CS280 Fall 2023 Assignment 1
Part A

Basics & MLP

October 31, 2023

Name: zhao yu

Student ID:2023232115

1. 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:

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

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

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

Show:

the log-likelihood $l(\theta)$ in terms of the Gaussian mixture model:

$$l(\theta) = \sum_{n=1}^{N} \log \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right)$$

take the derivative with respect to $\boldsymbol{\mu}_k$:

$$\frac{d}{d\boldsymbol{\mu}_k} l(\theta) = \sum_{n=1}^{N} \frac{1}{\sum_{k'} \pi_{k'} \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_{k'}, \boldsymbol{\Sigma}_{k'})} \cdot \frac{d}{d\boldsymbol{\mu}_k} \left(\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right)$$

the derivative inside the summation:

$$\frac{d}{d\boldsymbol{\mu}_k} \left(\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right) = \pi_k \frac{d}{d\boldsymbol{\mu}_k} \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

the derivative of the multivariate Gaussian density function with respect to $\boldsymbol{\mu}_k$ is:

$$\frac{d}{d\boldsymbol{\mu}_k} \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = -\boldsymbol{\Sigma}_k^{-1}(\mathbf{x}_n - \boldsymbol{\mu}_k) \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

substitute this back into up expression:

$$\frac{d}{d\boldsymbol{\mu}_k} l(\theta) = \sum_{n=1}^{N} \frac{\pi_k \cdot -\boldsymbol{\Sigma}_k^{-1}(\mathbf{x}_n - \boldsymbol{\mu}_k) \cdot \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{k'} \pi_{k'} \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_{k'}, \boldsymbol{\Sigma}_{k'})}$$

Finally, simplify the expression using the responsibility r_{nk}:

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

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

Given the definition of $l(\theta)$ and r_{nk} from the previous context, we have:

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

To derive $\frac{d}{d\pi_k} l(\theta)$, take the derivative inside the summation:

$$\sum_{n=1}^{N} \frac{d}{d\pi_k} \log p(\mathbf{x}_n|\theta) = \sum_{n=1}^{N} \frac{d}{d\pi_k} \log \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right)$$

Simplifying, we have:

$$\sum_{n=1}^{N} \frac{1}{\sum_{k=1}^{K} \pi_k}$$

So, the derivative of the log-likelihood with respect to π_k is $\max \frac{1}{\pi_i}, i = 1 \to K$.

Bonus:

Considering, with constraint $\sum_k \pi_k = 1$,

We define the Lagrangian function:

$$L(\theta, \lambda) = l(\theta) + \lambda \left(1 - \sum_k \pi_k \right)$$

we take partial derivatives with respect to π_k and λ, and set them equal to zero to find the optimal solution:

Taking the partial derivative with respect to π_k:

$$\frac{\partial L}{\partial \pi_k} = \frac{1}{\pi_k} - \lambda = 0$$

Solving for π_k:

$$\pi_k = \frac{1}{\lambda}$$

Taking the partial derivative with respect to λ:

$$\frac{\partial L}{\partial \lambda} = 1 - \sum_k \pi_k = 1 - K \cdot \frac{1}{\lambda} = 0$$

Solving for λ:

$$\lambda = K$$

Substituting $\lambda = K$ back into $\pi_k = \frac{1}{\lambda}$, we get:

$$\pi_k = \frac{1}{K}$$

This means that under the constraint $\sum_k \pi_k = 1$, the derivative of the log-likelihood $l(\theta)$ with respect to π_k remains $\frac{1}{\pi_k}$, but now π_k follows a uniform distribution, where each π_k is equal to $\frac{1}{K}$.

2. Sotfmax & Computation Graph (10 points).

Recall that the softmax function takes in a vector $(z_1, \ldots, z_D)$ and returns a vector $(y_1, \ldots, y_D)$. We can express it in the following form:

$$r = \sum_j e^{z_j} \qquad y = \frac{e^{z_j}}{r}$$

(a) Consider $D = 2$, i.e. just two inputs and outputs to the softmax. Draw the computation graph relating z_1, z_2, r, y_1, and y_2.

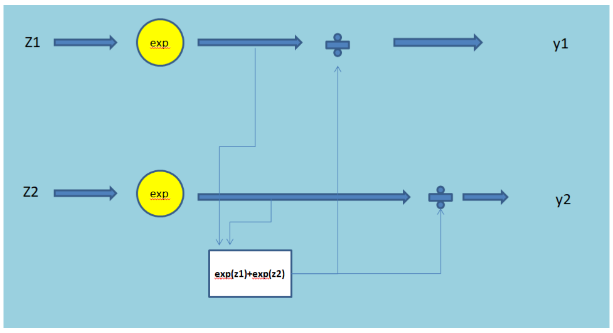


Figure 1: An image of a computation

(b) Determine the backprop updates for computing the $\bar{z}_j$ when given the $\bar{y}_i$. You need to justify your answer. (You may give your answer either for $D = 2$ or for the more general case.)

$$\bar{r} = -\sum_i \bar{y}_i \frac{e^{\bar{z}_i}}{r^2}$$

$$\bar{z}_j = \bar{y}_j \frac{e^{\bar{z}_j}}{r} + \bar{r} e^{\bar{z}_j}$$

(c) Write a function to implement the vector-Jacobian product (VJP) for the softmax function based on your answer from part (b). For efficiency, it should operate on a mini-batch. The inputs are:

- a matrix Z of size $N \times D$ giving a batch of input vectors. N is the batch size and D is the number of dimensions. Each row gives one input vector $z = (z_1, \ldots, z_D)$.
- A matrix $\mathbf{Y_{bar}}$ giving the output error signals. It is also $N \times D$

The output should be the error signal $\mathbf{Z_{bar}}$. Do not use a for loop.

function:

```
import numpy as np

def softmax_vjp(Z, Y_bar):
    R = np.sum(np.exp(Z), axis = 1, keepdims=True)
    R_bar = -np.sum(Y_bar * np.exp(Z), axis=1, keepdims=True)/R**2
    Z_bar = Y_bar * (np.exp(Z)/R) + R_bar * np.exp(Z)
    return Z_bar
```

perceptron

November 6, 2023

0.1 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 ####################################

$\frac{\partial L}{\partial W_k} = x_i$

$\frac{\partial L}{\partial b} = x_i$

$W_k^{new} = W_k^{old} - \eta \cdot \frac{\partial L}{\partial W_k}$

$b^{new} = b^{old} - \eta \cdot \frac{\partial L}{\partial b}$

$\eta = learningrate$

[4]:
```python
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)
```

```
/home/myu/.conda/envs/carla/lib/python3.7/importlib/_bootstrap.py:219:
RuntimeWarning: numpy.ufunc size changed, may indicate binary incompatibility.
```

```
Expected 192 from C header, got 216 from PyObject
  return f(*args, **kwds)
```

[5]:
```
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]
```

[6]:
```
## split the training set and test set
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.3,␣
 ↪random_state=0)
print('X_train_Shape:', X_train.shape)
print('X_test_Shape:',  X_test.shape)
print('y_train_Shape:', y_train.shape)
print('y_test_Shape:',  y_train.shape)

print(type(y_train))
```

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

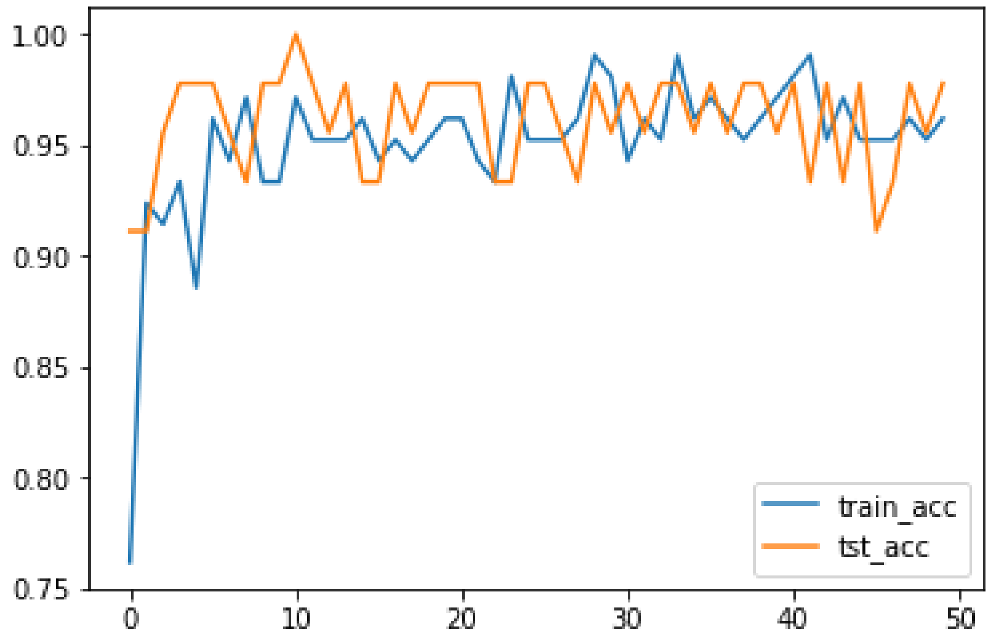
[7]:
```
import numpy as np
import matplotlib.pyplot as plt

class MultiClsPLA(object):

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

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

    def initial_weight(self, weight_dimension, num_cls):
        #########################################
        ##  ToDO: Initialize the weight with   ##
        ##  small std and zero mean gaussian   ##
        #########################################
        weight = np.random.normal(0, 0.01, size=(weight_dimension+1, num_cls))
        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 = 0.0
        for i in range(X_train.shape[0]):
            x = np.concatenate((X_train[i], [1]))  # Add 1 for bias term
            y = np.argmax(np.dot(x, self.weight), axis=0)

            if y != y_train[i]:
                self.weight[:, y_train[i]] += self.lr * x
                self.weight[:, y] -= self.lr * x

            train_acc += 1 if y == y_train[i] else 0

        train_acc /= X_train.shape[0]
        return train_acc

    def test_step(self, X_test, y_test):
        X_test = self.data_preprocessing(data=X_test)
        num_sample = X_test.shape[0]
        test_acc = 0.0

        for i in range(num_sample):
            x = np.concatenate((X_test[i], [1]))
            y = np.argmax(np.dot(x, self.weight), axis=0)
            test_acc += 1 if y == y_test[i] else 0

        test_acc /= num_sample
        return test_acc

    def train(self):
        self.X_train = self.data_preprocessing(data=self.X_train)
        num_sample = self.X_train.shape[0]
        shuffle_index = np.array(range(0, num_sample))

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

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

[8]:
```
np.random.seed(0)

