CS280 Fall 2022 Assignment 1
Part A

Basics & MLP

March 12, 2023

Name: Bingnan Li

Student ID: 2020533092

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

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

Proof. By the *chain rule*, we have

$$\begin{aligned}
\frac{\partial l(\theta)}{\partial \boldsymbol{\mu}_k} &= \sum_{n=1}^{N} \frac{1}{p(\boldsymbol{x}_n|\theta)} \frac{\partial p(\boldsymbol{x}_n|\theta)}{\partial \boldsymbol{\mu}_k} \\
&= \sum_{n=1}^{N} \frac{1}{p(\boldsymbol{x}_n|\theta)} \frac{\partial \pi_k \mathcal{N}(\boldsymbol{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\partial \boldsymbol{\mu}_k} \\
&= \sum_{n=1}^{N} \frac{\pi_k \mathcal{N}(\boldsymbol{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{p(\boldsymbol{x}_n|\theta)} \frac{\partial(-\frac{1}{2}(\boldsymbol{x}_n - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}_n - \boldsymbol{\mu}_k))}{\partial \boldsymbol{\mu}_k} \\
&= \sum_{n=1}^{N} r_{nk} \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}_n - \boldsymbol{\mu}_k)
\end{aligned}$$

■

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

Proof. For the case without any constriant on π_k, by the *chain rule*, we have:

$$\begin{aligned}
\frac{\partial l(\theta)}{\partial \pi_k} &= \sum_{n=1}^{N} \frac{1}{p(\boldsymbol{x}_n|\theta)} \frac{\partial p(\boldsymbol{x}_n|\theta)}{\partial \pi_k} \\
&= \sum_{n=1}^{N} \frac{1}{p(\boldsymbol{x}_n|\theta)} \mathcal{N}(\boldsymbol{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \\
&= \sum_{n=1}^{N} \frac{r_{nk}}{\pi_k}
\end{aligned}$$

■

2. *Sotfmax & Computation Graph* (10 points).

Recall that the softmax function takes in a vector $(z_1, \dots, z_D)$ and returns a vector $(y_1, \dots, y_D)$. 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.

Solution:

$$\begin{tikzcd}
z_1 \arrow[rr] \arrow[dr] & & y_1 \\
& r \arrow[ur] \arrow[dr] & \\
z_2 \arrow[rr] \arrow[ur] & & y_2
\end{tikzcd}$$

(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.)

Solution:

By the *chain rule* and the forward precess, we have the backprop updates as follows:

$$\bar{r} = \sum_{i=1}^{D} \bar{y}_i \frac{\partial y_i}{\partial r} = -\sum_{i=1}^{D} \bar{y}_i \frac{e^{z_i}}{r^2}$$

$$\bar{z}_i = \bar{y}_i \frac{\partial y_i}{\partial z_i} + \bar{r} \frac{\partial r}{\partial z_i} = \bar{y}_i \frac{e^{z_i}}{r} + \bar{r} e^{z_i}$$

(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 $\boldsymbol{Z}$ 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 $z = (z_1, \dots, z_D)$.
- 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.

Solution:

```python
def VJP(Z, Y_bar):
    exp_Z = np.exp(Z)
    R = np.sum(exp_Z, axis=1)
    R_bar = -np.sum(Y_bar * exp_Z / (R ** 2)[:, None], axis=1)
    Z_bar = Y_bar * exp_Z / R[:, None] + R_bar[:, None] * exp_Z
    return Z_bar
```

perceptron

February 24, 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, $x^i = (x_1^i, \dots x_d^i)$ * 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. $y_{pred}^i = \arg\max_k W_k * x^i + b$
Step2: Compute the gradients for parameters W_k, b. **Derive the parameter update equation Here (5 points)**

TODO: Derive you answer hear ###################################

$$\Delta W_k = \begin{cases} 0, & k = y_{pred}^i = y^i \\ x^i, & k = y_{pred}^i \neq y^i \end{cases} \text{ and } W_k^{new} = W_k^{old} - \eta \Delta W_k$$

$$\Delta b = \begin{cases} 0, & k = y_{pred}^i = y^i \\ 1, & k = y_{pred}^i \neq y^i \end{cases} \text{ and } b^{new} = b^{old} - \eta \Delta b$$

[106]:
```
from sklearn import datasets
import numpy as np
# from sklearn.cross_validation import train_test_split
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import random

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

[107]:
```
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]
```

[108]:
```
## 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'>
```

[109]:
```
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  # N x (D + 1)
        self.y_train = y_train  # N x 1
        self.X_test = X_test
        self.y_test = y_test
        self.weight = self.initial_weight(weight_dimension, num_cls)  # C x (D␣
↪+ 1)
        self.sample_mean = np.mean(self.X_train, 0)
        self.sample_std = np.std(self.X_train, 0)
        self.num_epoch = num_epoch
        self.lr = lr
        self.total_acc_train = []
        self.total_acc_tst = []
    def initial_weight(self, weight_dimension, num_cls):
        weight = None
        #########################################
        ##  ToDO: Initialize the weight with   ##
        ##  small std and zero mean gaussian   ##
        #########################################
        weight = np.random.normal(0, 0.01, (num_cls, weight_dimension))

        return weight

    def data_preprocessing(self, data):
        #####################################
        ##  ToDO: Normalize 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: np.ndarray = X_train[shuffle_idx]
        y_train: np.ndarray = y_train[shuffle_idx]
        train_acc = None
        ##############################################
        ## TODO: to implement the training process  ##
        ## and update the weights                   ##
        ##############################################
        y_pred = np.argmax(X_train @ self.weight.T, axis=0)

        train_acc = np.sum(y_pred == y_train) / y_train.size()
        self.total_acc_train.append(train_acc)
        diff_one_hot = np.eye(X_train.shape[0], self.weight.shape[0])[y !=␣
↪y_pred]  # N x C
        self.weight -= self.lr * diff_one_hot.T @ X_train

        return train_acc

    def test_step(self, X_test, y_test):
        X_test = self.data_preprocessing(data=X_test)
        num_sample = X_test.shape[0]
        test_acc = None

        #########################################
        ##  ToDO: Evaluate the test set and    ##
        ##  return the test acc                ##
        #########################################

        y_pred = np.argmax(X_test @ self.weight.T, axis=0)
        test_acc = np.sum(y_pred == y_test) / y_test.size()
        self.total_acc_tst.append(test_acc)

        return test_acc

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

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

        self.X_train = np.insert(self.X_train, self.X_train.shape[1], 1)
        self.X_test = np.insert(self.X_test, self.X_test.shape[1], 1)

