reg: 0.5 lr: 0.5 lr_decay: 0.75 val_acc: 0.119 train
_acc: 0.09961224489795918
batch_size: 100 reg: 0.5 lr: 0.5 lr_decay: 0.5 val_acc: 0.087 train_
acc: 0.10026530612244898
batch_size: 200 reg: 0.5 lr: 0.5 lr_decay: 0.5 val_acc: 0.113 train_
acc: 0.09973469387755102
batch_size: 400 reg: 0.5 lr: 0.5 lr_decay: 0.5 val_acc: 0.078 train_
acc: 0.10044897959183674
batch_size: 600 reg: 0.5 lr: 0.5 lr_decay: 0.5 val_acc: 0.102 train_
acc: 0.09995918367346938
best accuracy: 0.594
```

In [ ]:

```
# 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)
```

```
0.587
```