#######################################################
### TODO:
### 1. You need to import the model and pass some parameters.
### 2. Then training the model with some epoches.
### 3. Visualize the training acc and test acc verus epoches
lr = 0.01   #
num_epoch = 50   #
weight_dimension = X_train.shape[1]   #
num_cls = len(set(y_train))
model = MultiClsPLA(X_train, y_train, X_test, y_test, lr, num_epoch,␣
 ↪weight_dimension, num_cls)

#
model.train()

#
```

```
model.vis_acc_curve()
```

```
epoch: 0 traing_acc:0.762 tst_acc:0.911
epoch: 1 traing_acc:0.924 tst_acc:0.911
epoch: 2 traing_acc:0.914 tst_acc:0.956
epoch: 3 traing_acc:0.933 tst_acc:0.978
epoch: 4 traing_acc:0.886 tst_acc:0.978
epoch: 5 traing_acc:0.962 tst_acc:0.978
epoch: 6 traing_acc:0.943 tst_acc:0.956
epoch: 7 traing_acc:0.971 tst_acc:0.933
epoch: 8 traing_acc:0.933 tst_acc:0.978
epoch: 9 traing_acc:0.933 tst_acc:0.978
epoch: 10 traing_acc:0.971 tst_acc:1.000
epoch: 11 traing_acc:0.952 tst_acc:0.978
epoch: 12 traing_acc:0.952 tst_acc:0.956
epoch: 13 traing_acc:0.952 tst_acc:0.978
epoch: 14 traing_acc:0.962 tst_acc:0.933
epoch: 15 traing_acc:0.943 tst_acc:0.933
epoch: 16 traing_acc:0.952 tst_acc:0.978
epoch: 17 traing_acc:0.943 tst_acc:0.956
epoch: 18 traing_acc:0.952 tst_acc:0.978
epoch: 19 traing_acc:0.962 tst_acc:0.978
epoch: 20 traing_acc:0.962 tst_acc:0.978
epoch: 21 traing_acc:0.943 tst_acc:0.978
epoch: 22 traing_acc:0.933 tst_acc:0.933
epoch: 23 traing_acc:0.981 tst_acc:0.933
epoch: 24 traing_acc:0.952 tst_acc:0.978
epoch: 25 traing_acc:0.952 tst_acc:0.978
epoch: 26 traing_acc:0.952 tst_acc:0.956
epoch: 27 traing_acc:0.962 tst_acc:0.933
epoch: 28 traing_acc:0.990 tst_acc:0.978
epoch: 29 traing_acc:0.981 tst_acc:0.956
epoch: 30 traing_acc:0.943 tst_acc:0.978
epoch: 31 traing_acc:0.962 tst_acc:0.956
epoch: 32 traing_acc:0.952 tst_acc:0.978
epoch: 33 traing_acc:0.990 tst_acc:0.978
epoch: 34 traing_acc:0.962 tst_acc:0.956
epoch: 35 traing_acc:0.971 tst_acc:0.978
epoch: 36 traing_acc:0.962 tst_acc:0.956
epoch: 37 traing_acc:0.952 tst_acc:0.978
epoch: 38 traing_acc:0.962 tst_acc:0.978
epoch: 39 traing_acc:0.971 tst_acc:0.956
epoch: 40 traing_acc:0.981 tst_acc:0.978
epoch: 41 traing_acc:0.990 tst_acc:0.933
epoch: 42 traing_acc:0.952 tst_acc:0.978
epoch: 43 traing_acc:0.971 tst_acc:0.933
epoch: 44 traing_acc:0.952 tst_acc:0.978
epoch: 45 traing_acc:0.952 tst_acc:0.911
```

```
epoch: 46 traing_acc:0.952 tst_acc:0.933
epoch: 47 traing_acc:0.962 tst_acc:0.978
epoch: 48 traing_acc:0.952 tst_acc:0.956
epoch: 49 traing_acc:0.962 tst_acc:0.978
```



```
[ ]:
```

knn

November 6, 2023

1 k-Nearest Neighbor (kNN) exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.

The kNN classifier consists of two stages:

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

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

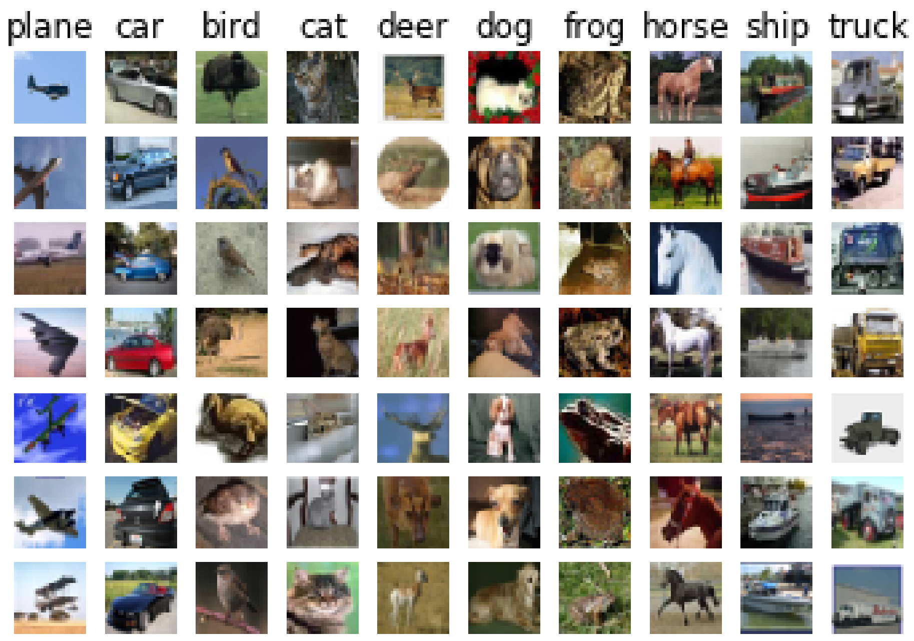
[1]:
```
# Run some setup code for this notebook.

import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

# This is a bit of magic to make matplotlib figures appear inline in the␣
 ↪notebook
# rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

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

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

```
# Cleaning up variables to prevent loading data multiple times (which may cause␣
↪memory 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,)
```

[3]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
↪'ship', '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()
```



[4]:
```
# 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)
```

[5]:
```
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.

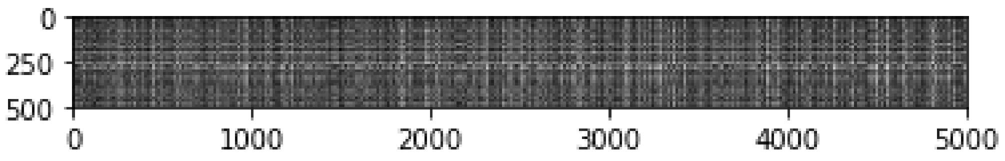
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

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.

[6]:
```
# 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)
```

[7]:
```
# 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?

$Your Answer:$ The large distance, maybe because they have some common background color.

[8]:
```
# 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`:

[9]:
```
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.

$\mathit{Your Answer:}$ *5*

$\mathit{Your Explanation:}$*For 1,2,3,4, because it changes the absolute mean and var, so the relative L1-distance will also change. For 5, no value have changed, so the L1-distance will stay the same.*

[10]:
```
# Now lets speed up distance matrix computation by using partial vectorization
# with one loop. Implement the function compute_distances_one_loop and run the
# code below:
```

```
dists_one = classifier.compute_distances_one_loop(X_test)

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

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

[11]:
```
# 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
```

[12]:
```
# Let's compare how fast the implementations are
def time_function(f, *args):
    """
    Call a function f with args and return the time (in seconds) that it took␣
 ↪to execute.
    """
    import time
    tic = time.time()
    f(*args)
    toc = time.time()
    return toc - tic

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

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

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

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

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