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

[110]:
```
np.random.seed(0)
random.seed(0)
#######################################################
### TODO:
### 1. You need to import the model and pass some parameters.
### 2. Then training the model with some epoches.
### 3. Visualize the training acc and test acc verus epoches
perceptronModel = MultiClsPLA(X_train, y_train, X_test, y_test, lr=1e-3,␣
↪num_epoch=1000,
                              weight_dimension=X_train.shape[1] + 1, num_cls=3)
perceptronModel.train()
```

```
fig = plt.figure()
plt.plot(perceptronModel.total_acc_train)
plt.plot(perceptronModel.total_acc_tst)
```

```
epoch: 0 traing_acc:0.029 tst_acc:0.022
epoch: 1 traing_acc:0.029 tst_acc:0.022
epoch: 2 traing_acc:0.029 tst_acc:0.022
epoch: 3 traing_acc:0.029 tst_acc:0.022
epoch: 4 traing_acc:0.029 tst_acc:0.022
epoch: 5 traing_acc:0.029 tst_acc:0.022
epoch: 6 traing_acc:0.029 tst_acc:0.022
epoch: 7 traing_acc:0.029 tst_acc:0.022
epoch: 8 traing_acc:0.029 tst_acc:0.022
epoch: 9 traing_acc:0.029 tst_acc:0.022
epoch: 10 traing_acc:0.029 tst_acc:0.022
epoch: 11 traing_acc:0.029 tst_acc:0.044
epoch: 12 traing_acc:0.038 tst_acc:0.044
epoch: 13 traing_acc:0.038 tst_acc:0.044
epoch: 14 traing_acc:0.038 tst_acc:0.044
epoch: 15 traing_acc:0.038 tst_acc:0.044
epoch: 16 traing_acc:0.038 tst_acc:0.044
epoch: 17 traing_acc:0.038 tst_acc:0.044
epoch: 18 traing_acc:0.038 tst_acc:0.044
epoch: 19 traing_acc:0.038 tst_acc:0.044
epoch: 20 traing_acc:0.038 tst_acc:0.044
epoch: 21 traing_acc:0.048 tst_acc:0.044
epoch: 22 traing_acc:0.048 tst_acc:0.044
epoch: 23 traing_acc:0.048 tst_acc:0.067
epoch: 24 traing_acc:0.048 tst_acc:0.067
epoch: 25 traing_acc:0.048 tst_acc:0.067
epoch: 26 traing_acc:0.048 tst_acc:0.067
epoch: 27 traing_acc:0.048 tst_acc:0.067
epoch: 28 traing_acc:0.067 tst_acc:0.067
epoch: 29 traing_acc:0.076 tst_acc:0.089
epoch: 30 traing_acc:0.076 tst_acc:0.089
epoch: 31 traing_acc:0.076 tst_acc:0.089
epoch: 32 traing_acc:0.076 tst_acc:0.089
epoch: 33 traing_acc:0.105 tst_acc:0.133
epoch: 34 traing_acc:0.152 tst_acc:0.244
epoch: 35 traing_acc:0.229 tst_acc:0.333
epoch: 36 traing_acc:0.381 tst_acc:0.378
epoch: 37 traing_acc:0.410 tst_acc:0.400
epoch: 38 traing_acc:0.495 tst_acc:0.467
epoch: 39 traing_acc:0.524 tst_acc:0.489
epoch: 40 traing_acc:0.562 tst_acc:0.489
epoch: 41 traing_acc:0.590 tst_acc:0.511
```

```
epoch: 42 traing_acc:0.610 tst_acc:0.533
epoch: 43 traing_acc:0.619 tst_acc:0.533
epoch: 44 traing_acc:0.638 tst_acc:0.533
epoch: 45 traing_acc:0.648 tst_acc:0.533
epoch: 46 traing_acc:0.629 tst_acc:0.533
epoch: 47 traing_acc:0.648 tst_acc:0.533
epoch: 48 traing_acc:0.667 tst_acc:0.556
epoch: 49 traing_acc:0.667 tst_acc:0.556
epoch: 50 traing_acc:0.667 tst_acc:0.578
epoch: 51 traing_acc:0.676 tst_acc:0.600
epoch: 52 traing_acc:0.676 tst_acc:0.622
epoch: 53 traing_acc:0.686 tst_acc:0.622
epoch: 54 traing_acc:0.705 tst_acc:0.622
epoch: 55 traing_acc:0.695 tst_acc:0.622
epoch: 56 traing_acc:0.705 tst_acc:0.622
epoch: 57 traing_acc:0.695 tst_acc:0.622
epoch: 58 traing_acc:0.705 tst_acc:0.622
epoch: 59 traing_acc:0.714 tst_acc:0.644
epoch: 60 traing_acc:0.686 tst_acc:0.644
epoch: 61 traing_acc:0.714 tst_acc:0.644
epoch: 62 traing_acc:0.695 tst_acc:0.644
epoch: 63 traing_acc:0.705 tst_acc:0.667
epoch: 64 traing_acc:0.695 tst_acc:0.667
epoch: 65 traing_acc:0.695 tst_acc:0.644
epoch: 66 traing_acc:0.695 tst_acc:0.644
epoch: 67 traing_acc:0.695 tst_acc:0.644
epoch: 68 traing_acc:0.695 tst_acc:0.644
epoch: 69 traing_acc:0.695 tst_acc:0.644
epoch: 70 traing_acc:0.695 tst_acc:0.644
epoch: 71 traing_acc:0.695 tst_acc:0.644
epoch: 72 traing_acc:0.695 tst_acc:0.644
epoch: 73 traing_acc:0.705 tst_acc:0.644
epoch: 74 traing_acc:0.705 tst_acc:0.644
epoch: 75 traing_acc:0.705 tst_acc:0.644
epoch: 76 traing_acc:0.705 tst_acc:0.644
epoch: 77 traing_acc:0.705 tst_acc:0.644
epoch: 78 traing_acc:0.705 tst_acc:0.644
epoch: 79 traing_acc:0.705 tst_acc:0.644
epoch: 80 traing_acc:0.705 tst_acc:0.644
epoch: 81 traing_acc:0.705 tst_acc:0.667
epoch: 82 traing_acc:0.705 tst_acc:0.667
epoch: 83 traing_acc:0.705 tst_acc:0.667
epoch: 84 traing_acc:0.695 tst_acc:0.644
epoch: 85 traing_acc:0.695 tst_acc:0.644
epoch: 86 traing_acc:0.705 tst_acc:0.644
epoch: 87 traing_acc:0.686 tst_acc:0.644
epoch: 88 traing_acc:0.695 tst_acc:0.644
epoch: 89 traing_acc:0.695 tst_acc:0.644
```

```
epoch: 90 traing_acc:0.695 tst_acc:0.644
epoch: 91 traing_acc:0.686 tst_acc:0.644
epoch: 92 traing_acc:0.695 tst_acc:0.644
epoch: 93 traing_acc:0.695 tst_acc:0.644
epoch: 94 traing_acc:0.705 tst_acc:0.644
epoch: 95 traing_acc:0.695 tst_acc:0.644
epoch: 96 traing_acc:0.705 tst_acc:0.644
epoch: 97 traing_acc:0.686 tst_acc:0.644
epoch: 98 traing_acc:0.686 tst_acc:0.644
epoch: 99 traing_acc:0.714 tst_acc:0.644
epoch: 100 traing_acc:0.686 tst_acc:0.644
epoch: 101 traing_acc:0.695 tst_acc:0.644
epoch: 102 traing_acc:0.686 tst_acc:0.622
epoch: 103 traing_acc:0.695 tst_acc:0.622
epoch: 104 traing_acc:0.705 tst_acc:0.622
epoch: 105 traing_acc:0.714 tst_acc:0.622
epoch: 106 traing_acc:0.714 tst_acc:0.622
epoch: 107 traing_acc:0.714 tst_acc:0.622
epoch: 108 traing_acc:0.714 tst_acc:0.622
epoch: 109 traing_acc:0.714 tst_acc:0.622
epoch: 110 traing_acc:0.714 tst_acc:0.622
epoch: 111 traing_acc:0.705 tst_acc:0.622
epoch: 112 traing_acc:0.714 tst_acc:0.622
epoch: 113 traing_acc:0.705 tst_acc:0.644
epoch: 114 traing_acc:0.705 tst_acc:0.644
epoch: 115 traing_acc:0.714 tst_acc:0.644
epoch: 116 traing_acc:0.714 tst_acc:0.644
epoch: 117 traing_acc:0.705 tst_acc:0.644
epoch: 118 traing_acc:0.724 tst_acc:0.644
epoch: 119 traing_acc:0.714 tst_acc:0.644
epoch: 120 traing_acc:0.714 tst_acc:0.644
epoch: 121 traing_acc:0.705 tst_acc:0.644
epoch: 122 traing_acc:0.724 tst_acc:0.644
epoch: 123 traing_acc:0.724 tst_acc:0.644
epoch: 124 traing_acc:0.733 tst_acc:0.644
epoch: 125 traing_acc:0.714 tst_acc:0.644
epoch: 126 traing_acc:0.724 tst_acc:0.644
epoch: 127 traing_acc:0.733 tst_acc:0.644
epoch: 128 traing_acc:0.724 tst_acc:0.644
epoch: 129 traing_acc:0.714 tst_acc:0.644
epoch: 130 traing_acc:0.733 tst_acc:0.644
epoch: 131 traing_acc:0.733 tst_acc:0.644
epoch: 132 traing_acc:0.695 tst_acc:0.644
epoch: 133 traing_acc:0.733 tst_acc:0.644
epoch: 134 traing_acc:0.733 tst_acc:0.644
epoch: 135 traing_acc:0.724 tst_acc:0.644
epoch: 136 traing_acc:0.705 tst_acc:0.644
epoch: 137 traing_acc:0.724 tst_acc:0.644
```

```
epoch: 138 traing_acc:0.714 tst_acc:0.667
epoch: 139 traing_acc:0.724 tst_acc:0.667
epoch: 140 traing_acc:0.705 tst_acc:0.667
epoch: 141 traing_acc:0.724 tst_acc:0.667
epoch: 142 traing_acc:0.705 tst_acc:0.667
epoch: 143 traing_acc:0.724 tst_acc:0.689
epoch: 144 traing_acc:0.705 tst_acc:0.689
epoch: 145 traing_acc:0.724 tst_acc:0.689
epoch: 146 traing_acc:0.705 tst_acc:0.689
epoch: 147 traing_acc:0.733 tst_acc:0.667
epoch: 148 traing_acc:0.724 tst_acc:0.689
epoch: 149 traing_acc:0.724 tst_acc:0.689
epoch: 150 traing_acc:0.743 tst_acc:0.667
epoch: 151 traing_acc:0.724 tst_acc:0.689
epoch: 152 traing_acc:0.733 tst_acc:0.667
epoch: 153 traing_acc:0.752 tst_acc:0.689
epoch: 154 traing_acc:0.733 tst_acc:0.689
epoch: 155 traing_acc:0.752 tst_acc:0.689
epoch: 156 traing_acc:0.752 tst_acc:0.689
epoch: 157 traing_acc:0.762 tst_acc:0.689
epoch: 158 traing_acc:0.752 tst_acc:0.689
epoch: 159 traing_acc:0.762 tst_acc:0.689
epoch: 160 traing_acc:0.762 tst_acc:0.667
epoch: 161 traing_acc:0.743 tst_acc:0.689
epoch: 162 traing_acc:0.762 tst_acc:0.689
epoch: 163 traing_acc:0.762 tst_acc:0.689
epoch: 164 traing_acc:0.762 tst_acc:0.689
epoch: 165 traing_acc:0.762 tst_acc:0.711
epoch: 166 traing_acc:0.762 tst_acc:0.689
epoch: 167 traing_acc:0.781 tst_acc:0.689
epoch: 168 traing_acc:0.781 tst_acc:0.689
epoch: 169 traing_acc:0.800 tst_acc:0.689
epoch: 170 traing_acc:0.790 tst_acc:0.711
epoch: 171 traing_acc:0.800 tst_acc:0.689
epoch: 172 traing_acc:0.781 tst_acc:0.711
epoch: 173 traing_acc:0.810 tst_acc:0.689
epoch: 174 traing_acc:0.790 tst_acc:0.711
epoch: 175 traing_acc:0.819 tst_acc:0.711
epoch: 176 traing_acc:0.800 tst_acc:0.733
epoch: 177 traing_acc:0.829 tst_acc:0.711
epoch: 178 traing_acc:0.810 tst_acc:0.733
epoch: 179 traing_acc:0.810 tst_acc:0.733
epoch: 180 traing_acc:0.829 tst_acc:0.778
epoch: 181 traing_acc:0.810 tst_acc:0.778
epoch: 182 traing_acc:0.829 tst_acc:0.778
epoch: 183 traing_acc:0.829 tst_acc:0.778
epoch: 184 traing_acc:0.829 tst_acc:0.778
epoch: 185 traing_acc:0.819 tst_acc:0.778
```

```
epoch: 186 traing_acc:0.829 tst_acc:0.778
epoch: 187 traing_acc:0.838 tst_acc:0.778
epoch: 188 traing_acc:0.829 tst_acc:0.778
epoch: 189 traing_acc:0.838 tst_acc:0.778
epoch: 190 traing_acc:0.857 tst_acc:0.778
epoch: 191 traing_acc:0.838 tst_acc:0.778
epoch: 192 traing_acc:0.867 tst_acc:0.800
epoch: 193 traing_acc:0.857 tst_acc:0.800
epoch: 194 traing_acc:0.867 tst_acc:0.800
epoch: 195 traing_acc:0.886 tst_acc:0.800
epoch: 196 traing_acc:0.886 tst_acc:0.800
epoch: 197 traing_acc:0.886 tst_acc:0.800
epoch: 198 traing_acc:0.895 tst_acc:0.800
epoch: 199 traing_acc:0.886 tst_acc:0.800
epoch: 200 traing_acc:0.895 tst_acc:0.800
epoch: 201 traing_acc:0.895 tst_acc:0.800
epoch: 202 traing_acc:0.886 tst_acc:0.800
epoch: 203 traing_acc:0.886 tst_acc:0.800
epoch: 204 traing_acc:0.886 tst_acc:0.800
epoch: 205 traing_acc:0.886 tst_acc:0.800
epoch: 206 traing_acc:0.895 tst_acc:0.800
epoch: 207 traing_acc:0.895 tst_acc:0.822
epoch: 208 traing_acc:0.886 tst_acc:0.800
epoch: 209 traing_acc:0.895 tst_acc:0.822
epoch: 210 traing_acc:0.895 tst_acc:0.778
epoch: 211 traing_acc:0.895 tst_acc:0.778
epoch: 212 traing_acc:0.886 tst_acc:0.800
epoch: 213 traing_acc:0.895 tst_acc:0.778
epoch: 214 traing_acc:0.895 tst_acc:0.822
epoch: 215 traing_acc:0.905 tst_acc:0.822
epoch: 216 traing_acc:0.895 tst_acc:0.822
epoch: 217 traing_acc:0.895 tst_acc:0.844
epoch: 218 traing_acc:0.924 tst_acc:0.822
epoch: 219 traing_acc:0.905 tst_acc:0.867
epoch: 220 traing_acc:0.924 tst_acc:0.822
epoch: 221 traing_acc:0.905 tst_acc:0.867
epoch: 222 traing_acc:0.914 tst_acc:0.867
epoch: 223 traing_acc:0.924 tst_acc:0.822
epoch: 224 traing_acc:0.905 tst_acc:0.867
epoch: 225 traing_acc:0.924 tst_acc:0.867
epoch: 226 traing_acc:0.914 tst_acc:0.867
epoch: 227 traing_acc:0.924 tst_acc:0.822
epoch: 228 traing_acc:0.905 tst_acc:0.867
epoch: 229 traing_acc:0.924 tst_acc:0.867
epoch: 230 traing_acc:0.924 tst_acc:0.867
epoch: 231 traing_acc:0.924 tst_acc:0.867
epoch: 232 traing_acc:0.914 tst_acc:0.867
epoch: 233 traing_acc:0.914 tst_acc:0.867
```

```
epoch: 234 traing_acc:0.914 tst_acc:0.867
epoch: 235 traing_acc:0.924 tst_acc:0.867
epoch: 236 traing_acc:0.924 tst_acc:0.867
epoch: 237 traing_acc:0.924 tst_acc:0.867
epoch: 238 traing_acc:0.924 tst_acc:0.867
epoch: 239 traing_acc:0.933 tst_acc:0.867
epoch: 240 traing_acc:0.924 tst_acc:0.867
epoch: 241 traing_acc:0.933 tst_acc:0.867
epoch: 242 traing_acc:0.933 tst_acc:0.867
epoch: 243 traing_acc:0.933 tst_acc:0.867
epoch: 244 traing_acc:0.933 tst_acc:0.867
epoch: 245 traing_acc:0.943 tst_acc:0.867
epoch: 246 traing_acc:0.933 tst_acc:0.867
epoch: 247 traing_acc:0.943 tst_acc:0.867
epoch: 248 traing_acc:0.924 tst_acc:0.867
epoch: 249 traing_acc:0.943 tst_acc:0.867
epoch: 250 traing_acc:0.943 tst_acc:0.867
epoch: 251 traing_acc:0.933 tst_acc:0.867
epoch: 252 traing_acc:0.943 tst_acc:0.867