```
Two loop version took 30.635617 seconds
One loop version took 36.857926 seconds
No loop version took 0.348547 seconds
```

1.0.1 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.

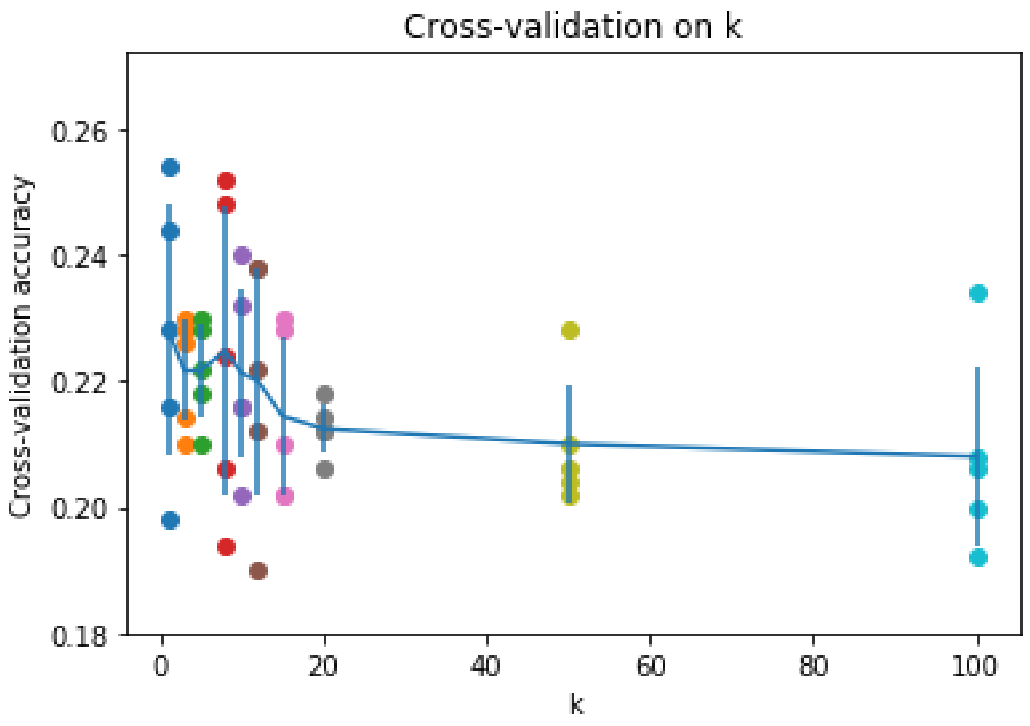
[15]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

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)*****
N = X_train.shape[0]
length = N // num_folds
for i in range(N):
    X_train_folds.append(X_train[i*length: (i+1)*length])
    y_train_folds.append(y_train[i*length: (i+1)*length])
# *****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):
        model = KNearestNeighbor()
        model.train(X_train_folds[i], y_train_folds[i])
        acc_list = []
        acc = np.mean(model.predict(X_test, k) == y_test)
        k_to_accuracies[k].append(acc)
# *****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 = 0.254000
k = 1, accuracy = 0.198000
k = 1, accuracy = 0.228000
k = 1, accuracy = 0.244000
k = 1, accuracy = 0.216000
k = 3, accuracy = 0.230000
k = 3, accuracy = 0.210000
k = 3, accuracy = 0.228000
k = 3, accuracy = 0.226000
k = 3, accuracy = 0.214000
k = 5, accuracy = 0.218000
k = 5, accuracy = 0.228000
k = 5, accuracy = 0.210000
k = 5, accuracy = 0.230000
k = 5, accuracy = 0.222000
k = 8, accuracy = 0.206000
k = 8, accuracy = 0.248000
k = 8, accuracy = 0.224000
k = 8, accuracy = 0.194000
k = 8, accuracy = 0.252000
k = 10, accuracy = 0.202000
k = 10, accuracy = 0.240000
```

```
k = 10, accuracy = 0.216000
k = 10, accuracy = 0.216000
k = 10, accuracy = 0.232000
k = 12, accuracy = 0.190000
k = 12, accuracy = 0.238000
k = 12, accuracy = 0.222000
k = 12, accuracy = 0.212000
k = 12, accuracy = 0.238000
k = 15, accuracy = 0.202000
k = 15, accuracy = 0.230000
k = 15, accuracy = 0.210000
k = 15, accuracy = 0.202000
k = 15, accuracy = 0.228000
k = 20, accuracy = 0.206000
k = 20, accuracy = 0.212000
k = 20, accuracy = 0.218000
k = 20, accuracy = 0.212000
k = 20, accuracy = 0.214000
k = 50, accuracy = 0.206000
k = 50, accuracy = 0.204000
k = 50, accuracy = 0.202000
k = 50, accuracy = 0.210000
k = 50, accuracy = 0.228000
k = 100, accuracy = 0.200000
k = 100, accuracy = 0.206000
k = 100, accuracy = 0.192000
k = 100, accuracy = 0.208000
k = 100, accuracy = 0.234000
```

[16]:
```
# 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()
```



[17]:
```
# 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 = 1

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

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

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

Inline Question 3

Which of the following statements about k-Nearest Neighbor (k-NN) are true in a classification setting, and for all k? Select all that apply. 1. The decision boundary of the k-NN classifier is linear. 2. The training error of a 1-NN will always be lower than that of 5-NN. 3. The test error of a 1-NN will always be lower than that of a 5-NN. 4. The time needed to classify a test example with the k-NN classifier grows with the size of the training set. 5. None of the above.

$Your Answer:$4

$YourExplanation:$*Because we need to compare the testing set with all training set.*

svm

November 6, 2023

1 Multiclass Support Vector Machine exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.

In this exercise you will:

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

[1]:
```
# Run some setup code for this notebook.
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

# This is a bit of magic to make matplotlib figures appear inline in the
# notebook rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

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

1.1 CIFAR-10 Data Loading and Preprocessing

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

# Cleaning up variables to prevent loading data multiple times (which may cause␣
↪memory 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,)
```

[3]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
↪'ship', '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()
```

[4]:
```python
# 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,)
```

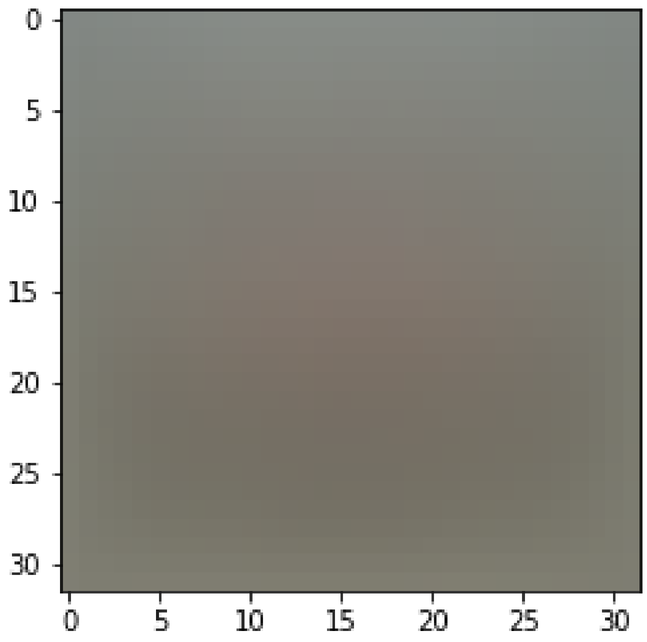
[5]:
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

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

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

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

```
# 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)
```

1.2 SVM Classifier

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

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

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

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

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

```
loss: 8.712269
```

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:

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

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

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

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

```
numerical: 0.462022 analytic: 1.591270, relative error: 5.499699e-01
numerical: 2.822385 analytic: 1.612666, relative error: 2.727632e-01
numerical: -3.725127 analytic: -6.520000, relative error: 2.728002e-01
numerical: -3.233472 analytic: -4.223566, relative error: 1.327731e-01
numerical: 5.064060 analytic: 4.705114, relative error: 3.674269e-02
numerical: -42.718297 analytic: -44.802138, relative error: 2.380977e-02
numerical: -12.763390 analytic: -14.568000, relative error: 6.602702e-02
numerical: 23.120211 analytic: 23.912948, relative error: 1.685485e-02
numerical: -6.249832 analytic: -6.432484, relative error: 1.440215e-02
```

```
numerical: -0.560797 analytic: -2.197030, relative error: 5.933053e-01
numerical: -0.707724 analytic: -0.448628, relative error: 2.240633e-01
numerical: 12.242362 analytic: 13.634125, relative error: 5.378483e-02
numerical: -56.097077 analytic: -57.558069, relative error: 1.285460e-02
numerical: 19.472924 analytic: 22.113520, relative error: 6.349655e-02
numerical: 19.803707 analytic: 20.695416, relative error: 2.201798e-02
numerical: -0.257747 analytic: -0.106115, relative error: 4.167282e-01
numerical: 8.460260 analytic: 8.696922, relative error: 1.379378e-02
numerical: 6.528693 analytic: 6.917492, relative error: 2.891520e-02
numerical: -8.407918 analytic: -11.296405, relative error: 1.465915e-01
numerical: -6.053650 analytic: -0.679731, relative error: 7.981011e-01
```

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 : 1.Yes, it is possible.2. because the right predictionclass brings the "0" gradient. 3. low down the margin and the numerical gradient would be accuracy.

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

from cs231n.classifiers.linear_svm import svm_loss_vectorized
tic = time.time()
loss_vectorized, _ = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

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

```
Naive loss: 8.712269e+00 computed in 0.213207s
Vectorized loss: 8.712269e+00 computed in 0.015558s
difference: 0.000000
```

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

# The naive implementation and the vectorized implementation should match, but
# the vectorized version should still be much faster.
```

```
tic = time.time()
_, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss and gradient: computed in %fs' % (toc - tic))

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

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

```
Naive loss and gradient: computed in 0.214273s
Vectorized loss and gradient: computed in 0.008178s
difference: 287.772188
```

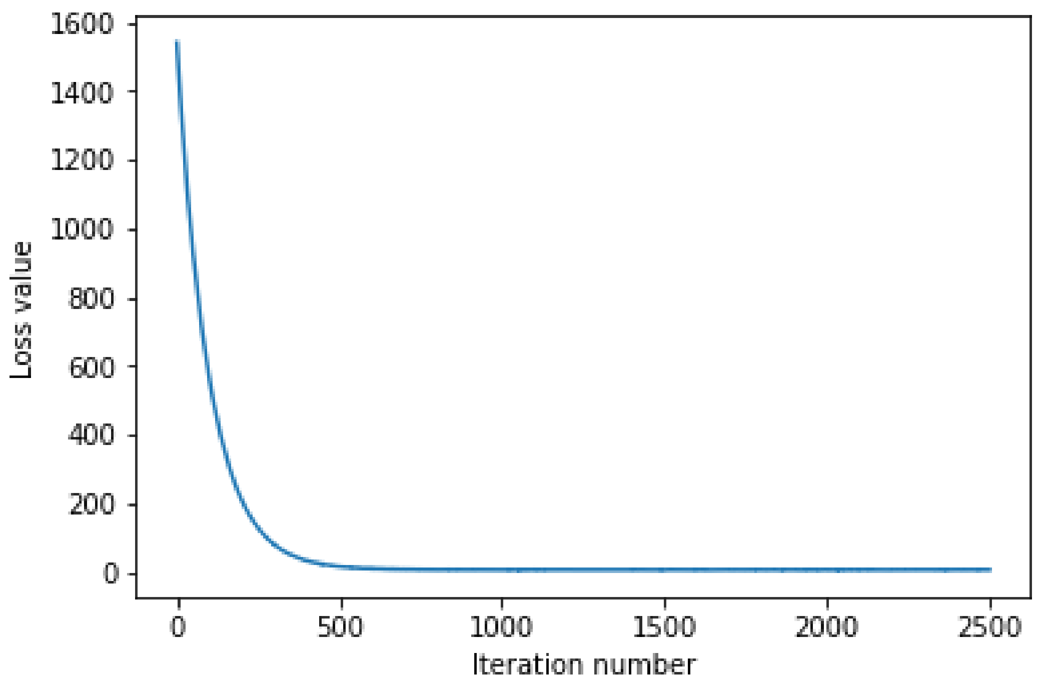
1.2.1 Stochastic Gradient Descent

We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss. Your code for this part will be written inside `cs231n/classifiers/linear_classifier.py`.