epoch: 253 traing_acc:0.933 tst_acc:0.867
epoch: 254 traing_acc:0.943 tst_acc:0.867
epoch: 255 traing_acc:0.933 tst_acc:0.867
epoch: 256 traing_acc:0.933 tst_acc:0.867
epoch: 257 traing_acc:0.933 tst_acc:0.867
epoch: 258 traing_acc:0.943 tst_acc:0.867
epoch: 259 traing_acc:0.933 tst_acc:0.867
epoch: 260 traing_acc:0.943 tst_acc:0.867
epoch: 261 traing_acc:0.943 tst_acc:0.867
epoch: 262 traing_acc:0.933 tst_acc:0.867
epoch: 263 traing_acc:0.943 tst_acc:0.867
epoch: 264 traing_acc:0.924 tst_acc:0.889
epoch: 265 traing_acc:0.943 tst_acc:0.867
epoch: 266 traing_acc:0.933 tst_acc:0.867
epoch: 267 traing_acc:0.943 tst_acc:0.867
epoch: 268 traing_acc:0.933 tst_acc:0.867
epoch: 269 traing_acc:0.943 tst_acc:0.867
epoch: 270 traing_acc:0.933 tst_acc:0.867
epoch: 271 traing_acc:0.943 tst_acc:0.867
epoch: 272 traing_acc:0.933 tst_acc:0.889
epoch: 273 traing_acc:0.943 tst_acc:0.889
epoch: 274 traing_acc:0.943 tst_acc:0.889
epoch: 275 traing_acc:0.933 tst_acc:0.889
epoch: 276 traing_acc:0.952 tst_acc:0.889
epoch: 277 traing_acc:0.943 tst_acc:0.889
epoch: 278 traing_acc:0.943 tst_acc:0.889
epoch: 279 traing_acc:0.933 tst_acc:0.889
epoch: 280 traing_acc:0.933 tst_acc:0.889
epoch: 281 traing_acc:0.952 tst_acc:0.889
```

```
epoch: 282 traing_acc:0.933 tst_acc:0.889
epoch: 283 traing_acc:0.943 tst_acc:0.889
epoch: 284 traing_acc:0.952 tst_acc:0.889
epoch: 285 traing_acc:0.933 tst_acc:0.889
epoch: 286 traing_acc:0.952 tst_acc:0.889
epoch: 287 traing_acc:0.924 tst_acc:0.889
epoch: 288 traing_acc:0.943 tst_acc:0.889
epoch: 289 traing_acc:0.943 tst_acc:0.889
epoch: 290 traing_acc:0.943 tst_acc:0.889
epoch: 291 traing_acc:0.952 tst_acc:0.889
epoch: 292 traing_acc:0.933 tst_acc:0.889
epoch: 293 traing_acc:0.943 tst_acc:0.889
epoch: 294 traing_acc:0.943 tst_acc:0.889
epoch: 295 traing_acc:0.933 tst_acc:0.889
epoch: 296 traing_acc:0.952 tst_acc:0.889
epoch: 297 traing_acc:0.933 tst_acc:0.889
epoch: 298 traing_acc:0.943 tst_acc:0.889
epoch: 299 traing_acc:0.943 tst_acc:0.889
epoch: 300 traing_acc:0.943 tst_acc:0.889
epoch: 301 traing_acc:0.952 tst_acc:0.889
epoch: 302 traing_acc:0.933 tst_acc:0.889
epoch: 303 traing_acc:0.933 tst_acc:0.889
epoch: 304 traing_acc:0.943 tst_acc:0.889
epoch: 305 traing_acc:0.943 tst_acc:0.889
epoch: 306 traing_acc:0.943 tst_acc:0.889
epoch: 307 traing_acc:0.943 tst_acc:0.889
epoch: 308 traing_acc:0.943 tst_acc:0.889
epoch: 309 traing_acc:0.943 tst_acc:0.889
epoch: 310 traing_acc:0.943 tst_acc:0.889
epoch: 311 traing_acc:0.933 tst_acc:0.889
epoch: 312 traing_acc:0.943 tst_acc:0.889
epoch: 313 traing_acc:0.943 tst_acc:0.889
epoch: 314 traing_acc:0.943 tst_acc:0.889
epoch: 315 traing_acc:0.943 tst_acc:0.889
epoch: 316 traing_acc:0.943 tst_acc:0.889
epoch: 317 traing_acc:0.943 tst_acc:0.889
epoch: 318 traing_acc:0.924 tst_acc:0.889
epoch: 319 traing_acc:0.952 tst_acc:0.889
epoch: 320 traing_acc:0.943 tst_acc:0.889
epoch: 321 traing_acc:0.943 tst_acc:0.889
epoch: 322 traing_acc:0.943 tst_acc:0.889
epoch: 323 traing_acc:0.933 tst_acc:0.889
epoch: 324 traing_acc:0.952 tst_acc:0.889
epoch: 325 traing_acc:0.943 tst_acc:0.889
epoch: 326 traing_acc:0.933 tst_acc:0.889
epoch: 327 traing_acc:0.943 tst_acc:0.889
epoch: 328 traing_acc:0.933 tst_acc:0.889
epoch: 329 traing_acc:0.952 tst_acc:0.889
```

```
epoch: 330 traing_acc:0.943 tst_acc:0.889
epoch: 331 traing_acc:0.943 tst_acc:0.889
epoch: 332 traing_acc:0.943 tst_acc:0.889
epoch: 333 traing_acc:0.933 tst_acc:0.889
epoch: 334 traing_acc:0.943 tst_acc:0.889
epoch: 335 traing_acc:0.933 tst_acc:0.889
epoch: 336 traing_acc:0.952 tst_acc:0.889
epoch: 337 traing_acc:0.943 tst_acc:0.889
epoch: 338 traing_acc:0.933 tst_acc:0.889
epoch: 339 traing_acc:0.943 tst_acc:0.889
epoch: 340 traing_acc:0.943 tst_acc:0.889
epoch: 341 traing_acc:0.933 tst_acc:0.889
epoch: 342 traing_acc:0.943 tst_acc:0.889
epoch: 343 traing_acc:0.943 tst_acc:0.889
epoch: 344 traing_acc:0.943 tst_acc:0.889
epoch: 345 traing_acc:0.962 tst_acc:0.889
epoch: 346 traing_acc:0.943 tst_acc:0.889
epoch: 347 traing_acc:0.952 tst_acc:0.889
epoch: 348 traing_acc:0.933 tst_acc:0.889
epoch: 349 traing_acc:0.933 tst_acc:0.889
epoch: 350 traing_acc:0.943 tst_acc:0.889
epoch: 351 traing_acc:0.933 tst_acc:0.889
epoch: 352 traing_acc:0.952 tst_acc:0.889
epoch: 353 traing_acc:0.933 tst_acc:0.889
epoch: 354 traing_acc:0.962 tst_acc:0.911
epoch: 355 traing_acc:0.943 tst_acc:0.911
epoch: 356 traing_acc:0.943 tst_acc:0.911
epoch: 357 traing_acc:0.943 tst_acc:0.911
epoch: 358 traing_acc:0.943 tst_acc:0.911
epoch: 359 traing_acc:0.933 tst_acc:0.911
epoch: 360 traing_acc:0.952 tst_acc:0.911
epoch: 361 traing_acc:0.933 tst_acc:0.911
epoch: 362 traing_acc:0.943 tst_acc:0.911
epoch: 363 traing_acc:0.943 tst_acc:0.911
epoch: 364 traing_acc:0.943 tst_acc:0.911
epoch: 365 traing_acc:0.943 tst_acc:0.911
epoch: 366 traing_acc:0.943 tst_acc:0.911
epoch: 367 traing_acc:0.952 tst_acc:0.911
epoch: 368 traing_acc:0.943 tst_acc:0.911
epoch: 369 traing_acc:0.943 tst_acc:0.911
epoch: 370 traing_acc:0.943 tst_acc:0.911
epoch: 371 traing_acc:0.924 tst_acc:0.911
epoch: 372 traing_acc:0.943 tst_acc:0.911
epoch: 373 traing_acc:0.952 tst_acc:0.911
epoch: 374 traing_acc:0.933 tst_acc:0.911
epoch: 375 traing_acc:0.952 tst_acc:0.911
epoch: 376 traing_acc:0.952 tst_acc:0.911
epoch: 377 traing_acc:0.943 tst_acc:0.911
```

```
epoch: 378 traing_acc:0.933 tst_acc:0.911
epoch: 379 traing_acc:0.943 tst_acc:0.911
epoch: 380 traing_acc:0.943 tst_acc:0.911
epoch: 381 traing_acc:0.943 tst_acc:0.911
epoch: 382 traing_acc:0.943 tst_acc:0.911
epoch: 383 traing_acc:0.943 tst_acc:0.911
epoch: 384 traing_acc:0.943 tst_acc:0.911
epoch: 385 traing_acc:0.952 tst_acc:0.911
epoch: 386 traing_acc:0.943 tst_acc:0.911
epoch: 387 traing_acc:0.943 tst_acc:0.911
epoch: 388 traing_acc:0.943 tst_acc:0.911
epoch: 389 traing_acc:0.933 tst_acc:0.911
epoch: 390 traing_acc:0.943 tst_acc:0.911
epoch: 391 traing_acc:0.952 tst_acc:0.911
epoch: 392 traing_acc:0.933 tst_acc:0.911
epoch: 393 traing_acc:0.962 tst_acc:0.911
epoch: 394 traing_acc:0.943 tst_acc:0.911
epoch: 395 traing_acc:0.952 tst_acc:0.911
epoch: 396 traing_acc:0.933 tst_acc:0.911
epoch: 397 traing_acc:0.933 tst_acc:0.911
epoch: 398 traing_acc:0.943 tst_acc:0.911
epoch: 399 traing_acc:0.943 tst_acc:0.911
epoch: 400 traing_acc:0.943 tst_acc:0.911
epoch: 401 traing_acc:0.943 tst_acc:0.911
epoch: 402 traing_acc:0.943 tst_acc:0.911
epoch: 403 traing_acc:0.952 tst_acc:0.911
epoch: 404 traing_acc:0.943 tst_acc:0.911
epoch: 405 traing_acc:0.952 tst_acc:0.911
epoch: 406 traing_acc:0.952 tst_acc:0.911
epoch: 407 traing_acc:0.943 tst_acc:0.911
epoch: 408 traing_acc:0.943 tst_acc:0.911
epoch: 409 traing_acc:0.943 tst_acc:0.911
epoch: 410 traing_acc:0.952 tst_acc:0.911
epoch: 411 traing_acc:0.943 tst_acc:0.911
epoch: 412 traing_acc:0.962 tst_acc:0.911
epoch: 413 traing_acc:0.952 tst_acc:0.911
epoch: 414 traing_acc:0.952 tst_acc:0.911
epoch: 415 traing_acc:0.943 tst_acc:0.911
epoch: 416 traing_acc:0.933 tst_acc:0.911
epoch: 417 traing_acc:0.943 tst_acc:0.911
epoch: 418 traing_acc:0.943 tst_acc:0.911
epoch: 419 traing_acc:0.943 tst_acc:0.911
epoch: 420 traing_acc:0.933 tst_acc:0.911
epoch: 421 traing_acc:0.962 tst_acc:0.911
epoch: 422 traing_acc:0.952 tst_acc:0.911
epoch: 423 traing_acc:0.952 tst_acc:0.911
epoch: 424 traing_acc:0.943 tst_acc:0.911
epoch: 425 traing_acc:0.943 tst_acc:0.911
```

```
epoch: 426 traing_acc:0.943 tst_acc:0.911
epoch: 427 traing_acc:0.943 tst_acc:0.911
epoch: 428 traing_acc:0.943 tst_acc:0.911
epoch: 429 traing_acc:0.943 tst_acc:0.911
epoch: 430 traing_acc:0.952 tst_acc:0.911
epoch: 431 traing_acc:0.962 tst_acc:0.911
epoch: 432 traing_acc:0.952 tst_acc:0.911
epoch: 433 traing_acc:0.952 tst_acc:0.911
epoch: 434 traing_acc:0.943 tst_acc:0.911
epoch: 435 traing_acc:0.924 tst_acc:0.911
epoch: 436 traing_acc:0.943 tst_acc:0.911
epoch: 437 traing_acc:0.952 tst_acc:0.911
epoch: 438 traing_acc:0.943 tst_acc:0.911
epoch: 439 traing_acc:0.933 tst_acc:0.911
epoch: 440 traing_acc:0.952 tst_acc:0.911
epoch: 441 traing_acc:0.952 tst_acc:0.911
epoch: 442 traing_acc:0.943 tst_acc:0.911
epoch: 443 traing_acc:0.952 tst_acc:0.911
epoch: 444 traing_acc:0.952 tst_acc:0.911
epoch: 445 traing_acc:0.952 tst_acc:0.911
epoch: 446 traing_acc:0.933 tst_acc:0.911
epoch: 447 traing_acc:0.943 tst_acc:0.911
epoch: 448 traing_acc:0.943 tst_acc:0.911
epoch: 449 traing_acc:0.952 tst_acc:0.911
epoch: 450 traing_acc:0.952 tst_acc:0.911
epoch: 451 traing_acc:0.943 tst_acc:0.911
epoch: 452 traing_acc:0.933 tst_acc:0.911
epoch: 453 traing_acc:0.952 tst_acc:0.911
epoch: 454 traing_acc:0.952 tst_acc:0.911
epoch: 455 traing_acc:0.943 tst_acc:0.911
epoch: 456 traing_acc:0.952 tst_acc:0.911
epoch: 457 traing_acc:0.933 tst_acc:0.911
epoch: 458 traing_acc:0.952 tst_acc:0.911
epoch: 459 traing_acc:0.952 tst_acc:0.911
epoch: 460 traing_acc:0.943 tst_acc:0.911
epoch: 461 traing_acc:0.933 tst_acc:0.911
epoch: 462 traing_acc:0.952 tst_acc:0.911
epoch: 463 traing_acc:0.952 tst_acc:0.911
epoch: 464 traing_acc:0.952 tst_acc:0.911
epoch: 465 traing_acc:0.943 tst_acc:0.911
epoch: 466 traing_acc:0.943 tst_acc:0.911
epoch: 467 traing_acc:0.943 tst_acc:0.911
epoch: 468 traing_acc:0.952 tst_acc:0.911
epoch: 469 traing_acc:0.952 tst_acc:0.911
epoch: 470 traing_acc:0.943 tst_acc:0.911
epoch: 471 traing_acc:0.933 tst_acc:0.911
epoch: 472 traing_acc:0.943 tst_acc:0.911
epoch: 473 traing_acc:0.943 tst_acc:0.911
```

```
epoch: 474 traing_acc:0.933 tst_acc:0.911
epoch: 475 traing_acc:0.943 tst_acc:0.911
epoch: 476 traing_acc:0.943 tst_acc:0.911
epoch: 477 traing_acc:0.952 tst_acc:0.911
epoch: 478 traing_acc:0.962 tst_acc:0.911
epoch: 479 traing_acc:0.952 tst_acc:0.911
epoch: 480 traing_acc:0.943 tst_acc:0.911
epoch: 481 traing_acc:0.943 tst_acc:0.911
epoch: 482 traing_acc:0.943 tst_acc:0.911
epoch: 483 traing_acc:0.933 tst_acc:0.911
epoch: 484 traing_acc:0.952 tst_acc:0.911
epoch: 485 traing_acc:0.943 tst_acc:0.911
epoch: 486 traing_acc:0.952 tst_acc:0.911
epoch: 487 traing_acc:0.952 tst_acc:0.911
epoch: 488 traing_acc:0.962 tst_acc:0.911
epoch: 489 traing_acc:0.943 tst_acc:0.911
epoch: 490 traing_acc:0.952 tst_acc:0.911
epoch: 491 traing_acc:0.943 tst_acc:0.911
epoch: 492 traing_acc:0.924 tst_acc:0.911
epoch: 493 traing_acc:0.943 tst_acc:0.911
epoch: 494 traing_acc:0.943 tst_acc:0.911
epoch: 495 traing_acc:0.933 tst_acc:0.911
epoch: 496 traing_acc:0.943 tst_acc:0.911
epoch: 497 traing_acc:0.943 tst_acc:0.911
epoch: 498 traing_acc:0.943 tst_acc:0.911
epoch: 499 traing_acc:0.962 tst_acc:0.911
epoch: 500 traing_acc:0.952 tst_acc:0.911
epoch: 501 traing_acc:0.952 tst_acc:0.911
epoch: 502 traing_acc:0.943 tst_acc:0.911
epoch: 503 traing_acc:0.952 tst_acc:0.911
epoch: 504 traing_acc:0.933 tst_acc:0.911
epoch: 505 traing_acc:0.962 tst_acc:0.911
epoch: 506 traing_acc:0.943 tst_acc:0.911
epoch: 507 traing_acc:0.952 tst_acc:0.911
epoch: 508 traing_acc:0.943 tst_acc:0.911
epoch: 509 traing_acc:0.924 tst_acc:0.911
epoch: 510 traing_acc:0.952 tst_acc:0.911
epoch: 511 traing_acc:0.962 tst_acc:0.911
epoch: 512 traing_acc:0.943 tst_acc:0.911
epoch: 513 traing_acc:0.952 tst_acc:0.911
epoch: 514 traing_acc:0.943 tst_acc:0.911
epoch: 515 traing_acc:0.952 tst_acc:0.911
epoch: 516 traing_acc:0.943 tst_acc:0.911
epoch: 517 traing_acc:0.943 tst_acc:0.911
epoch: 518 traing_acc:0.933 tst_acc:0.911