[15]:
```
# 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=2500, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 2500: loss 1539.965648
iteration 100 / 2500: loss 561.379683
iteration 200 / 2500: loss 207.386594
iteration 300 / 2500: loss 78.995212
iteration 400 / 2500: loss 32.351515
iteration 500 / 2500: loss 15.276761
iteration 600 / 2500: loss 8.815441
iteration 700 / 2500: loss 7.317820
iteration 800 / 2500: loss 6.649019
iteration 900 / 2500: loss 6.301109
iteration 1000 / 2500: loss 5.924322
iteration 1100 / 2500: loss 5.760378
iteration 1200 / 2500: loss 6.003845
iteration 1300 / 2500: loss 5.814065
iteration 1400 / 2500: loss 5.133910
iteration 1500 / 2500: loss 6.075364
iteration 1600 / 2500: loss 5.915600
iteration 1700 / 2500: loss 5.860075
iteration 1800 / 2500: loss 5.437388
iteration 1900 / 2500: loss 5.473663
iteration 2000 / 2500: loss 5.837305
iteration 2100 / 2500: loss 5.474469
iteration 2200 / 2500: loss 6.162901
iteration 2300 / 2500: loss 5.657615
iteration 2400 / 2500: loss 6.004385
That took 36.945611s
```

[16]:
```python
# A useful debugging strategy is to plot the loss as a function of
# iteration number:
plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```



[17]:
```python
# Write the LinearSVM.predict function and evaluate the performance on both the
# training and validation set
y_train_pred = svm.predict(X_train)
```

```
print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
y_val_pred = svm.predict(X_val)
print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

```
training accuracy: 0.373673
validation accuracy: 0.390000
```

[23]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.39 on the validation set.

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

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

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

# Provided as a reference. You may or may not want to change these␣
 ↪hyperparameters
learning_rates = [1e-7, 2.7e-7, 5e-7]
regularization_strengths = [2.5e4,2.8e4, 3e4, 5e4]

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for lr in learning_rates:
    for reg in regularization_strengths:
        svm = LinearSVM()
        svm.train(X_train, y_train, learning_rate=lr, reg=reg,
                      num_iters=2500, verbose=True)
        y_pre_val = svm.predict(X_val)
        acc_val = np.mean(y_pre_val == y_val)
        y_pre_train = svm.predict(X_train)
        acc_train = np.mean(y_pre_train == y_train)
        results[(lr, reg)] = (acc_train, acc_val)
        if acc_val > best_val:
            best_val = acc_val
            best_svm = svm
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

print('best validation accuracy achieved during cross-validation: %f' %␣
 ↪best_val)
```

```
iteration 0 / 2500: loss 1526.764108
iteration 100 / 2500: loss 555.903409
iteration 200 / 2500: loss 206.748601
iteration 300 / 2500: loss 78.707859
iteration 400 / 2500: loss 32.706547
iteration 500 / 2500: loss 15.322632
iteration 600 / 2500: loss 8.930914
iteration 700 / 2500: loss 6.704234
iteration 800 / 2500: loss 5.647990
iteration 900 / 2500: loss 5.768647
iteration 1000 / 2500: loss 5.984512
iteration 1100 / 2500: loss 5.759987
iteration 1200 / 2500: loss 5.655340
iteration 1300 / 2500: loss 5.893454
iteration 1400 / 2500: loss 5.268225
iteration 1500 / 2500: loss 5.531155
iteration 1600 / 2500: loss 5.315420
iteration 1700 / 2500: loss 5.838575
iteration 1800 / 2500: loss 6.029941
iteration 1900 / 2500: loss 6.637006
iteration 2000 / 2500: loss 5.721638
iteration 2100 / 2500: loss 6.276350
iteration 2200 / 2500: loss 5.776588
iteration 2300 / 2500: loss 5.540626
iteration 2400 / 2500: loss 6.238012
iteration 0 / 2500: loss 1727.438276
iteration 100 / 2500: loss 558.400934
iteration 200 / 2500: loss 183.303270
iteration 300 / 2500: loss 63.565720
iteration 400 / 2500: loss 23.564693
iteration 500 / 2500: loss 12.314244
iteration 600 / 2500: loss 7.406196
iteration 700 / 2500: loss 6.972662
iteration 800 / 2500: loss 6.282835
iteration 900 / 2500: loss 5.495134
iteration 1000 / 2500: loss 5.902560
iteration 1100 / 2500: loss 6.259144
iteration 1200 / 2500: loss 5.465933
iteration 1300 / 2500: loss 5.603724
iteration 1400 / 2500: loss 5.286954
iteration 1500 / 2500: loss 6.092411
iteration 1600 / 2500: loss 5.245575
iteration 1700 / 2500: loss 5.885536
iteration 1800 / 2500: loss 5.856265
iteration 1900 / 2500: loss 5.489966
iteration 2000 / 2500: loss 5.817674
iteration 2100 / 2500: loss 5.500146
iteration 2200 / 2500: loss 5.850321
iteration 2300 / 2500: loss 6.020091
iteration 2400 / 2500: loss 6.220059
iteration 0 / 2500: loss 1862.771412
iteration 100 / 2500: loss 557.241667
iteration 200 / 2500: loss 169.529050
iteration 300 / 2500: loss 54.828325
iteration 400 / 2500: loss 20.231291
iteration 500 / 2500: loss 10.133873
iteration 600 / 2500: loss 6.751382
iteration 700 / 2500: loss 6.104370
iteration 800 / 2500: loss 5.980064
iteration 900 / 2500: loss 6.753961
iteration 1000 / 2500: loss 5.733495
iteration 1100 / 2500: loss 5.894225
iteration 1200 / 2500: loss 5.600322
iteration 1300 / 2500: loss 5.582106
iteration 1400 / 2500: loss 5.546703
iteration 1500 / 2500: loss 5.595681
iteration 1600 / 2500: loss 5.657109
iteration 1700 / 2500: loss 5.790101
iteration 1800 / 2500: loss 5.808692
iteration 1900 / 2500: loss 5.782612
iteration 2000 / 2500: loss 6.070458
iteration 2100 / 2500: loss 5.438751
iteration 2200 / 2500: loss 5.624044
iteration 2300 / 2500: loss 5.413619
iteration 2400 / 2500: loss 6.263974
iteration 0 / 2500: loss 3087.615412
iteration 100 / 2500: loss 413.723687
iteration 200 / 2500: loss 60.228434
iteration 300 / 2500: loss 13.028564
iteration 400 / 2500: loss 7.461951
iteration 500 / 2500: loss 6.052723
iteration 600 / 2500: loss 6.002491
iteration 700 / 2500: loss 6.398235
iteration 800 / 2500: loss 6.250052
iteration 900 / 2500: loss 6.205412
iteration 1000 / 2500: loss 5.785935
iteration 1100 / 2500: loss 6.732863
iteration 1200 / 2500: loss 5.795034
iteration 1300 / 2500: loss 6.181659
iteration 1400 / 2500: loss 5.800003
iteration 1500 / 2500: loss 6.606022
iteration 1600 / 2500: loss 5.947812
iteration 1700 / 2500: loss 5.970044
iteration 1800 / 2500: loss 6.225508
iteration 1900 / 2500: loss 6.120258
iteration 2000 / 2500: loss 5.630256
iteration 2100 / 2500: loss 6.375773
iteration 2200 / 2500: loss 6.276251
iteration 2300 / 2500: loss 6.307940
iteration 2400 / 2500: loss 6.115284
iteration 0 / 2500: loss 1564.727854
iteration 100 / 2500: loss 105.349479
iteration 200 / 2500: loss 12.004695
iteration 300 / 2500: loss 6.290239
iteration 400 / 2500: loss 6.003306
iteration 500 / 2500: loss 5.751160
iteration 600 / 2500: loss 6.160714
iteration 700 / 2500: loss 6.225619
iteration 800 / 2500: loss 5.694208
iteration 900 / 2500: loss 6.295579
iteration 1000 / 2500: loss 5.996834
iteration 1100 / 2500: loss 5.734729
iteration 1200 / 2500: loss 5.736463
iteration 1300 / 2500: loss 6.338441
iteration 1400 / 2500: loss 6.390982
iteration 1500 / 2500: loss 5.824731
iteration 1600 / 2500: loss 6.109472
iteration 1700 / 2500: loss 6.224350
iteration 1800 / 2500: loss 5.943407
iteration 1900 / 2500: loss 6.133244
iteration 2000 / 2500: loss 5.871505
iteration 2100 / 2500: loss 5.726525
iteration 2200 / 2500: loss 5.390790
iteration 2300 / 2500: loss 5.889485
iteration 2400 / 2500: loss 6.022573
iteration 0 / 2500: loss 1729.905332
iteration 100 / 2500: loss 85.785315
iteration 200 / 2500: loss 9.322036
iteration 300 / 2500: loss 5.906160
iteration 400 / 2500: loss 5.672016
iteration 500 / 2500: loss 5.671883
iteration 600 / 2500: loss 5.920673
iteration 700 / 2500: loss 6.550629
iteration 800 / 2500: loss 5.751968
iteration 900 / 2500: loss 5.823675
iteration 1000 / 2500: loss 6.528528
iteration 1100 / 2500: loss 6.631678
iteration 1200 / 2500: loss 5.987235
iteration 1300 / 2500: loss 6.174091
iteration 1400 / 2500: loss 6.198019
iteration 1500 / 2500: loss 6.406457
iteration 1600 / 2500: loss 5.789751
iteration 1700 / 2500: loss 6.106288
iteration 1800 / 2500: loss 5.475280
iteration 1900 / 2500: loss 6.263491
iteration 2000 / 2500: loss 6.135025
iteration 2100 / 2500: loss 6.100293
iteration 2200 / 2500: loss 5.586282
iteration 2300 / 2500: loss 6.330452
iteration 2400 / 2500: loss 6.001261
iteration 0 / 2500: loss 1870.790240
iteration 100 / 2500: loss 75.355762
iteration 200 / 2500: loss 8.446794
iteration 300 / 2500: loss 6.153208
iteration 400 / 2500: loss 5.898120
iteration 500 / 2500: loss 5.740559
iteration 600 / 2500: loss 6.314207
iteration 700 / 2500: loss 6.129737
iteration 800 / 2500: loss 5.801667
iteration 900 / 2500: loss 6.064820
iteration 1000 / 2500: loss 6.937610
iteration 1100 / 2500: loss 6.401892
iteration 1200 / 2500: loss 5.500643
iteration 1300 / 2500: loss 6.114677
iteration 1400 / 2500: loss 5.952831
iteration 1500 / 2500: loss 5.692656
iteration 1600 / 2500: loss 5.854318
iteration 1700 / 2500: loss 5.860182
iteration 1800 / 2500: loss 5.982419
iteration 1900 / 2500: loss 6.004835
iteration 2000 / 2500: loss 6.126762
iteration 2100 / 2500: loss 5.143830
iteration 2200 / 2500: loss 6.128610
iteration 2300 / 2500: loss 6.063196
iteration 2400 / 2500: loss 6.779481
iteration 0 / 2500: loss 3087.924943
iteration 100 / 2500: loss 18.985306
iteration 200 / 2500: loss 6.154246
iteration 300 / 2500: loss 6.350338
iteration 400 / 2500: loss 6.368004
iteration 500 / 2500: loss 6.572740
iteration 600 / 2500: loss 6.427202
iteration 700 / 2500: loss 6.562124
iteration 800 / 2500: loss 6.262650
iteration 900 / 2500: loss 6.714682
iteration 1000 / 2500: loss 6.452524
iteration 1100 / 2500: loss 6.203776
iteration 1200 / 2500: loss 6.801486
iteration 1300 / 2500: loss 6.879437
iteration 1400 / 2500: loss 6.443098
iteration 1500 / 2500: loss 6.260388
iteration 1600 / 2500: loss 6.782349
iteration 1700 / 2500: loss 6.603363
iteration 1800 / 2500: loss 6.400724
iteration 1900 / 2500: loss 6.708523
iteration 2000 / 2500: loss 6.413788
iteration 2100 / 2500: loss 6.512039
iteration 2200 / 2500: loss 6.903329
iteration 2300 / 2500: loss 6.283869
iteration 2400 / 2500: loss 6.502541
iteration 0 / 2500: loss 1556.910936
iteration 100 / 2500: loss 15.545196
iteration 200 / 2500: loss 6.847975
iteration 300 / 2500: loss 6.496135
iteration 400 / 2500: loss 5.947165
iteration 500 / 2500: loss 5.839728
iteration 600 / 2500: loss 7.476450
iteration 700 / 2500: loss 5.900250
iteration 800 / 2500: loss 6.802623
iteration 900 / 2500: loss 6.571946
iteration 1000 / 2500: loss 5.809760
iteration 1100 / 2500: loss 6.195540
iteration 1200 / 2500: loss 6.044200
iteration 1300 / 2500: loss 5.753824
iteration 1400 / 2500: loss 7.276126
iteration 1500 / 2500: loss 5.696942
iteration 1600 / 2500: loss 5.725970
iteration 1700 / 2500: loss 6.405830
iteration 1800 / 2500: loss 6.305769
iteration 1900 / 2500: loss 6.447331
iteration 2000 / 2500: loss 5.967933
iteration 2100 / 2500: loss 6.517507
iteration 2200 / 2500: loss 5.934676
iteration 2300 / 2500: loss 6.386009
iteration 2400 / 2500: loss 6.285869
iteration 0 / 2500: loss 1735.403853
iteration 100 / 2500: loss 12.109051
iteration 200 / 2500: loss 6.475169
iteration 300 / 2500: loss 6.651912
iteration 400 / 2500: loss 5.904546
iteration 500 / 2500: loss 6.952874
iteration 600 / 2500: loss 5.874488
iteration 700 / 2500: loss 6.614702
iteration 800 / 2500: loss 6.276269
iteration 900 / 2500: loss 6.622174
iteration 1000 / 2500: loss 5.834377
iteration 1100 / 2500: loss 6.551499
iteration 1200 / 2500: loss 6.541336
iteration 1300 / 2500: loss 6.182718
iteration 1400 / 2500: loss 6.719306
iteration 1500 / 2500: loss 6.478988
iteration 1600 / 2500: loss 6.408590
iteration 1700 / 2500: loss 6.558326
iteration 1800 / 2500: loss 6.477376
iteration 1900 / 2500: loss 6.312322
iteration 2000 / 2500: loss 6.493144
iteration 2100 / 2500: loss 5.872902
iteration 2200 / 2500: loss 5.963154
iteration 2300 / 2500: loss 6.537677
iteration 2400 / 2500: loss 7.017686
iteration 0 / 2500: loss 1850.733208
iteration 100 / 2500: loss 10.398369
iteration 200 / 2500: loss 6.677977
iteration 300 / 2500: loss 5.918756
iteration 400 / 2500: loss 5.959824
iteration 500 / 2500: loss 6.510411
iteration 600 / 2500: loss 6.144874
iteration 700 / 2500: loss 6.140353
iteration 800 / 2500: loss 6.663407
iteration 900 / 2500: loss 6.511005
iteration 1000 / 2500: loss 6.789212
iteration 1100 / 2500: loss 5.909318
iteration 1200 / 2500: loss 6.480827
iteration 1300 / 2500: loss 5.943624
iteration 1400 / 2500: loss 6.190863
iteration 1500 / 2500: loss 6.531566
iteration 1600 / 2500: loss 6.453937
iteration 1700 / 2500: loss 6.647823
iteration 1800 / 2500: loss 6.761740
iteration 1900 / 2500: loss 6.847304
iteration 2000 / 2500: loss 6.152155
iteration 2100 / 2500: loss 6.570023
iteration 2200 / 2500: loss 6.242002
iteration 2300 / 2500: loss 6.349284
iteration 2400 / 2500: loss 6.451821
iteration 0 / 2500: loss 3095.042501
iteration 100 / 2500: loss 6.634344
iteration 200 / 2500: loss 6.739679
iteration 300 / 2500: loss 6.988874
iteration 400 / 2500: loss 7.367875
iteration 500 / 2500: loss 7.377774
iteration 600 / 2500: loss 6.802317
iteration 700 / 2500: loss 6.756541
iteration 800 / 2500: loss 7.151827
iteration 900 / 2500: loss 6.375207
iteration 1000 / 2500: loss 6.520180
iteration 1100 / 2500: loss 6.809941
iteration 1200 / 2500: loss 6.709783
iteration 1300 / 2500: loss 6.924267
iteration 1400 / 2500: loss 6.812407
iteration 1500 / 2500: loss 6.346697
iteration 1600 / 2500: loss 6.557444
iteration 1700 / 2500: loss 7.078765
iteration 1800 / 2500: loss 6.809614
iteration 1900 / 2500: loss 6.679415
iteration 2000 / 2500: loss 6.725176
iteration 2100 / 2500: loss 6.537507
iteration 2200 / 2500: loss 7.348272
iteration 2300 / 2500: loss 6.686828
iteration 2400 / 2500: loss 6.489666
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.371408 val accuracy: 0.382000
lr 1.000000e-07 reg 2.800000e+04 train accuracy: 0.364449 val accuracy: 0.380000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.366939 val accuracy: 0.372000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.356143 val accuracy: 0.354000
lr 2.700000e-07 reg 2.500000e+04 train accuracy: 0.357694 val accuracy: 0.358000
lr 2.700000e-07 reg 2.800000e+04 train accuracy: 0.346041 val accuracy: 0.356000
lr 2.700000e-07 reg 3.000000e+04 train accuracy: 0.358796 val accuracy: 0.362000
lr 2.700000e-07 reg 5.000000e+04 train accuracy: 0.338224 val accuracy: 0.348000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.322367 val accuracy: 0.322000
lr 5.000000e-07 reg 2.800000e+04 train accuracy: 0.322694 val accuracy: 0.333000
lr 5.000000e-07 reg 3.000000e+04 train accuracy: 0.336857 val accuracy: 0.343000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.301633 val accuracy: 0.317000
best validation accuracy achieved during cross-validation: 0.382000
```

```
[ ]: # Visualize the cross-validation results
     import math
     import pdb

     # pdb.set_trace()

     x_scatter = [math.log10(x[0]) for x in results]
     y_scatter = [math.log10(x[1]) for x in results]

     # plot training accuracy
     marker_size = 100
     colors = [results[x][0] for x in results]
     plt.subplot(2, 1, 1)
     plt.tight_layout(pad=3)
     plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap=plt.cm.coolwarm)
     plt.colorbar()
     plt.xlabel('log learning rate')
     plt.ylabel('log regularization strength')
     plt.title('CIFAR-10 training accuracy')

     # plot validation accuracy
     colors = [results[x][1] for x in results] # default size of markers is 20
     plt.subplot(2, 1, 2)
     plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap=plt.cm.coolwarm)
     plt.colorbar()
     plt.xlabel('log learning rate')
     plt.ylabel('log regularization strength')
     plt.title('CIFAR-10 validation accuracy')
     plt.show()
```

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

```
[ ]: # Visualize the learned weights for each class.
     # Depending on your choice of learning rate and regularization strength, these␣
      ↪may
     # or may not be nice to look at.
     w = best_svm.W[:-1,:] # strip out the bias
     w = w.reshape(32, 32, 3, 10)
     w_min, w_max = np.min(w), np.max(w)
     classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
      ↪'ship', 'truck']
     for i in range(10):
         plt.subplot(2, 5, i + 1)
```

```
    # Rescale the weights to be between 0 and 255
    wimg = 255.0 * (w[:, :, :, i].squeeze() - w_min) / (w_max - w_min)
    plt.imshow(wimg.astype('uint8'))
    plt.axis('off')
    plt.title(classes[i])
```

Inline question 2

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

$Your Answer:$ *The Description: the weight looks like a blur photo of each class and it's also like the combination of each photos.The interpretation: The shape of the image content looks like the original object of the class because the shape of original class has some feature of it. the weight just combine them together.*

softmax

November 6, 2023

1 Softmax exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.

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

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

[1]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading extenrnal modules
# see http://stackoverflow.com/questions/1907993/
 ↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

[2]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000,␣
 ↪num_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the linear classifier. These are the same steps as we used for the
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

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

    # subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

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

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis = 0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image
    X_dev -= mean_image

    # add bias dimension and transform into columns
    X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
    X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
    X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
    X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

    return X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev
```

```
# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev =␣
 ↪get_CIFAR10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
print('dev data shape: ', X_dev.shape)
print('dev labels shape: ', y_dev.shape)
```

```
Train data shape:  (49000, 3073)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3073)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3073)
Test labels shape:  (1000,)
dev data shape:  (500, 3073)
dev labels shape:  (500,)
```