epoch: 519 traing_acc:0.952 tst_acc:0.911
epoch: 520 traing_acc:0.952 tst_acc:0.911
epoch: 521 traing_acc:0.943 tst_acc:0.911
```

```
epoch: 522 traing_acc:0.952 tst_acc:0.911
epoch: 523 traing_acc:0.933 tst_acc:0.911
epoch: 524 traing_acc:0.962 tst_acc:0.911
epoch: 525 traing_acc:0.943 tst_acc:0.911
epoch: 526 traing_acc:0.952 tst_acc:0.911
epoch: 527 traing_acc:0.933 tst_acc:0.911
epoch: 528 traing_acc:0.933 tst_acc:0.911
epoch: 529 traing_acc:0.952 tst_acc:0.911
epoch: 530 traing_acc:0.952 tst_acc:0.911
epoch: 531 traing_acc:0.943 tst_acc:0.911
epoch: 532 traing_acc:0.952 tst_acc:0.911
epoch: 533 traing_acc:0.952 tst_acc:0.911
epoch: 534 traing_acc:0.952 tst_acc:0.911
epoch: 535 traing_acc:0.943 tst_acc:0.911
epoch: 536 traing_acc:0.933 tst_acc:0.911
epoch: 537 traing_acc:0.943 tst_acc:0.911
epoch: 538 traing_acc:0.943 tst_acc:0.911
epoch: 539 traing_acc:0.943 tst_acc:0.911
epoch: 540 traing_acc:0.933 tst_acc:0.911
epoch: 541 traing_acc:0.952 tst_acc:0.911
epoch: 542 traing_acc:0.943 tst_acc:0.911
epoch: 543 traing_acc:0.962 tst_acc:0.911
epoch: 544 traing_acc:0.952 tst_acc:0.911
epoch: 545 traing_acc:0.952 tst_acc:0.911
epoch: 546 traing_acc:0.943 tst_acc:0.911
epoch: 547 traing_acc:0.943 tst_acc:0.911
epoch: 548 traing_acc:0.943 tst_acc:0.911
epoch: 549 traing_acc:0.933 tst_acc:0.911
epoch: 550 traing_acc:0.943 tst_acc:0.911
epoch: 551 traing_acc:0.933 tst_acc:0.911
epoch: 552 traing_acc:0.962 tst_acc:0.911
epoch: 553 traing_acc:0.952 tst_acc:0.911
epoch: 554 traing_acc:0.952 tst_acc:0.911
epoch: 555 traing_acc:0.943 tst_acc:0.911
epoch: 556 traing_acc:0.943 tst_acc:0.911
epoch: 557 traing_acc:0.952 tst_acc:0.911
epoch: 558 traing_acc:0.933 tst_acc:0.911
epoch: 559 traing_acc:0.943 tst_acc:0.911
epoch: 560 traing_acc:0.952 tst_acc:0.911
epoch: 561 traing_acc:0.943 tst_acc:0.911
epoch: 562 traing_acc:0.962 tst_acc:0.911
epoch: 563 traing_acc:0.952 tst_acc:0.911
epoch: 564 traing_acc:0.952 tst_acc:0.911
epoch: 565 traing_acc:0.943 tst_acc:0.911
epoch: 566 traing_acc:0.933 tst_acc:0.911
epoch: 567 traing_acc:0.943 tst_acc:0.911
epoch: 568 traing_acc:0.943 tst_acc:0.911
epoch: 569 traing_acc:0.943 tst_acc:0.911
```

```
epoch: 570 traing_acc:0.943 tst_acc:0.911
epoch: 571 traing_acc:0.962 tst_acc:0.911
epoch: 572 traing_acc:0.952 tst_acc:0.911
epoch: 573 traing_acc:0.952 tst_acc:0.911
epoch: 574 traing_acc:0.943 tst_acc:0.911
epoch: 575 traing_acc:0.943 tst_acc:0.911
epoch: 576 traing_acc:0.943 tst_acc:0.911
epoch: 577 traing_acc:0.933 tst_acc:0.911
epoch: 578 traing_acc:0.943 tst_acc:0.911
epoch: 579 traing_acc:0.943 tst_acc:0.911
epoch: 580 traing_acc:0.952 tst_acc:0.911
epoch: 581 traing_acc:0.952 tst_acc:0.911
epoch: 582 traing_acc:0.943 tst_acc:0.911
epoch: 583 traing_acc:0.933 tst_acc:0.911
epoch: 584 traing_acc:0.952 tst_acc:0.911
epoch: 585 traing_acc:0.952 tst_acc:0.911
epoch: 586 traing_acc:0.943 tst_acc:0.911
epoch: 587 traing_acc:0.952 tst_acc:0.911
epoch: 588 traing_acc:0.933 tst_acc:0.911
epoch: 589 traing_acc:0.952 tst_acc:0.911
epoch: 590 traing_acc:0.952 tst_acc:0.911
epoch: 591 traing_acc:0.943 tst_acc:0.911
epoch: 592 traing_acc:0.933 tst_acc:0.911
epoch: 593 traing_acc:0.952 tst_acc:0.911
epoch: 594 traing_acc:0.952 tst_acc:0.911
epoch: 595 traing_acc:0.952 tst_acc:0.911
epoch: 596 traing_acc:0.943 tst_acc:0.911
epoch: 597 traing_acc:0.943 tst_acc:0.911
epoch: 598 traing_acc:0.943 tst_acc:0.911
epoch: 599 traing_acc:0.952 tst_acc:0.911
epoch: 600 traing_acc:0.952 tst_acc:0.911
epoch: 601 traing_acc:0.943 tst_acc:0.911
epoch: 602 traing_acc:0.933 tst_acc:0.911
epoch: 603 traing_acc:0.952 tst_acc:0.911
epoch: 604 traing_acc:0.952 tst_acc:0.911
epoch: 605 traing_acc:0.943 tst_acc:0.911
epoch: 606 traing_acc:0.952 tst_acc:0.911
epoch: 607 traing_acc:0.933 tst_acc:0.911
epoch: 608 traing_acc:0.962 tst_acc:0.911
epoch: 609 traing_acc:0.943 tst_acc:0.911
epoch: 610 traing_acc:0.952 tst_acc:0.911
epoch: 611 traing_acc:0.933 tst_acc:0.911
epoch: 612 traing_acc:0.943 tst_acc:0.911
epoch: 613 traing_acc:0.933 tst_acc:0.911
epoch: 614 traing_acc:0.943 tst_acc:0.911
epoch: 615 traing_acc:0.943 tst_acc:0.911
epoch: 616 traing_acc:0.943 tst_acc:0.911
epoch: 617 traing_acc:0.952 tst_acc:0.911
```

```
epoch: 618 traing_acc:0.952 tst_acc:0.911
epoch: 619 traing_acc:0.962 tst_acc:0.911
epoch: 620 traing_acc:0.943 tst_acc:0.911
epoch: 621 traing_acc:0.943 tst_acc:0.911
epoch: 622 traing_acc:0.943 tst_acc:0.911
epoch: 623 traing_acc:0.943 tst_acc:0.911
epoch: 624 traing_acc:0.933 tst_acc:0.911
epoch: 625 traing_acc:0.952 tst_acc:0.911
epoch: 626 traing_acc:0.933 tst_acc:0.911
epoch: 627 traing_acc:0.952 tst_acc:0.911
epoch: 628 traing_acc:0.962 tst_acc:0.911
epoch: 629 traing_acc:0.962 tst_acc:0.911
epoch: 630 traing_acc:0.962 tst_acc:0.911
epoch: 631 traing_acc:0.952 tst_acc:0.911
epoch: 632 traing_acc:0.943 tst_acc:0.911
epoch: 633 traing_acc:0.952 tst_acc:0.911
epoch: 634 traing_acc:0.952 tst_acc:0.911
epoch: 635 traing_acc:0.962 tst_acc:0.911
epoch: 636 traing_acc:0.943 tst_acc:0.911
epoch: 637 traing_acc:0.952 tst_acc:0.911
epoch: 638 traing_acc:0.943 tst_acc:0.911
epoch: 639 traing_acc:0.952 tst_acc:0.911
epoch: 640 traing_acc:0.952 tst_acc:0.911
epoch: 641 traing_acc:0.943 tst_acc:0.911
epoch: 642 traing_acc:0.943 tst_acc:0.911
epoch: 643 traing_acc:0.943 tst_acc:0.911
epoch: 644 traing_acc:0.952 tst_acc:0.911
epoch: 645 traing_acc:0.962 tst_acc:0.911
epoch: 646 traing_acc:0.962 tst_acc:0.911
epoch: 647 traing_acc:0.962 tst_acc:0.911
epoch: 648 traing_acc:0.952 tst_acc:0.911
epoch: 649 traing_acc:0.952 tst_acc:0.911
epoch: 650 traing_acc:0.962 tst_acc:0.911
epoch: 651 traing_acc:0.952 tst_acc:0.911
epoch: 652 traing_acc:0.952 tst_acc:0.911
epoch: 653 traing_acc:0.943 tst_acc:0.911
epoch: 654 traing_acc:0.952 tst_acc:0.911
epoch: 655 traing_acc:0.943 tst_acc:0.911
epoch: 656 traing_acc:0.943 tst_acc:0.911
epoch: 657 traing_acc:0.952 tst_acc:0.911
epoch: 658 traing_acc:0.962 tst_acc:0.911
epoch: 659 traing_acc:0.952 tst_acc:0.911
epoch: 660 traing_acc:0.962 tst_acc:0.911
epoch: 661 traing_acc:0.962 tst_acc:0.911
epoch: 662 traing_acc:0.971 tst_acc:0.911
epoch: 663 traing_acc:0.971 tst_acc:0.911
epoch: 664 traing_acc:0.971 tst_acc:0.911
epoch: 665 traing_acc:0.962 tst_acc:0.911
```

```
epoch: 666 traing_acc:0.962 tst_acc:0.911
epoch: 667 traing_acc:0.952 tst_acc:0.911
epoch: 668 traing_acc:0.952 tst_acc:0.911
epoch: 669 traing_acc:0.952 tst_acc:0.911
epoch: 670 traing_acc:0.943 tst_acc:0.911
epoch: 671 traing_acc:0.943 tst_acc:0.911
epoch: 672 traing_acc:0.952 tst_acc:0.911
epoch: 673 traing_acc:0.952 tst_acc:0.911
epoch: 674 traing_acc:0.962 tst_acc:0.911
epoch: 675 traing_acc:0.952 tst_acc:0.911
epoch: 676 traing_acc:0.962 tst_acc:0.911
epoch: 677 traing_acc:0.962 tst_acc:0.911
epoch: 678 traing_acc:0.952 tst_acc:0.911
epoch: 679 traing_acc:0.962 tst_acc:0.911
epoch: 680 traing_acc:0.952 tst_acc:0.911
epoch: 681 traing_acc:0.971 tst_acc:0.911
epoch: 682 traing_acc:0.971 tst_acc:0.911
epoch: 683 traing_acc:0.971 tst_acc:0.911
epoch: 684 traing_acc:0.962 tst_acc:0.911
epoch: 685 traing_acc:0.962 tst_acc:0.911
epoch: 686 traing_acc:0.952 tst_acc:0.911
epoch: 687 traing_acc:0.962 tst_acc:0.911
epoch: 688 traing_acc:0.952 tst_acc:0.911
epoch: 689 traing_acc:0.952 tst_acc:0.911
epoch: 690 traing_acc:0.962 tst_acc:0.911
epoch: 691 traing_acc:0.962 tst_acc:0.911
epoch: 692 traing_acc:0.971 tst_acc:0.911
epoch: 693 traing_acc:0.971 tst_acc:0.911
epoch: 694 traing_acc:0.962 tst_acc:0.911
epoch: 695 traing_acc:0.971 tst_acc:0.911
epoch: 696 traing_acc:0.962 tst_acc:0.911
epoch: 697 traing_acc:0.962 tst_acc:0.911
epoch: 698 traing_acc:0.962 tst_acc:0.911
epoch: 699 traing_acc:0.952 tst_acc:0.911
epoch: 700 traing_acc:0.962 tst_acc:0.911
epoch: 701 traing_acc:0.971 tst_acc:0.911
epoch: 702 traing_acc:0.952 tst_acc:0.911
epoch: 703 traing_acc:0.971 tst_acc:0.911
epoch: 704 traing_acc:0.971 tst_acc:0.911
epoch: 705 traing_acc:0.962 tst_acc:0.911
epoch: 706 traing_acc:0.971 tst_acc:0.911
epoch: 707 traing_acc:0.971 tst_acc:0.911
epoch: 708 traing_acc:0.962 tst_acc:0.911
epoch: 709 traing_acc:0.971 tst_acc:0.911
epoch: 710 traing_acc:0.971 tst_acc:0.911
epoch: 711 traing_acc:0.952 tst_acc:0.911
epoch: 712 traing_acc:0.962 tst_acc:0.911
epoch: 713 traing_acc:0.971 tst_acc:0.911
```

```
epoch: 714 traing_acc:0.952 tst_acc:0.911
epoch: 715 traing_acc:0.971 tst_acc:0.911
epoch: 716 traing_acc:0.971 tst_acc:0.911
epoch: 717 traing_acc:0.962 tst_acc:0.911
epoch: 718 traing_acc:0.971 tst_acc:0.911
epoch: 719 traing_acc:0.971 tst_acc:0.911
epoch: 720 traing_acc:0.962 tst_acc:0.911
epoch: 721 traing_acc:0.971 tst_acc:0.911
epoch: 722 traing_acc:0.971 tst_acc:0.911
epoch: 723 traing_acc:0.962 tst_acc:0.911
epoch: 724 traing_acc:0.971 tst_acc:0.911
epoch: 725 traing_acc:0.971 tst_acc:0.911
epoch: 726 traing_acc:0.971 tst_acc:0.911
epoch: 727 traing_acc:0.971 tst_acc:0.911
epoch: 728 traing_acc:0.962 tst_acc:0.911
epoch: 729 traing_acc:0.971 tst_acc:0.911
epoch: 730 traing_acc:0.971 tst_acc:0.911
epoch: 731 traing_acc:0.952 tst_acc:0.911
epoch: 732 traing_acc:0.971 tst_acc:0.911
epoch: 733 traing_acc:0.962 tst_acc:0.911
epoch: 734 traing_acc:0.971 tst_acc:0.911
epoch: 735 traing_acc:0.971 tst_acc:0.911
epoch: 736 traing_acc:0.971 tst_acc:0.911
epoch: 737 traing_acc:0.962 tst_acc:0.911
epoch: 738 traing_acc:0.981 tst_acc:0.911
epoch: 739 traing_acc:0.971 tst_acc:0.911
epoch: 740 traing_acc:0.952 tst_acc:0.911
epoch: 741 traing_acc:0.981 tst_acc:0.911
epoch: 742 traing_acc:0.971 tst_acc:0.911
epoch: 743 traing_acc:0.962 tst_acc:0.911
epoch: 744 traing_acc:0.962 tst_acc:0.911
epoch: 745 traing_acc:0.971 tst_acc:0.911
epoch: 746 traing_acc:0.981 tst_acc:0.911
epoch: 747 traing_acc:0.962 tst_acc:0.911
epoch: 748 traing_acc:0.962 tst_acc:0.911
epoch: 749 traing_acc:0.971 tst_acc:0.911
epoch: 750 traing_acc:0.971 tst_acc:0.911
epoch: 751 traing_acc:0.981 tst_acc:0.911
epoch: 752 traing_acc:0.962 tst_acc:0.911
epoch: 753 traing_acc:0.971 tst_acc:0.911
epoch: 754 traing_acc:0.971 tst_acc:0.911
epoch: 755 traing_acc:0.971 tst_acc:0.911
epoch: 756 traing_acc:0.962 tst_acc:0.911
epoch: 757 traing_acc:0.981 tst_acc:0.911
epoch: 758 traing_acc:0.971 tst_acc:0.911
epoch: 759 traing_acc:0.962 tst_acc:0.911
epoch: 760 traing_acc:0.962 tst_acc:0.911
epoch: 761 traing_acc:0.981 tst_acc:0.911
```

```
epoch: 762 traing_acc:0.971 tst_acc:0.911
epoch: 763 traing_acc:0.952 tst_acc:0.911
epoch: 764 traing_acc:0.981 tst_acc:0.911
epoch: 765 traing_acc:0.971 tst_acc:0.911
epoch: 766 traing_acc:0.962 tst_acc:0.911
epoch: 767 traing_acc:0.962 tst_acc:0.911
epoch: 768 traing_acc:0.981 tst_acc:0.911
epoch: 769 traing_acc:0.971 tst_acc:0.911
epoch: 770 traing_acc:0.962 tst_acc:0.911
epoch: 771 traing_acc:0.962 tst_acc:0.911
epoch: 772 traing_acc:0.971 tst_acc:0.911
epoch: 773 traing_acc:0.981 tst_acc:0.911
epoch: 774 traing_acc:0.971 tst_acc:0.911
epoch: 775 traing_acc:0.962 tst_acc:0.911
epoch: 776 traing_acc:0.971 tst_acc:0.911
epoch: 777 traing_acc:0.971 tst_acc:0.911
epoch: 778 traing_acc:0.962 tst_acc:0.911
epoch: 779 traing_acc:0.981 tst_acc:0.911
epoch: 780 traing_acc:0.971 tst_acc:0.911
epoch: 781 traing_acc:0.971 tst_acc:0.933
epoch: 782 traing_acc:0.962 tst_acc:0.933
epoch: 783 traing_acc:0.962 tst_acc:0.933
epoch: 784 traing_acc:0.981 tst_acc:0.933