1.1 Softmax Classifier

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

[5]:
```
# 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: 2.355453
sanity check: 2.302585
```

Inline Question 1

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

YourAnswer : Because if a classifier randomly predict what the label is, the distribution should be *uniform distribution, then the probability of 10 classes is 0.1, then cross entropy loss is 1*-log(0.1)*

[6]:
```
# Complete the implementation of softmax_loss_naive and implement a (naive)
# version of the gradient that uses nested loops.
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# As we did for the SVM, use numeric gradient checking as a debugging tool.
# The numeric gradient should be close to the analytic gradient.
from cs231n.gradient_check import grad_check_sparse
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)

# similar to SVM case, do another gradient check with regularization
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 5e1)
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 5e1)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)
```

```
numerical: 2.508709 analytic: 2.508708, relative error: 3.799069e-08
numerical: -0.770382 analytic: -0.770382, relative error: 5.984394e-08
numerical: -0.909556 analytic: -0.909556, relative error: 8.764137e-08
numerical: -1.964704 analytic: -1.964704, relative error: 3.195548e-08
numerical: -1.049105 analytic: -1.049105, relative error: 1.669349e-08
numerical: -0.805085 analytic: -0.805085, relative error: 3.318745e-09
numerical: 1.097290 analytic: 1.097290, relative error: 1.022567e-08
numerical: -2.356726 analytic: -2.356726, relative error: 3.885246e-09
numerical: 1.774974 analytic: 1.774974, relative error: 4.130705e-08
numerical: -1.555162 analytic: -1.555162, relative error: 3.767624e-08
numerical: 0.544930 analytic: 0.544930, relative error: 4.331318e-08
numerical: 1.961087 analytic: 1.961087, relative error: 1.934304e-08
numerical: -0.935372 analytic: -0.935372, relative error: 3.841384e-09
numerical: -1.251169 analytic: -1.251169, relative error: 2.636359e-08
numerical: -0.689613 analytic: -0.689613, relative error: 2.852454e-08
numerical: 0.271505 analytic: 0.271505, relative error: 7.968182e-09
numerical: 0.303463 analytic: 0.303463, relative error: 1.281233e-07
numerical: -0.391398 analytic: -0.391398, relative error: 3.125624e-08
numerical: 0.569887 analytic: 0.569887, relative error: 2.238434e-08
numerical: 0.550485 analytic: 0.550485, relative error: 6.345798e-09
```

[8]:
```
# Now that we have a naive implementation of the softmax loss function and its␣
 ↪gradient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version␣
 ↪should be
# much faster.
tic = time.time()
loss_naive, grad_naive = softmax_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('naive loss: %e computed in %fs' % (loss_naive, toc - tic))
```

```
from cs231n.classifiers.softmax import softmax_loss_vectorized
tic = time.time()
loss_vectorized, grad_vectorized = softmax_loss_vectorized(W, X_dev, y_dev, 0.
 ↪000005)
toc = time.time()
print('vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

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

```
naive loss: 2.355453e+00 computed in 0.253335s
vectorized loss: 2.355453e+00 computed in 0.004535s
Loss difference: 0.000000
Gradient difference: 0.000000
```

[10]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of over 0.35 on the validation set.

from cs231n.classifiers.linear_classifier import Softmax
results = {}
best_val = -1
best_softmax = None

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained softmax classifer in best_softmax.                          #
################################################################################

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

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

for lr in learning_rates:
    for reg in regularization_strengths:
        model = Softmax()
        model.train(X_train, y_train, lr, reg, num_iters=500)
        y_pre_train, y_pre_val = np.mean(model.predict(X_train) == y_train), np.
 ↪mean(model.predict(X_val) == y_val)
        results[(lr, reg)] = (y_pre_train, y_pre_val)
        if y_pre_val > best_val:
            best_val = y_pre_val
            best_softmax = model

# *****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: 0.306327 val accuracy: 0.331000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.307694 val accuracy: 0.325000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.322735 val accuracy: 0.332000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.294286 val accuracy: 0.312000
best validation accuracy achieved during cross-validation: 0.332000
```

[11]:
```
# evaluate on test set
# Evaluate the best softmax on test set
y_test_pred = best_softmax.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('softmax on raw pixels final test set accuracy: %f' % (test_accuracy, ))
```

```
softmax on raw pixels final test set accuracy: 0.339000
```

Inline Question 2 - *True or False*

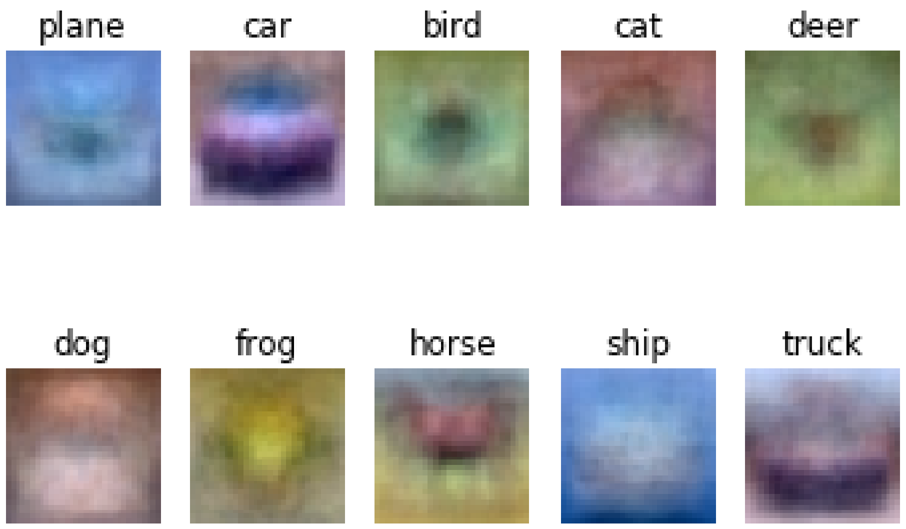
Suppose the overall training loss is defined as the sum of the per-datapoint loss over all training examples. It is possible to add a new datapoint to a training set that would leave the SVM loss unchanged, but this is not the case with the Softmax classifier loss.

$Your Answer$:True

$Your Explanation$: Because SVM classifier is about its margin and the softmax classifier is about probability. If we add a new datapoint, maybe the loss of this point would be zero when the right label's score of this point is margin Δ larger than than others.
But for softmax, add a new data point will cause a new loss $-y_i log(y_i)$

[12]:
```
# Visualize the learned weights for each class
w = best_softmax.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)

w_min, w_max = np.min(w), np.max(w)
```

```
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
for i in range(10):
    plt.subplot(2, 5, i + 1)

    # Rescale the weights to be between 0 and 255
    wimg = 255.0 * (w[:, :, :, i].squeeze() - w_min) / (w_max - w_min)
    plt.imshow(wimg.astype('uint8'))
    plt.axis('off')
    plt.title(classes[i])
```



```
[ ]:
```

two_layer_net

November 6, 2023

1 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.

[1]:
```
# A bit of setup

import numpy as np
import matplotlib.pyplot as plt

from cs231n.classifiers.neural_net import TwoLayerNet

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

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

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

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.

[2]:
```
# 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()
```

2 Forward pass: compute scores

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

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

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

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

3 Forward pass: compute loss

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

[20]:
```
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)))
```

```
(5, 4) (4, 10)
(5, 10) (10, 3)
Difference between your loss and correct loss:
0.018965419606062905
```

4 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:

[22]:
```
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])))
```

```
W1 max relative error: 1.000000e+00
b1 max relative error: 2.738421e-09
W2 max relative error: 1.000000e+00
b2 max relative error: 3.865070e-11
```

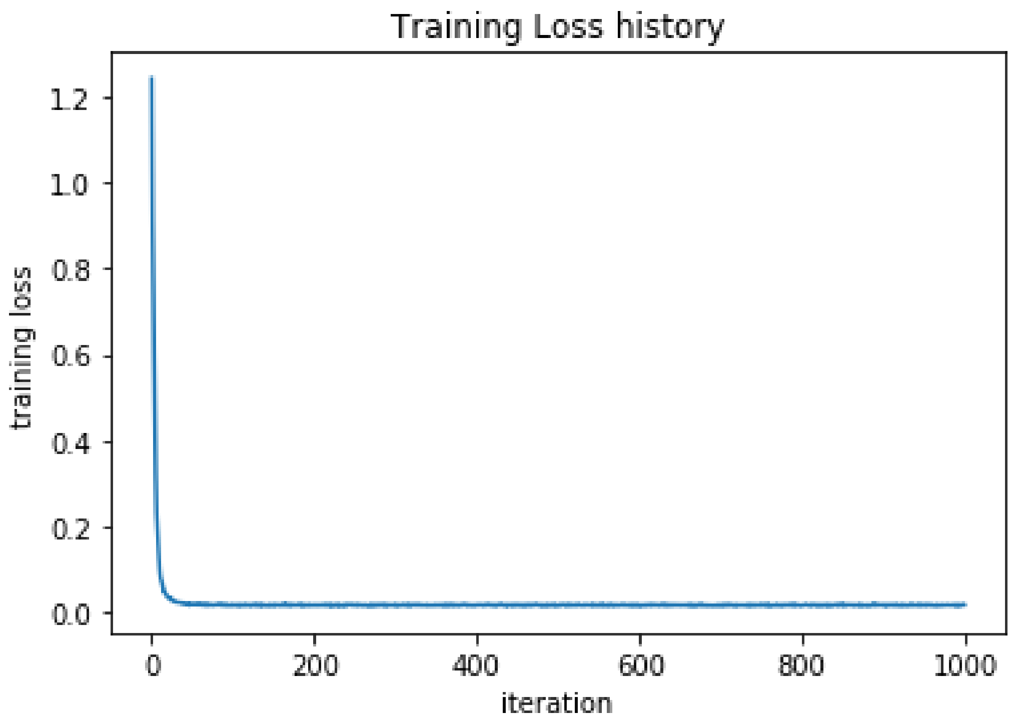
5 Train the network

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.

[23]:
```python
net = init_toy_model()
stats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=1000, 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:  0.0164982537363036
```



6 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.