epoch: 785 traing_acc:0.981 tst_acc:0.933
epoch: 786 traing_acc:0.952 tst_acc:0.933
epoch: 787 traing_acc:0.981 tst_acc:0.933
epoch: 788 traing_acc:0.981 tst_acc:0.933
epoch: 789 traing_acc:0.971 tst_acc:0.933
epoch: 790 traing_acc:0.962 tst_acc:0.933
epoch: 791 traing_acc:0.981 tst_acc:0.933
epoch: 792 traing_acc:0.981 tst_acc:0.933
epoch: 793 traing_acc:0.962 tst_acc:0.933
epoch: 794 traing_acc:0.971 tst_acc:0.933
epoch: 795 traing_acc:0.971 tst_acc:0.933
epoch: 796 traing_acc:0.981 tst_acc:0.933
epoch: 797 traing_acc:0.971 tst_acc:0.933
epoch: 798 traing_acc:0.971 tst_acc:0.933
epoch: 799 traing_acc:0.971 tst_acc:0.933
epoch: 800 traing_acc:0.981 tst_acc:0.933
epoch: 801 traing_acc:0.981 tst_acc:0.933
epoch: 802 traing_acc:0.981 tst_acc:0.933
epoch: 803 traing_acc:0.962 tst_acc:0.933
epoch: 804 traing_acc:0.981 tst_acc:0.933
epoch: 805 traing_acc:0.971 tst_acc:0.933
epoch: 806 traing_acc:0.971 tst_acc:0.933
epoch: 807 traing_acc:0.981 tst_acc:0.933
epoch: 808 traing_acc:0.971 tst_acc:0.933
epoch: 809 traing_acc:0.962 tst_acc:0.933
```

```
epoch: 810 traing_acc:0.981 tst_acc:0.933
epoch: 811 traing_acc:0.981 tst_acc:0.933
epoch: 812 traing_acc:0.962 tst_acc:0.933
epoch: 813 traing_acc:0.971 tst_acc:0.933
epoch: 814 traing_acc:0.981 tst_acc:0.933
epoch: 815 traing_acc:0.990 tst_acc:0.933
epoch: 816 traing_acc:0.981 tst_acc:0.933
epoch: 817 traing_acc:0.971 tst_acc:0.933
epoch: 818 traing_acc:0.971 tst_acc:0.933
epoch: 819 traing_acc:0.981 tst_acc:0.933
epoch: 820 traing_acc:0.981 tst_acc:0.933
epoch: 821 traing_acc:0.981 tst_acc:0.933
epoch: 822 traing_acc:0.971 tst_acc:0.933
epoch: 823 traing_acc:0.981 tst_acc:0.933
epoch: 824 traing_acc:0.981 tst_acc:0.933
epoch: 825 traing_acc:0.981 tst_acc:0.933
epoch: 826 traing_acc:0.971 tst_acc:0.933
epoch: 827 traing_acc:0.971 tst_acc:0.933
epoch: 828 traing_acc:0.981 tst_acc:0.933
epoch: 829 traing_acc:0.981 tst_acc:0.933
epoch: 830 traing_acc:0.981 tst_acc:0.933
epoch: 831 traing_acc:0.981 tst_acc:0.933
epoch: 832 traing_acc:0.981 tst_acc:0.933
epoch: 833 traing_acc:0.962 tst_acc:0.933
epoch: 834 traing_acc:0.971 tst_acc:0.933
epoch: 835 traing_acc:0.981 tst_acc:0.933
epoch: 836 traing_acc:0.971 tst_acc:0.933
epoch: 837 traing_acc:0.962 tst_acc:0.933
epoch: 838 traing_acc:0.981 tst_acc:0.933
epoch: 839 traing_acc:0.971 tst_acc:0.933
epoch: 840 traing_acc:0.971 tst_acc:0.933
epoch: 841 traing_acc:0.981 tst_acc:0.933
epoch: 842 traing_acc:0.971 tst_acc:0.933
epoch: 843 traing_acc:0.981 tst_acc:0.933
epoch: 844 traing_acc:0.990 tst_acc:0.933
epoch: 845 traing_acc:0.981 tst_acc:0.933
epoch: 846 traing_acc:0.981 tst_acc:0.933
epoch: 847 traing_acc:0.990 tst_acc:0.933
epoch: 848 traing_acc:0.981 tst_acc:0.933
epoch: 849 traing_acc:0.981 tst_acc:0.933
epoch: 850 traing_acc:0.981 tst_acc:0.933
epoch: 851 traing_acc:0.981 tst_acc:0.933
epoch: 852 traing_acc:0.990 tst_acc:0.933
epoch: 853 traing_acc:0.990 tst_acc:0.933
epoch: 854 traing_acc:0.981 tst_acc:0.933
epoch: 855 traing_acc:0.981 tst_acc:0.933
epoch: 856 traing_acc:0.990 tst_acc:0.933
epoch: 857 traing_acc:0.981 tst_acc:0.933
```

```
epoch: 858 traing_acc:0.981 tst_acc:0.933
epoch: 859 traing_acc:0.971 tst_acc:0.933
epoch: 860 traing_acc:0.990 tst_acc:0.933
epoch: 861 traing_acc:0.981 tst_acc:0.933
epoch: 862 traing_acc:0.981 tst_acc:0.933
epoch: 863 traing_acc:0.971 tst_acc:0.933
epoch: 864 traing_acc:0.971 tst_acc:0.933
epoch: 865 traing_acc:0.981 tst_acc:0.933
epoch: 866 traing_acc:0.981 tst_acc:0.933
epoch: 867 traing_acc:0.971 tst_acc:0.933
epoch: 868 traing_acc:0.981 tst_acc:0.933
epoch: 869 traing_acc:0.981 tst_acc:0.933
epoch: 870 traing_acc:0.971 tst_acc:0.933
epoch: 871 traing_acc:0.981 tst_acc:0.933
epoch: 872 traing_acc:0.971 tst_acc:0.933
epoch: 873 traing_acc:0.971 tst_acc:0.933
epoch: 874 traing_acc:0.971 tst_acc:0.933
epoch: 875 traing_acc:0.962 tst_acc:0.933
epoch: 876 traing_acc:0.981 tst_acc:0.933
epoch: 877 traing_acc:0.981 tst_acc:0.933
epoch: 878 traing_acc:0.990 tst_acc:0.933
epoch: 879 traing_acc:0.981 tst_acc:0.933
epoch: 880 traing_acc:0.981 tst_acc:0.933
epoch: 881 traing_acc:0.990 tst_acc:0.933
epoch: 882 traing_acc:0.990 tst_acc:0.933
epoch: 883 traing_acc:0.981 tst_acc:0.933
epoch: 884 traing_acc:0.990 tst_acc:0.933
epoch: 885 traing_acc:0.990 tst_acc:0.933
epoch: 886 traing_acc:0.971 tst_acc:0.933
epoch: 887 traing_acc:0.981 tst_acc:0.933
epoch: 888 traing_acc:0.981 tst_acc:0.933
epoch: 889 traing_acc:0.990 tst_acc:0.933
epoch: 890 traing_acc:0.990 tst_acc:0.933
epoch: 891 traing_acc:0.981 tst_acc:0.933
epoch: 892 traing_acc:0.981 tst_acc:0.933
epoch: 893 traing_acc:0.981 tst_acc:0.933
epoch: 894 traing_acc:0.990 tst_acc:0.933
epoch: 895 traing_acc:0.981 tst_acc:0.933
epoch: 896 traing_acc:0.990 tst_acc:0.933
epoch: 897 traing_acc:0.981 tst_acc:0.933
epoch: 898 traing_acc:0.981 tst_acc:0.933
epoch: 899 traing_acc:0.981 tst_acc:0.933
epoch: 900 traing_acc:0.981 tst_acc:0.933
epoch: 901 traing_acc:0.990 tst_acc:0.933
epoch: 902 traing_acc:0.990 tst_acc:0.933
epoch: 903 traing_acc:0.981 tst_acc:0.933
epoch: 904 traing_acc:0.981 tst_acc:0.933
epoch: 905 traing_acc:0.981 tst_acc:0.933
```

```
epoch: 906 traing_acc:0.971 tst_acc:0.933
epoch: 907 traing_acc:0.981 tst_acc:0.933
epoch: 908 traing_acc:0.971 tst_acc:0.933
epoch: 909 traing_acc:0.981 tst_acc:0.933
epoch: 910 traing_acc:0.981 tst_acc:0.933
epoch: 911 traing_acc:0.981 tst_acc:0.933
epoch: 912 traing_acc:0.962 tst_acc:0.933
epoch: 913 traing_acc:0.981 tst_acc:0.933
epoch: 914 traing_acc:0.971 tst_acc:0.933
epoch: 915 traing_acc:0.971 tst_acc:0.933
epoch: 916 traing_acc:0.990 tst_acc:0.933
epoch: 917 traing_acc:0.981 tst_acc:0.933
epoch: 918 traing_acc:0.971 tst_acc:0.933
epoch: 919 traing_acc:0.981 tst_acc:0.933
epoch: 920 traing_acc:0.990 tst_acc:0.933
epoch: 921 traing_acc:0.990 tst_acc:0.933
epoch: 922 traing_acc:0.990 tst_acc:0.933
epoch: 923 traing_acc:0.981 tst_acc:0.933
epoch: 924 traing_acc:0.981 tst_acc:0.933
epoch: 925 traing_acc:0.981 tst_acc:0.933
epoch: 926 traing_acc:0.981 tst_acc:0.933
epoch: 927 traing_acc:0.990 tst_acc:0.933
epoch: 928 traing_acc:0.990 tst_acc:0.933
epoch: 929 traing_acc:0.981 tst_acc:0.933
epoch: 930 traing_acc:0.981 tst_acc:0.933
epoch: 931 traing_acc:0.981 tst_acc:0.933
epoch: 932 traing_acc:0.981 tst_acc:0.933
epoch: 933 traing_acc:0.990 tst_acc:0.933
epoch: 934 traing_acc:0.990 tst_acc:0.933
epoch: 935 traing_acc:0.981 tst_acc:0.933
epoch: 936 traing_acc:0.981 tst_acc:0.933
epoch: 937 traing_acc:0.981 tst_acc:0.933
epoch: 938 traing_acc:0.990 tst_acc:0.933
epoch: 939 traing_acc:0.990 tst_acc:0.933
epoch: 940 traing_acc:0.971 tst_acc:0.933
epoch: 941 traing_acc:0.981 tst_acc:0.933
epoch: 942 traing_acc:0.981 tst_acc:0.933
epoch: 943 traing_acc:0.981 tst_acc:0.933
epoch: 944 traing_acc:0.990 tst_acc:0.933
epoch: 945 traing_acc:0.990 tst_acc:0.933
epoch: 946 traing_acc:0.971 tst_acc:0.933
epoch: 947 traing_acc:0.981 tst_acc:0.933
epoch: 948 traing_acc:0.971 tst_acc:0.933
epoch: 949 traing_acc:0.981 tst_acc:0.933
epoch: 950 traing_acc:0.981 tst_acc:0.933
epoch: 951 traing_acc:0.981 tst_acc:0.933
epoch: 952 traing_acc:0.981 tst_acc:0.933
epoch: 953 traing_acc:0.971 tst_acc:0.933
```

```
epoch: 954 traing_acc:0.981 tst_acc:0.933
epoch: 955 traing_acc:0.971 tst_acc:0.933
epoch: 956 traing_acc:0.981 tst_acc:0.933
epoch: 957 traing_acc:0.990 tst_acc:0.933
epoch: 958 traing_acc:0.981 tst_acc:0.933
epoch: 959 traing_acc:0.981 tst_acc:0.933
epoch: 960 traing_acc:0.990 tst_acc:0.933
epoch: 961 traing_acc:0.990 tst_acc:0.933
epoch: 962 traing_acc:0.990 tst_acc:0.933
epoch: 963 traing_acc:0.981 tst_acc:0.933
epoch: 964 traing_acc:0.981 tst_acc:0.933
epoch: 965 traing_acc:0.981 tst_acc:0.933
epoch: 966 traing_acc:0.981 tst_acc:0.933
epoch: 967 traing_acc:0.990 tst_acc:0.933
epoch: 968 traing_acc:0.981 tst_acc:0.933
epoch: 969 traing_acc:0.981 tst_acc:0.933
epoch: 970 traing_acc:0.981 tst_acc:0.933
epoch: 971 traing_acc:0.990 tst_acc:0.933
epoch: 972 traing_acc:0.990 tst_acc:0.933
epoch: 973 traing_acc:0.990 tst_acc:0.933
epoch: 974 traing_acc:0.981 tst_acc:0.933
epoch: 975 traing_acc:0.981 tst_acc:0.933
epoch: 976 traing_acc:0.981 tst_acc:0.933
epoch: 977 traing_acc:0.981 tst_acc:0.933
epoch: 978 traing_acc:0.990 tst_acc:0.933
epoch: 979 traing_acc:0.990 tst_acc:0.933
epoch: 980 traing_acc:0.981 tst_acc:0.933
epoch: 981 traing_acc:0.971 tst_acc:0.933
epoch: 982 traing_acc:0.981 tst_acc:0.933
epoch: 983 traing_acc:0.990 tst_acc:0.933
epoch: 984 traing_acc:0.990 tst_acc:0.933
epoch: 985 traing_acc:0.990 tst_acc:0.933
epoch: 986 traing_acc:0.981 tst_acc:0.933
epoch: 987 traing_acc:0.981 tst_acc:0.933
epoch: 988 traing_acc:0.990 tst_acc:0.933
epoch: 989 traing_acc:0.981 tst_acc:0.933
epoch: 990 traing_acc:0.981 tst_acc:0.933
epoch: 991 traing_acc:0.990 tst_acc:0.933
epoch: 992 traing_acc:0.981 tst_acc:0.933
epoch: 993 traing_acc:0.981 tst_acc:0.933
epoch: 994 traing_acc:0.981 tst_acc:0.933
epoch: 995 traing_acc:0.981 tst_acc:0.933
epoch: 996 traing_acc:0.981 tst_acc:0.933
epoch: 997 traing_acc:0.981 tst_acc:0.933
epoch: 998 traing_acc:0.981 tst_acc:0.933
epoch: 999 traing_acc:0.990 tst_acc:0.933
epoch: 1000 traing_acc:0.981 tst_acc:0.933
epoch: 1001 traing_acc:0.990 tst_acc:0.933
```

```
epoch: 1002 traing_acc:0.981 tst_acc:0.933
epoch: 1003 traing_acc:0.981 tst_acc:0.933
epoch: 1004 traing_acc:0.990 tst_acc:0.933
epoch: 1005 traing_acc:1.000 tst_acc:0.933
epoch: 1006 traing_acc:1.000 tst_acc:0.933
epoch: 1007 traing_acc:1.000 tst_acc:0.933
epoch: 1008 traing_acc:1.000 tst_acc:0.933
epoch: 1009 traing_acc:1.000 tst_acc:0.933
epoch: 1010 traing_acc:1.000 tst_acc:0.933
epoch: 1011 traing_acc:1.000 tst_acc:0.933
epoch: 1012 traing_acc:1.000 tst_acc:0.933
epoch: 1013 traing_acc:1.000 tst_acc:0.933
epoch: 1014 traing_acc:1.000 tst_acc:0.933
epoch: 1015 traing_acc:1.000 tst_acc:0.933
epoch: 1016 traing_acc:1.000 tst_acc:0.933
epoch: 1017 traing_acc:1.000 tst_acc:0.933
epoch: 1018 traing_acc:1.000 tst_acc:0.933
epoch: 1019 traing_acc:1.000 tst_acc:0.933
epoch: 1020 traing_acc:1.000 tst_acc:0.933
epoch: 1021 traing_acc:1.000 tst_acc:0.933
epoch: 1022 traing_acc:1.000 tst_acc:0.933
epoch: 1023 traing_acc:1.000 tst_acc:0.933
epoch: 1024 traing_acc:1.000 tst_acc:0.933
epoch: 1025 traing_acc:1.000 tst_acc:0.933
epoch: 1026 traing_acc:1.000 tst_acc:0.933
epoch: 1027 traing_acc:1.000 tst_acc:0.933
epoch: 1028 traing_acc:1.000 tst_acc:0.933
epoch: 1029 traing_acc:1.000 tst_acc:0.933
epoch: 1030 traing_acc:1.000 tst_acc:0.933
epoch: 1031 traing_acc:1.000 tst_acc:0.933
epoch: 1032 traing_acc:1.000 tst_acc:0.933
epoch: 1033 traing_acc:1.000 tst_acc:0.933
epoch: 1034 traing_acc:1.000 tst_acc:0.933
epoch: 1035 traing_acc:1.000 tst_acc:0.933
epoch: 1036 traing_acc:1.000 tst_acc:0.933
epoch: 1037 traing_acc:1.000 tst_acc:0.933
epoch: 1038 traing_acc:1.000 tst_acc:0.933
epoch: 1039 traing_acc:1.000 tst_acc:0.933
epoch: 1040 traing_acc:1.000 tst_acc:0.933
epoch: 1041 traing_acc:1.000 tst_acc:0.933
epoch: 1042 traing_acc:1.000 tst_acc:0.933
epoch: 1043 traing_acc:1.000 tst_acc:0.933
epoch: 1044 traing_acc:1.000 tst_acc:0.933
epoch: 1045 traing_acc:1.000 tst_acc:0.933
epoch: 1046 traing_acc:1.000 tst_acc:0.933
epoch: 1047 traing_acc:1.000 tst_acc:0.933
epoch: 1048 traing_acc:1.000 tst_acc:0.933
epoch: 1049 traing_acc:1.000 tst_acc:0.933
```