```
[24]: from cs231n.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the two-layer neural net classifier. These are the same steps as
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

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

    # Subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis=0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image

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

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

```
# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

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

7 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.

[25]:
```
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 2.302749
iteration 100 / 1000: loss 2.301992
iteration 200 / 1000: loss 2.296469
iteration 300 / 1000: loss 2.257815
iteration 400 / 1000: loss 2.190234
iteration 500 / 1000: loss 2.126190
iteration 600 / 1000: loss 2.051932
iteration 700 / 1000: loss 2.014070
iteration 800 / 1000: loss 1.914260
iteration 900 / 1000: loss 1.925339
Validation accuracy:  0.282
```

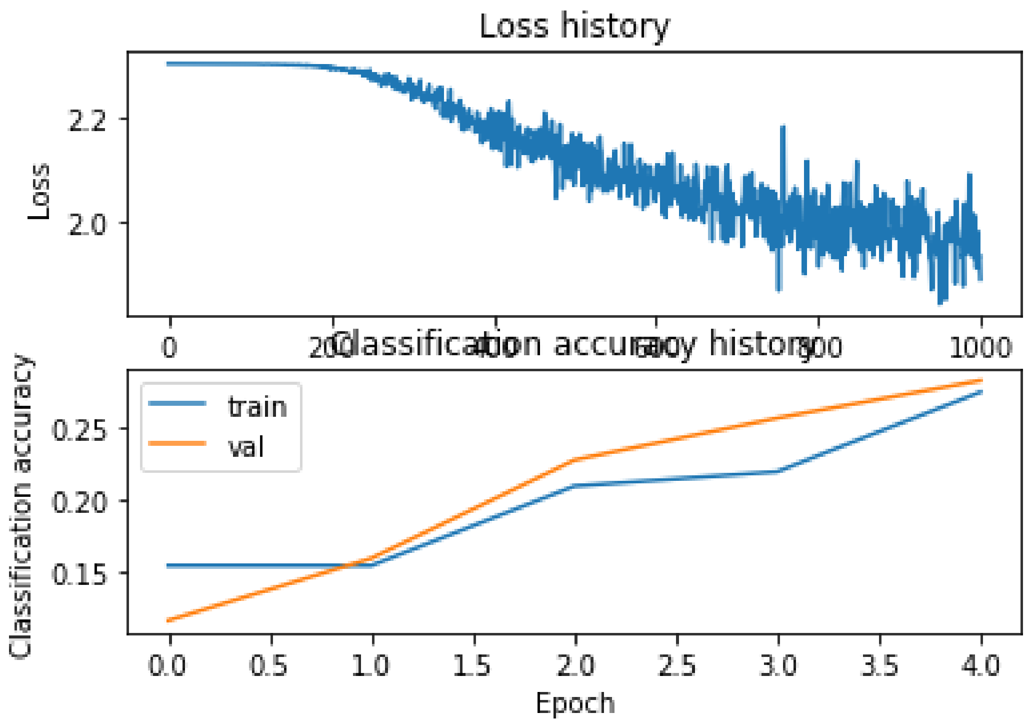
8 Debug the training

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.

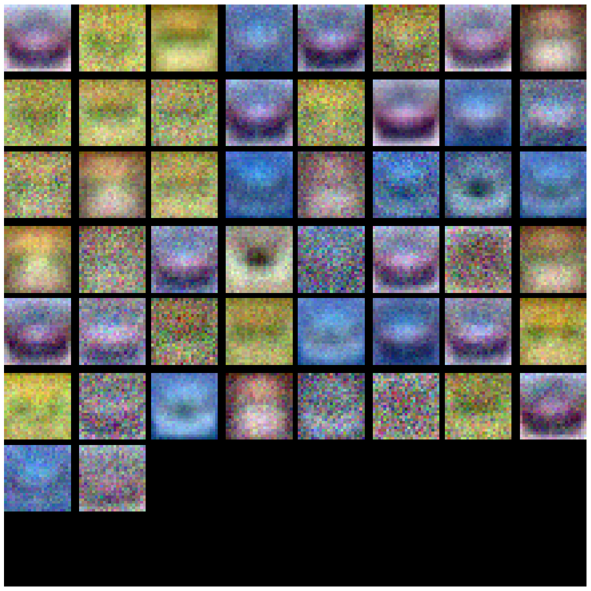
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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.

[26]:
```
# 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()
```



[27]:
```
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)
```



9 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.

$\textit{Your Answer :}$

[28]:
```
best_net = None # store the best model into this

#################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained ␣
 ↪#
# model in best_net.                                                          ␣
 ↪#
#                                                                             ␣
 ↪#
# To help debug your network, it may help to use visualizations similar to the ␣
 ↪#
# ones we used above; these visualizations will have significant qualitative  ␣
 ↪#
# differences from the ones we saw above for the poorly tuned network.        ␣
 ↪#
#                                                                             ␣
 ↪#
# Tweaking hyperparameters by hand can be fun, but you might find it useful to ␣
 ↪#
# write code to sweep through possible combinations of hyperparameters        ␣
 ↪#
# automatically like we did on the previous exercises.                        ␣
 ↪#
#################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

hidden_size = [75, 100, 125]

results = {}
best_val_acc = 0
best_net = None
```

```
learning_rates = np.array([0.7, 0.8, 0.9, 1, 1.1])*1e-3
regularization_strengths = [0.75, 1, 1.25]

print ('running')
for hs in hidden_size:
    for lr in learning_rates:
        for reg in regularization_strengths:
            print( '.'),
            net = TwoLayerNet(input_size, hs, num_classes)
            # Train the network
            stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=1500, batch_size=200,
            learning_rate=lr, learning_rate_decay=0.95,
            reg= reg, verbose=False)
            val_acc = (net.predict(X_val) == y_val).mean()
            if val_acc > best_val_acc:
                best_val_acc = val_acc
                best_net = net
            results[(hs,lr,reg)] = val_acc

print ("finshed")
# Print out results.
for hs,lr, reg in sorted(results):
    val_acc = results[(hs, lr, reg)]
    print ('hs %d lr %e reg %e val accuracy: %f' % (hs, lr, reg,  val_acc))

print ('best validation accuracy achieved during cross-validation: %f' %␣
 ↪best_val_acc
)
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
running
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
```

```
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
finshed
hs 75 lr 7.000000e-04 reg 7.500000e-01 val accuracy: 0.496000
hs 75 lr 7.000000e-04 reg 1.000000e+00 val accuracy: 0.470000
hs 75 lr 7.000000e-04 reg 1.250000e+00 val accuracy: 0.482000
hs 75 lr 8.000000e-04 reg 7.500000e-01 val accuracy: 0.489000
hs 75 lr 8.000000e-04 reg 1.000000e+00 val accuracy: 0.481000
hs 75 lr 8.000000e-04 reg 1.250000e+00 val accuracy: 0.477000
hs 75 lr 9.000000e-04 reg 7.500000e-01 val accuracy: 0.469000
hs 75 lr 9.000000e-04 reg 1.000000e+00 val accuracy: 0.494000
hs 75 lr 9.000000e-04 reg 1.250000e+00 val accuracy: 0.509000
hs 75 lr 1.000000e-03 reg 7.500000e-01 val accuracy: 0.507000
hs 75 lr 1.000000e-03 reg 1.000000e+00 val accuracy: 0.478000
hs 75 lr 1.000000e-03 reg 1.250000e+00 val accuracy: 0.489000
hs 75 lr 1.100000e-03 reg 7.500000e-01 val accuracy: 0.509000
hs 75 lr 1.100000e-03 reg 1.000000e+00 val accuracy: 0.486000
hs 75 lr 1.100000e-03 reg 1.250000e+00 val accuracy: 0.522000
hs 100 lr 7.000000e-04 reg 7.500000e-01 val accuracy: 0.491000
hs 100 lr 7.000000e-04 reg 1.000000e+00 val accuracy: 0.493000
hs 100 lr 7.000000e-04 reg 1.250000e+00 val accuracy: 0.495000
hs 100 lr 8.000000e-04 reg 7.500000e-01 val accuracy: 0.494000
hs 100 lr 8.000000e-04 reg 1.000000e+00 val accuracy: 0.479000
hs 100 lr 8.000000e-04 reg 1.250000e+00 val accuracy: 0.497000
hs 100 lr 9.000000e-04 reg 7.500000e-01 val accuracy: 0.493000
hs 100 lr 9.000000e-04 reg 1.000000e+00 val accuracy: 0.512000
hs 100 lr 9.000000e-04 reg 1.250000e+00 val accuracy: 0.500000
hs 100 lr 1.000000e-03 reg 7.500000e-01 val accuracy: 0.505000
hs 100 lr 1.000000e-03 reg 1.000000e+00 val accuracy: 0.488000
hs 100 lr 1.000000e-03 reg 1.250000e+00 val accuracy: 0.481000
hs 100 lr 1.100000e-03 reg 7.500000e-01 val accuracy: 0.524000
hs 100 lr 1.100000e-03 reg 1.000000e+00 val accuracy: 0.510000
hs 100 lr 1.100000e-03 reg 1.250000e+00 val accuracy: 0.494000
hs 125 lr 7.000000e-04 reg 7.500000e-01 val accuracy: 0.490000
hs 125 lr 7.000000e-04 reg 1.000000e+00 val accuracy: 0.489000
hs 125 lr 7.000000e-04 reg 1.250000e+00 val accuracy: 0.486000
hs 125 lr 8.000000e-04 reg 7.500000e-01 val accuracy: 0.495000
hs 125 lr 8.000000e-04 reg 1.000000e+00 val accuracy: 0.506000
hs 125 lr 8.000000e-04 reg 1.250000e+00 val accuracy: 0.491000
hs 125 lr 9.000000e-04 reg 7.500000e-01 val accuracy: 0.498000
hs 125 lr 9.000000e-04 reg 1.000000e+00 val accuracy: 0.497000
hs 125 lr 9.000000e-04 reg 1.250000e+00 val accuracy: 0.491000
hs 125 lr 1.000000e-03 reg 7.500000e-01 val accuracy: 0.505000
hs 125 lr 1.000000e-03 reg 1.000000e+00 val accuracy: 0.492000
hs 125 lr 1.000000e-03 reg 1.250000e+00 val accuracy: 0.515000
hs 125 lr 1.100000e-03 reg 7.500000e-01 val accuracy: 0.489000
hs 125 lr 1.100000e-03 reg 1.000000e+00 val accuracy: 0.505000
hs 125 lr 1.100000e-03 reg 1.250000e+00 val accuracy: 0.516000
best validation accuracy achieved during cross-validation: 0.524000
```

[29]:
```
# 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:  0.524
```

[30]:
```
# Visualize the weights of the best network
show_net_weights(best_net)
```

10 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%.

[31]:
```python
# 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:  0.512
```

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.

$\textit{Your Answer:}$1,3

$\textit{Your Explanation:}$ Because the lager dataset can solve the generlization error problem and the larger reg could mak the model more weak to prevent overfitting.

features

November 6, 2023

1 Image features exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.

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

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

[1]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt


%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading extenrnal modules
# see http://stackoverflow.com/questions/1907993/
 ↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

1.1 Load data

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

[2]:
```
from cs231n.features import color_histogram_hsv, hog_feature

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'
    # Cleaning up variables to prevent loading data multiple times (which may␣
↪cause memory issue)
    try:
       del X_train, y_train
       del X_test, y_test
       print('Clear previously loaded data.')
    except:
       pass

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

    # Subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]

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

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

1.2 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.