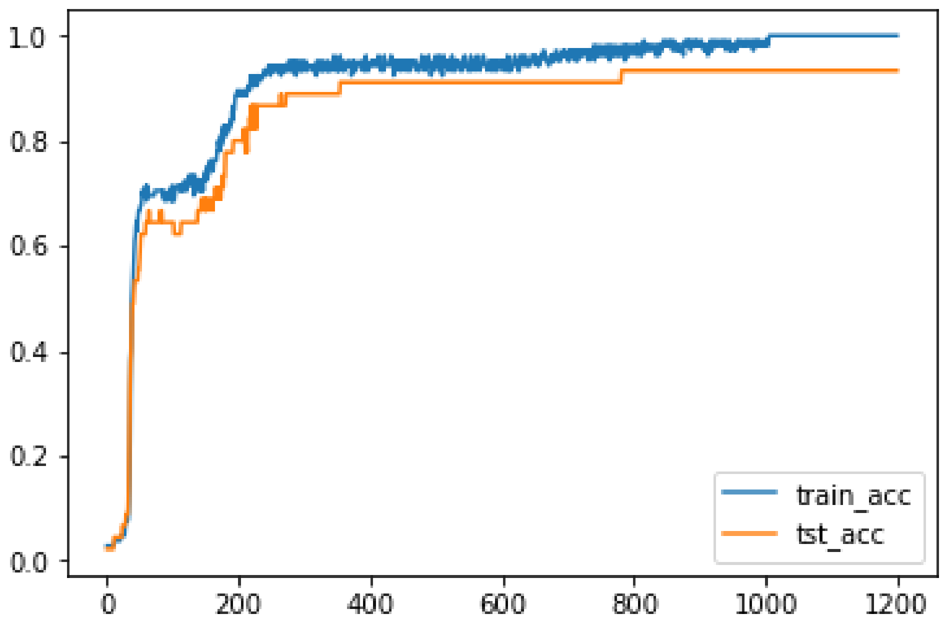
```
epoch: 1050 traing_acc:1.000 tst_acc:0.933
epoch: 1051 traing_acc:1.000 tst_acc:0.933
epoch: 1052 traing_acc:1.000 tst_acc:0.933
epoch: 1053 traing_acc:1.000 tst_acc:0.933
epoch: 1054 traing_acc:1.000 tst_acc:0.933
epoch: 1055 traing_acc:1.000 tst_acc:0.933
epoch: 1056 traing_acc:1.000 tst_acc:0.933
epoch: 1057 traing_acc:1.000 tst_acc:0.933
epoch: 1058 traing_acc:1.000 tst_acc:0.933
epoch: 1059 traing_acc:1.000 tst_acc:0.933
epoch: 1060 traing_acc:1.000 tst_acc:0.933
epoch: 1061 traing_acc:1.000 tst_acc:0.933
epoch: 1062 traing_acc:1.000 tst_acc:0.933
epoch: 1063 traing_acc:1.000 tst_acc:0.933
epoch: 1064 traing_acc:1.000 tst_acc:0.933
epoch: 1065 traing_acc:1.000 tst_acc:0.933
epoch: 1066 traing_acc:1.000 tst_acc:0.933
epoch: 1067 traing_acc:1.000 tst_acc:0.933
epoch: 1068 traing_acc:1.000 tst_acc:0.933
epoch: 1069 traing_acc:1.000 tst_acc:0.933
epoch: 1070 traing_acc:1.000 tst_acc:0.933
epoch: 1071 traing_acc:1.000 tst_acc:0.933
epoch: 1072 traing_acc:1.000 tst_acc:0.933
epoch: 1073 traing_acc:1.000 tst_acc:0.933
epoch: 1074 traing_acc:1.000 tst_acc:0.933
epoch: 1075 traing_acc:1.000 tst_acc:0.933
epoch: 1076 traing_acc:1.000 tst_acc:0.933
epoch: 1077 traing_acc:1.000 tst_acc:0.933
epoch: 1078 traing_acc:1.000 tst_acc:0.933
epoch: 1079 traing_acc:1.000 tst_acc:0.933
epoch: 1080 traing_acc:1.000 tst_acc:0.933
epoch: 1081 traing_acc:1.000 tst_acc:0.933
epoch: 1082 traing_acc:1.000 tst_acc:0.933
epoch: 1083 traing_acc:1.000 tst_acc:0.933
epoch: 1084 traing_acc:1.000 tst_acc:0.933
epoch: 1085 traing_acc:1.000 tst_acc:0.933
epoch: 1086 traing_acc:1.000 tst_acc:0.933
epoch: 1087 traing_acc:1.000 tst_acc:0.933
epoch: 1088 traing_acc:1.000 tst_acc:0.933
epoch: 1089 traing_acc:1.000 tst_acc:0.933
epoch: 1090 traing_acc:1.000 tst_acc:0.933
epoch: 1091 traing_acc:1.000 tst_acc:0.933
epoch: 1092 traing_acc:1.000 tst_acc:0.933
epoch: 1093 traing_acc:1.000 tst_acc:0.933
epoch: 1094 traing_acc:1.000 tst_acc:0.933
epoch: 1095 traing_acc:1.000 tst_acc:0.933
epoch: 1096 traing_acc:1.000 tst_acc:0.933
epoch: 1097 traing_acc:1.000 tst_acc:0.933
```

```
epoch: 1098 traing_acc:1.000 tst_acc:0.933
epoch: 1099 traing_acc:1.000 tst_acc:0.933
epoch: 1100 traing_acc:1.000 tst_acc:0.933
epoch: 1101 traing_acc:1.000 tst_acc:0.933
epoch: 1102 traing_acc:1.000 tst_acc:0.933
epoch: 1103 traing_acc:1.000 tst_acc:0.933
epoch: 1104 traing_acc:1.000 tst_acc:0.933
epoch: 1105 traing_acc:1.000 tst_acc:0.933
epoch: 1106 traing_acc:1.000 tst_acc:0.933
epoch: 1107 traing_acc:1.000 tst_acc:0.933
epoch: 1108 traing_acc:1.000 tst_acc:0.933
epoch: 1109 traing_acc:1.000 tst_acc:0.933
epoch: 1110 traing_acc:1.000 tst_acc:0.933
epoch: 1111 traing_acc:1.000 tst_acc:0.933
epoch: 1112 traing_acc:1.000 tst_acc:0.933
epoch: 1113 traing_acc:1.000 tst_acc:0.933
epoch: 1114 traing_acc:1.000 tst_acc:0.933
epoch: 1115 traing_acc:1.000 tst_acc:0.933
epoch: 1116 traing_acc:1.000 tst_acc:0.933
epoch: 1117 traing_acc:1.000 tst_acc:0.933
epoch: 1118 traing_acc:1.000 tst_acc:0.933
epoch: 1119 traing_acc:1.000 tst_acc:0.933
epoch: 1120 traing_acc:1.000 tst_acc:0.933
epoch: 1121 traing_acc:1.000 tst_acc:0.933
epoch: 1122 traing_acc:1.000 tst_acc:0.933
epoch: 1123 traing_acc:1.000 tst_acc:0.933
epoch: 1124 traing_acc:1.000 tst_acc:0.933
epoch: 1125 traing_acc:1.000 tst_acc:0.933
epoch: 1126 traing_acc:1.000 tst_acc:0.933
epoch: 1127 traing_acc:1.000 tst_acc:0.933
epoch: 1128 traing_acc:1.000 tst_acc:0.933
epoch: 1129 traing_acc:1.000 tst_acc:0.933
epoch: 1130 traing_acc:1.000 tst_acc:0.933
epoch: 1131 traing_acc:1.000 tst_acc:0.933
epoch: 1132 traing_acc:1.000 tst_acc:0.933
epoch: 1133 traing_acc:1.000 tst_acc:0.933
epoch: 1134 traing_acc:1.000 tst_acc:0.933
epoch: 1135 traing_acc:1.000 tst_acc:0.933
epoch: 1136 traing_acc:1.000 tst_acc:0.933
epoch: 1137 traing_acc:1.000 tst_acc:0.933
epoch: 1138 traing_acc:1.000 tst_acc:0.933
epoch: 1139 traing_acc:1.000 tst_acc:0.933
epoch: 1140 traing_acc:1.000 tst_acc:0.933
epoch: 1141 traing_acc:1.000 tst_acc:0.933
epoch: 1142 traing_acc:1.000 tst_acc:0.933
epoch: 1143 traing_acc:1.000 tst_acc:0.933
epoch: 1144 traing_acc:1.000 tst_acc:0.933
epoch: 1145 traing_acc:1.000 tst_acc:0.933
```

```
epoch: 1146 traing_acc:1.000 tst_acc:0.933
epoch: 1147 traing_acc:1.000 tst_acc:0.933
epoch: 1148 traing_acc:1.000 tst_acc:0.933
epoch: 1149 traing_acc:1.000 tst_acc:0.933
epoch: 1150 traing_acc:1.000 tst_acc:0.933
epoch: 1151 traing_acc:1.000 tst_acc:0.933
epoch: 1152 traing_acc:1.000 tst_acc:0.933
epoch: 1153 traing_acc:1.000 tst_acc:0.933
epoch: 1154 traing_acc:1.000 tst_acc:0.933
epoch: 1155 traing_acc:1.000 tst_acc:0.933
epoch: 1156 traing_acc:1.000 tst_acc:0.933
epoch: 1157 traing_acc:1.000 tst_acc:0.933
epoch: 1158 traing_acc:1.000 tst_acc:0.933
epoch: 1159 traing_acc:1.000 tst_acc:0.933
epoch: 1160 traing_acc:1.000 tst_acc:0.933
epoch: 1161 traing_acc:1.000 tst_acc:0.933
epoch: 1162 traing_acc:1.000 tst_acc:0.933
epoch: 1163 traing_acc:1.000 tst_acc:0.933
epoch: 1164 traing_acc:1.000 tst_acc:0.933
epoch: 1165 traing_acc:1.000 tst_acc:0.933
epoch: 1166 traing_acc:1.000 tst_acc:0.933
epoch: 1167 traing_acc:1.000 tst_acc:0.933
epoch: 1168 traing_acc:1.000 tst_acc:0.933
epoch: 1169 traing_acc:1.000 tst_acc:0.933
epoch: 1170 traing_acc:1.000 tst_acc:0.933
epoch: 1171 traing_acc:1.000 tst_acc:0.933
epoch: 1172 traing_acc:1.000 tst_acc:0.933
epoch: 1173 traing_acc:1.000 tst_acc:0.933
epoch: 1174 traing_acc:1.000 tst_acc:0.933
epoch: 1175 traing_acc:1.000 tst_acc:0.933
epoch: 1176 traing_acc:1.000 tst_acc:0.933
epoch: 1177 traing_acc:1.000 tst_acc:0.933
epoch: 1178 traing_acc:1.000 tst_acc:0.933
epoch: 1179 traing_acc:1.000 tst_acc:0.933
epoch: 1180 traing_acc:1.000 tst_acc:0.933
epoch: 1181 traing_acc:1.000 tst_acc:0.933
epoch: 1182 traing_acc:1.000 tst_acc:0.933
epoch: 1183 traing_acc:1.000 tst_acc:0.933
epoch: 1184 traing_acc:1.000 tst_acc:0.933
epoch: 1185 traing_acc:1.000 tst_acc:0.933
epoch: 1186 traing_acc:1.000 tst_acc:0.933
epoch: 1187 traing_acc:1.000 tst_acc:0.933
epoch: 1188 traing_acc:1.000 tst_acc:0.933
epoch: 1189 traing_acc:1.000 tst_acc:0.933
epoch: 1190 traing_acc:1.000 tst_acc:0.933
epoch: 1191 traing_acc:1.000 tst_acc:0.933
epoch: 1192 traing_acc:1.000 tst_acc:0.933
epoch: 1193 traing_acc:1.000 tst_acc:0.933
```

```
epoch: 1194 traing_acc:1.000 tst_acc:0.933
epoch: 1195 traing_acc:1.000 tst_acc:0.933
epoch: 1196 traing_acc:1.000 tst_acc:0.933
epoch: 1197 traing_acc:1.000 tst_acc:0.933
epoch: 1198 traing_acc:1.000 tst_acc:0.933
epoch: 1199 traing_acc:1.000 tst_acc:0.933
```

```
[110]: <matplotlib.legend.Legend at 0x12c6f0dc0>
```



```
[110]:
```

knn

February 24, 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.

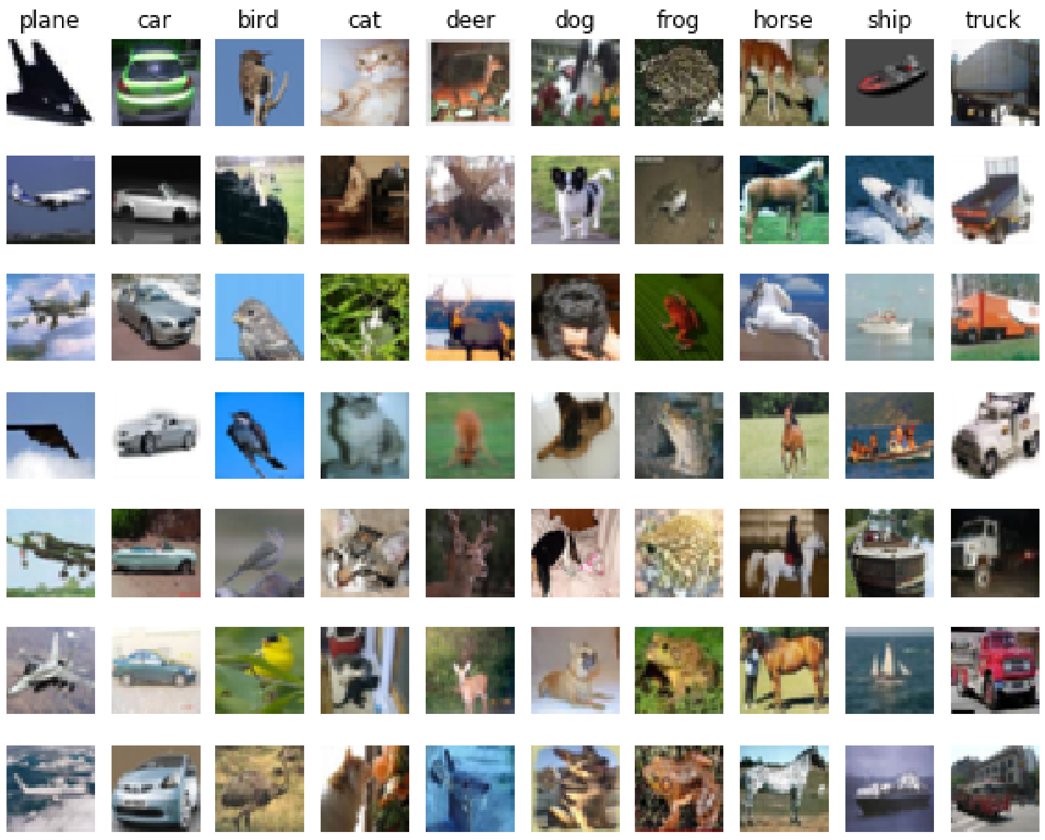
[30]:
```
# Run some setup code for this notebook.

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

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

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

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

[31]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'
# Cleaning up variables to prevent loading data multiple times (which may cause␣
 ↪memory issue)
try:
   del X_train, y_train
   del X_test, y_test
   print('Clear previously loaded data.')
except:
   pass

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

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

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

[32]:
```
# 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()
```



[33]:
```
# 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)
```

[34]:
```python
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.

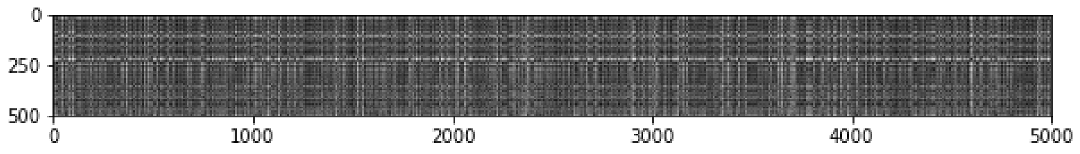
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

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.

[35]:
```python
# 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)
```

[36]:
```python
# 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?

$\color{blue}{\textit{Your Answer:}}$

If a row (say the corresponding test data r) is distinctly brighter than others, then it means the corresponding distances from r to all training points are all distinctly larger than other test points.

If a column (say the corresponding training data c) is distinctly brighter than others, then it means the corresponding distances from all test data to c are distinctly larger than other training points.

[37]:
```python
# 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`:

[38]:
```python
y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

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

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

Inline Question 2

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

the mean μ across all pixels over all images is

$$\mu=\frac{1}{nhw}\sum_{k=1}^{n}\sum_{i=1}^{h}\sum_{j=1}^{w}p_{ij}^{(k)}$$

And the pixel-wise mean μ_{ij} across all images is

$$\mu_{ij}=\frac{1}{n}\sum_{k=1}^{n}p_{ij}^{(k)}.$$

The general standard deviation σ and pixel-wise standard deviation σ_{ij} is defined similarly.

Which of the following preprocessing steps will not change the performance of a Nearest Neighbor classifier that uses L1 distance? Select all that apply. 1. Subtracting the mean μ ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu$.) 2. Subtracting the per pixel mean μ_{ij} ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu_{ij}$.) 3. Subtracting the mean μ and dividing by the standard deviation σ. 4. Subtracting the pixel-wise mean μ_{ij} and dividing by the pixel-wise standard deviation σ_{ij}. 5. Rotating the coordinate axes of the data.

$Your Answer:$

I choose 1, 2, 3, 5

$Your Explanation:$

The L1 distance is defined as follows:

$$d(I_p, I_q) = \sum_{i=1}^{h}\sum_{j=1}^{w} ||p_{i,j}^p - p_{i,j}^q||_1$$

Then define $d_i(\cdot,\cdot)$ as the new distance metric for the $i-th$ preprocessing step.

For $d_1(\cdot,\cdot)$, we have

$$d_1(I_p, I_q) = \sum_{i=1}^{h}\sum_{j=1}^{w} ||p_{i,j}^p - \mu - p_{i,j}^q + \mu||_1 = d(I_p, I_q)$$

For $d_2(\cdot,\cdot)$, we have

$$d_2(I_p, I_q) = \sum_{i=1}^{h}\sum_{j=1}^{w} ||p_{i,j}^p - \mu_{i,j} - p_{i,j}^q + \mu_{i,j}||_1 = d(I_p, I_q)$$

For $d_3(\cdot,\cdot)$, we have

$$d_3(I_p, I_q) = \sum_{i=1}^{h}\sum_{j=1}^{w} ||\frac{p_{i,j}^p - \mu}{\sigma} - \frac{p_{i,j}^q - \mu}{\sigma}||_1 = \frac{d(I_p, I_q)}{\sigma} \propto d(I_p, I_q)$$

For $d_4(\cdot,\cdot)$, we have

$$d_4(I_p, I_q) = \sum_{i=1}^{h}\sum_{j=1}^{w} ||\frac{p_{i,j}^p - \mu_{i,j}}{\sigma_{i,j}} - \frac{p_{i,j}^q - \mu_{i,j}}{\sigma_{i,j}}||_1 = \sum_{i=1}^{h}\sum_{j=1}^{w} ||\frac{p_{i,j}^p - p_{i,j}^q}{\sigma_{i,j}}||_1$$

For $d_5(\cdot,\cdot)$, we have

$$d_5(I_p, I_q) = \sum_{i=1}^{w}\sum_{j=1}^{h} ||p_{i,j}^p - p_{i,j}^q||_1 = d(I_p, I_q)$$

[39]:
```
# 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
```

[40]:
```
# 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
```

[41]:
```
# 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 12.713599 seconds
One loop version took 14.369608 seconds
No loop version took 0.144269 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.