[3]:
```
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)]
```

```python
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
```

1.3 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.

[21]:
```
# 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.       #
```

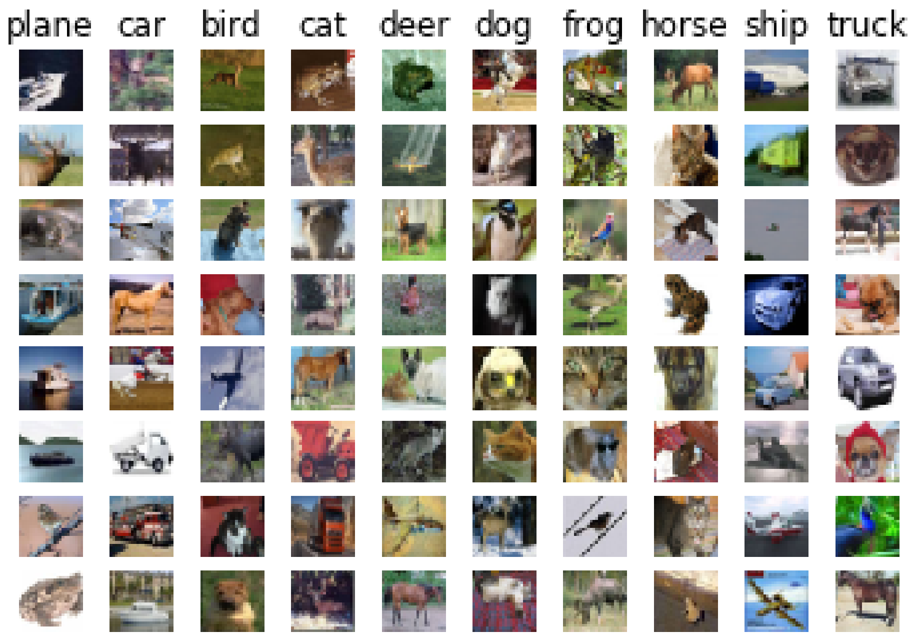
```
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
best_val = 0
for rs in regularization_strengths:
    for lr in learning_rates:
        svm = LinearSVM()
        loss_hist = svm.train(X_train_feats, y_train, lr, rs, num_iters=6000)
        y_train_pred = svm.predict(X_train_feats)
        train_accuracy = np.mean(y_train == y_train_pred)
        y_val_pred = svm.predict(X_val_feats)
        val_accuracy = np.mean(y_val == y_val_pred)
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_svm = svm
        results[(lr,rs)] = 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-09 reg 5.000000e+04 train accuracy: 0.096673 val accuracy: 0.115000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.165204 val accuracy: 0.169000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.414204 val accuracy: 0.411000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.345531 val accuracy: 0.363000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.416163 val accuracy: 0.417000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.401143 val accuracy: 0.385000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.415245 val accuracy: 0.411000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.406714 val accuracy: 0.408000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.318837 val accuracy: 0.304000
best validation accuracy achieved during cross-validation: 0.417000
```

[5]:
```
# Evaluate your trained SVM on the test set: you should be able to get at least␣
  ↪0.40
y_test_pred = best_svm.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print(test_accuracy)
```

```
0.422
```

[6]:
```
# 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()
```



1.3.1 Inline question 1:

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

Your Answer :Since we are using The color histogram features and the HOG features for some misclassification results which have special background or outline, they do make sense. For example, the objects which have a blue background tend to be misclassified as plane and some dogs(trucks) tend be misclassified as cat(car).

1.4 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.

[7]:
```
# Preprocessing: Remove the bias dimension
# Make sure to run this cell only ONCE
print(X_train_feats.shape)
X_train_feats = X_train_feats[:, :-1]
X_val_feats = X_val_feats[:, :-1]
X_test_feats = X_test_feats[:, :-1]

print(X_train_feats.shape)
```

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

[28]:
```
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)

net = TwoLayerNet(input_dim, hidden_dim, num_classes)

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

results = {}
best_val = -1
best_net = None

learning_rates = [1e-2 ,1e-1, 5e-1, 1, 5]
regularization_strengths = [1e-3, 5e-3, 1e-2, 1e-1, 0.5, 1]

for lr in learning_rates:
    for reg in regularization_strengths:
        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=1500, batch_size=200,
        learning_rate=lr, learning_rate_decay=0.95,
        reg= reg, verbose=False)
        val_acc = (net.predict(X_val_feats) == y_val).mean()
        print(val_acc)
        if val_acc > best_val:
            best_val = val_acc
            best_net = net
        results[(lr,reg)] = val_acc
# Print out results.
for lr, reg in sorted(results):
    val_acc = results[(lr, reg)]
    print (f'lr {lr} reg {reg} val accuracy: {val_acc}' )

print ('best validation accuracy achieved during cross-validation: %f' %␣
 ↪best_val
)# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
begin to train…
0.174
begin to train…
0.196
begin to train…
0.189
begin to train…
0.116
begin to train…
0.079
begin to train…
0.087
begin to train…
0.523
begin to train…
0.523
begin to train…
0.516
begin to train…
0.44
begin to train…
0.113
begin to train…
0.113
begin to train…
0.593
begin to train…
```

```
0.575
begin to train…
0.524
begin to train…
0.386
begin to train…
0.196
begin to train…
0.113
begin to train…
0.581
begin to train…
0.56
begin to train…
0.531
begin to train…
0.392
begin to train…
0.203
begin to train…
0.107
begin to train…
```

```
/home/mywork/CS280-Fall23-Assignment1/Homework1_partB/cs231n/classifiers/neural_
net.py:99: RuntimeWarning: divide by zero encountered in log
  loss = -np.sum(np.log(softmax_output[range(N), list(y)]))
/home/mywork/CS280-Fall23-Assignment1/Homework1_partB/cs231n/classifiers/neural_
net.py:97: RuntimeWarning: overflow encountered in subtract
  shift_scores = scores - np.max(scores, axis = 1).reshape(-1,1)
/home/mywork/CS280-Fall23-Assignment1/Homework1_partB/cs231n/classifiers/neural_
net.py:101: RuntimeWarning: overflow encountered in multiply
  loss +=  0.5* reg * (np.sum(W1 * W1) + np.sum(W2 * W2))
/home/mywork/CS280-Fall23-Assignment1/Homework1_partB/cs231n/classifiers/neural_
net.py:97: RuntimeWarning: invalid value encountered in subtract
  shift_scores = scores - np.max(scores, axis = 1).reshape(-1,1)
/home/myu/.conda/envs/carla/lib/python3.7/site-
packages/numpy/core/fromnumeric.py:86: RuntimeWarning: overflow encountered in
reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/home/mywork/CS280-Fall23-Assignment1/Homework1_partB/cs231n/classifiers/neural_
net.py:122: RuntimeWarning: invalid value encountered in greater
  dh_ReLu = (h_output > 0) * dh
```

```
0.087
begin to train…
0.087
begin to train…
```

```
/home/mywork/CS280-Fall23-Assignment1/Homework1_partB/cs231n/classifiers/neural_
```

```
net.py:101: RuntimeWarning: overflow encountered in double_scalars
  loss +=  0.5* reg * (np.sum(W1 * W1) + np.sum(W2 * W2))
```

```
0.087
begin to train…
0.087
begin to train…
0.087
begin to train…
0.087
lr 0.01 reg 0.001 val accuracy: 0.174
lr 0.01 reg 0.005 val accuracy: 0.196
lr 0.01 reg 0.01 val accuracy: 0.189
lr 0.01 reg 0.1 val accuracy: 0.116
lr 0.01 reg 0.5 val accuracy: 0.079
lr 0.01 reg 1 val accuracy: 0.087
lr 0.1 reg 0.001 val accuracy: 0.523
lr 0.1 reg 0.005 val accuracy: 0.523
lr 0.1 reg 0.01 val accuracy: 0.516
lr 0.1 reg 0.1 val accuracy: 0.44
lr 0.1 reg 0.5 val accuracy: 0.113
lr 0.1 reg 1 val accuracy: 0.113
lr 0.5 reg 0.001 val accuracy: 0.593
lr 0.5 reg 0.005 val accuracy: 0.575
lr 0.5 reg 0.01 val accuracy: 0.524
lr 0.5 reg 0.1 val accuracy: 0.386
lr 0.5 reg 0.5 val accuracy: 0.196
lr 0.5 reg 1 val accuracy: 0.113
lr 1 reg 0.001 val accuracy: 0.581
lr 1 reg 0.005 val accuracy: 0.56
lr 1 reg 0.01 val accuracy: 0.531
lr 1 reg 0.1 val accuracy: 0.392
lr 1 reg 0.5 val accuracy: 0.203
lr 1 reg 1 val accuracy: 0.107
lr 5 reg 0.001 val accuracy: 0.087
lr 5 reg 0.005 val accuracy: 0.087
lr 5 reg 0.01 val accuracy: 0.087
lr 5 reg 0.1 val accuracy: 0.087
lr 5 reg 0.5 val accuracy: 0.087
lr 5 reg 1 val accuracy: 0.087
best validation accuracy achieved during cross-validation: 0.593000
```

[29]:
```
# 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)
```

```
0.589
```