[42]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

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

X_train_folds = np.array_split(X_train, num_folds)
y_train_folds = np.array_split(y_train, num_folds)

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

# A dictionary holding the accuracies for different values of k that we find
# when running cross-validation. After running cross-validation,
# k_to_accuracies[k] should be a list of length num_folds giving the different
# accuracy values that we found when using that value of k.
k_to_accuracies = {}
```

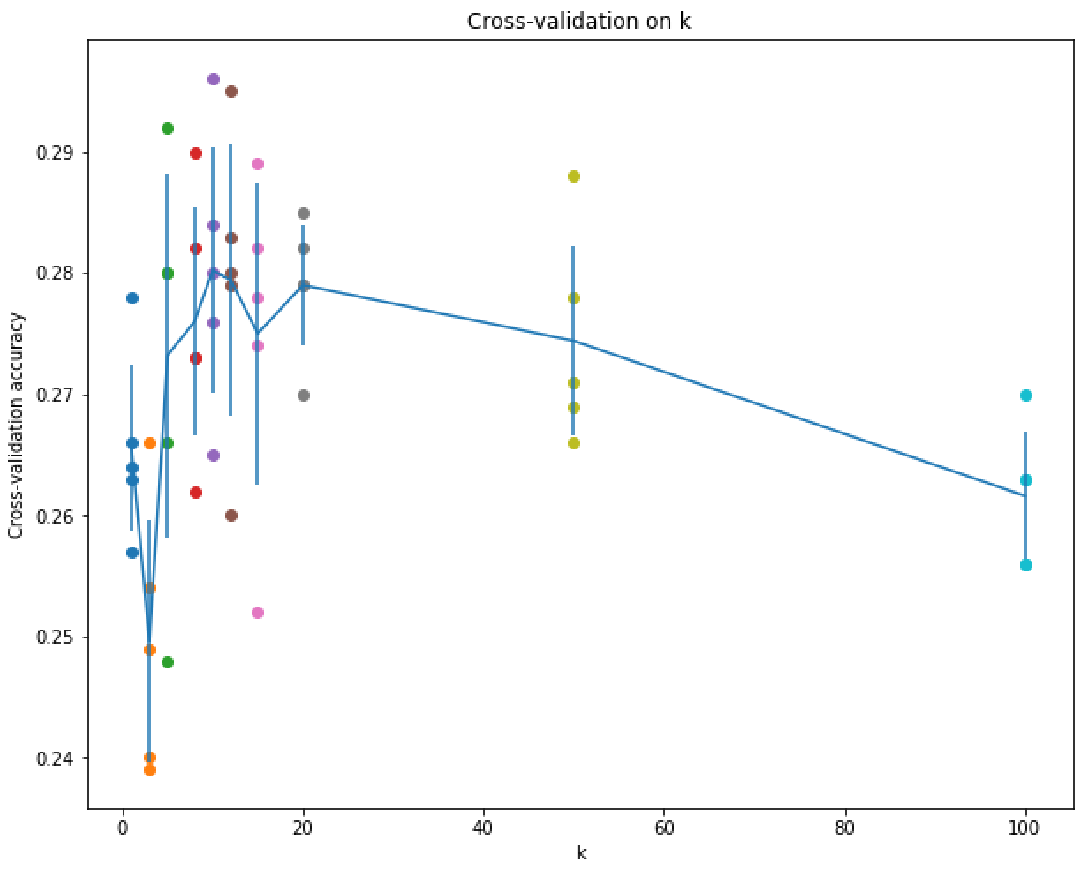
```
################################################################################
# TODO:                                                                        #
# Perform k-fold cross validation to find the best value of k. For each        #
# possible value of k, run the k-nearest-neighbor algorithm num_folds times,   #
# where in each case you use all but one of the folds as training data and the #
# last fold as a validation set. Store the accuracies for all fold and all     #
# values of k in the k_to_accuracies dictionary.                               #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

for k in k_choices:
    k_to_accuracies[k] = []
    for i in range(num_folds):
        X_train_fold = np.concatenate(X_train_folds[:i] + X_train_folds[i+1:])
        y_train_fold = np.concatenate(y_train_folds[:i] + y_train_folds[i+1:])
        X_val_fold = X_train_folds[i]
        y_val_fold = y_train_folds[i]
        classifier.train(X_train_fold, y_train_fold)
        y_pred = classifier.predict(X_val_fold, k=k)
        k_to_accuracies[k].append(np.sum(y_pred == y_val_fold) / y_val_fold.
 ↪size)

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

# Print out the computed accuracies
for k in sorted(k_to_accuracies):
    for accuracy in k_to_accuracies[k]:
        print('k = %d, accuracy = %f' % (k, accuracy))
```

```
k = 1, accuracy = 0.263000
k = 1, accuracy = 0.257000
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
k = 1, accuracy = 0.266000
k = 3, accuracy = 0.239000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.240000
k = 3, accuracy = 0.266000
k = 3, accuracy = 0.254000
k = 5, accuracy = 0.248000
k = 5, accuracy = 0.266000
k = 5, accuracy = 0.280000
k = 5, accuracy = 0.292000
k = 5, accuracy = 0.280000
k = 8, accuracy = 0.262000
k = 8, accuracy = 0.282000
```

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

[43]:

```python
# 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()
```



[44]:
```
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 10

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

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

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

Inline Question 3

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

$Your Answer:$

I choose 2, 4.

$Your Explanation:$

1. False. The decision boundary of the k-NN classifier is non-linear since the unit "sphere" under L2 norm is a unit circle in 2D and a unit sphere in 3D which is not linear.
2. True. The training error of a 1-NN is always 0 since the nearest neighbor of a point is itself.
3. False. The test error does not increase with k just like show in the picture above.
4. True. The process of classifying a test sample is to traverse through all the training data, which is an $O(n)$ operation.

```
[44]:
```

svm

February 25, 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

[2]:
```
# 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

[3]:
```
# 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,)
```

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

[5]:
```
# 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,)
```

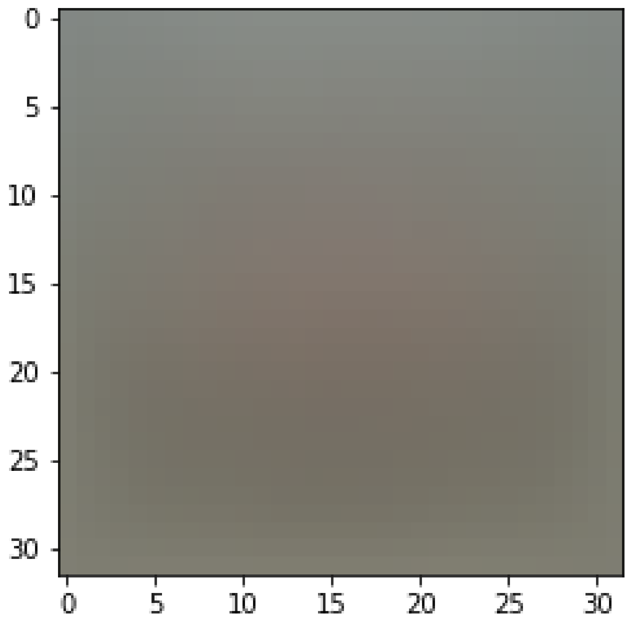
[6]:
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# 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)
```

[7]:
```
# 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.

[8]:
```
# Evaluate the naive implementation of the loss we provided for you:
from cs231n.classifiers.linear_svm import svm_loss_naive
import time

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

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

```
loss: 9.318401
```

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:

[9]:
```
# 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: 4.066394 analytic: 4.129151, relative error: 7.657412e-03
numerical: -5.600240 analytic: -5.600240, relative error: 7.350527e-11
numerical: 4.651978 analytic: 4.651978, relative error: 8.995426e-13
numerical: -22.334557 analytic: -22.334557, relative error: 1.318483e-11
numerical: -3.009755 analytic: -3.009755, relative error: 1.557965e-10
numerical: -11.344121 analytic: -11.344121, relative error: 4.304613e-13
numerical: 3.158882 analytic: 3.204693, relative error: 7.198895e-03
numerical: -18.502906 analytic: -18.382096, relative error: 3.275296e-03
numerical: -6.049070 analytic: -6.049070, relative error: 7.427929e-12
```

```
numerical: 30.572300 analytic: 30.572300, relative error: 7.611381e-12
numerical: -6.443605 analytic: -6.548512, relative error: 8.074646e-03
numerical: -3.362234 analytic: -3.294436, relative error: 1.018498e-02
numerical: -1.974795 analytic: -1.974795, relative error: 2.956606e-10
numerical: -42.768838 analytic: -42.768838, relative error: 2.457229e-12
numerical: 2.931244 analytic: 2.931244, relative error: 4.977740e-11
numerical: 1.526142 analytic: 1.601676, relative error: 2.414897e-02
numerical: -2.584603 analytic: -2.584603, relative error: 9.321052e-11
numerical: -12.417974 analytic: -12.417974, relative error: 2.307688e-11
numerical: -22.304136 analytic: -22.232664, relative error: 1.604777e-03
numerical: 25.988804 analytic: 25.988804, relative error: 1.067483e-11
```

Inline Question 1

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

$Your Answer:$

To begin with, the loss function is not totally differentiable at 0. Since the numerical result is get from approximation, the approximation will fail in the position that the loss function is not differentiable.

Given that the error is caused by approximation, it is not a concern.

A simple example in 1-D: consider ReLU function, $f(x) = max(0, x)$, if we approximate the gradient at $x = -0.02$ and take the interval length $h = 0.02$, then the approximated gradient is

$$f'(x) = \frac{f(0.01) - f(-0.03)}{2 \times 0.02} = \frac{1}{4} \neq 0$$

but the true gradient is 0.

The way to reduce the effect is to reduce the interval length h.

[10]:
```
# Next implement the function svm_loss_vectorized; for now only compute the␣
↪loss;
# we will implement the gradient in a moment.
tic = time.time()
loss_naive, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

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

# The losses should match but your vectorized implementation should be much␣
↪faster.
```

```
print('difference: %f' % (loss_naive - loss_vectorized))
```

```
Naive loss: 9.318401e+00 computed in 0.045376s
Vectorized loss: 9.318401e+00 computed in 0.008753s
difference: -0.000000
```

[11]:
```
# 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 0.075716s
Vectorized loss and gradient: computed in 0.009150s
difference: 0.000000
```

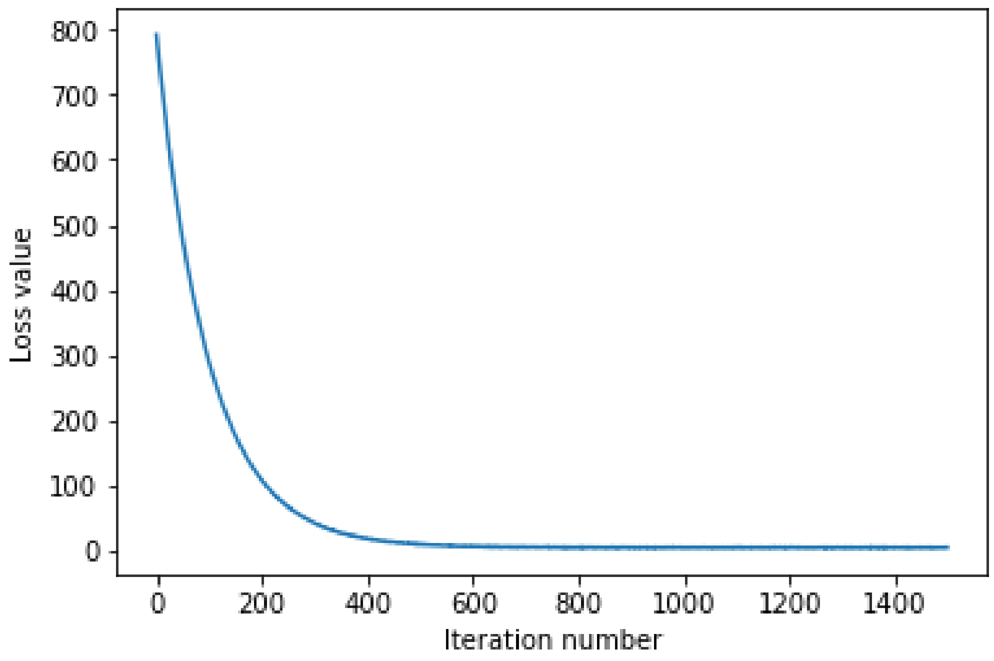
1.2.1 Stochastic Gradient Descent

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

[12]:
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cs231n.classifiers import LinearSVM
svm = LinearSVM()
tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 791.981194
iteration 100 / 1500: loss 289.379744
iteration 200 / 1500: loss 108.449161
iteration 300 / 1500: loss 42.430152
iteration 400 / 1500: loss 18.899496
iteration 500 / 1500: loss 10.146439
iteration 600 / 1500: loss 7.472539
iteration 700 / 1500: loss 5.959729
iteration 800 / 1500: loss 5.459304
iteration 900 / 1500: loss 5.492587
iteration 1000 / 1500: loss 5.032883
iteration 1100 / 1500: loss 5.570110
iteration 1200 / 1500: loss 5.641019
iteration 1300 / 1500: loss 5.025879
iteration 1400 / 1500: loss 5.251474
That took 7.209703s
```

[13]:
```
# 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()
```



[14]:
```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: 0.369653
validation accuracy: 0.386000
```

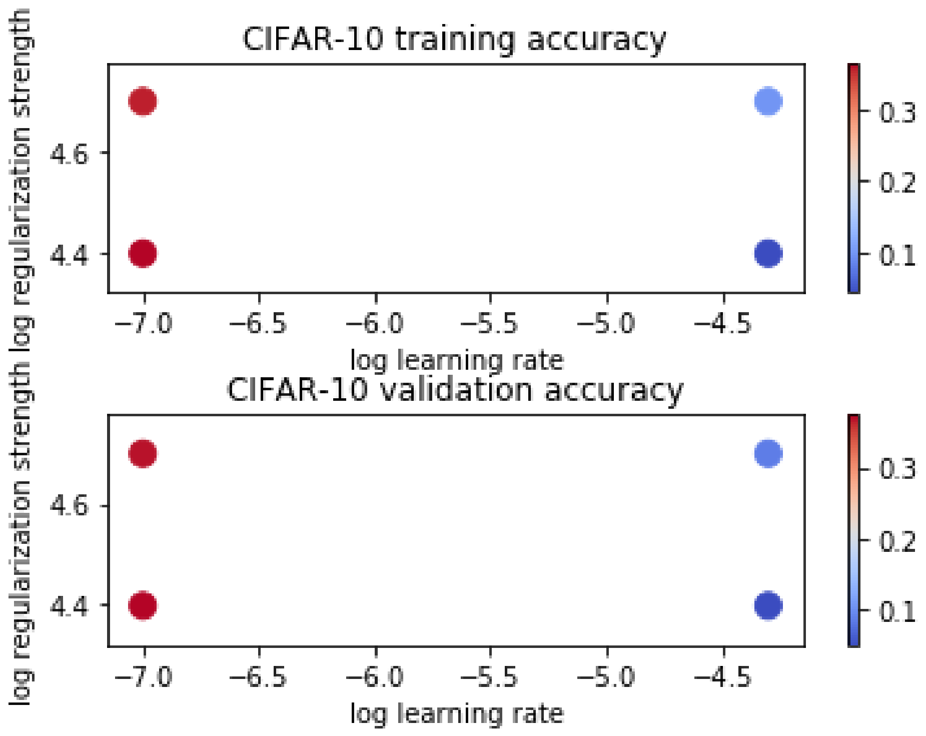
[15]:
```python
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.39 on the validation set.

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

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

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

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

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

for lr in learning_rates:
    for r in regularization_strengths:
        svm = LinearSVM()
        svm.train(X_train, y_train, learning_rate=lr, reg=r, num_iters=1500)
        y_train_pred = svm.predict(X_train)
        y_val_pred = svm.predict(X_val)
        train_accuracy = np.mean(y_train_pred == y_train)
        val_accuracy = np.mean(y_val_pred == y_val)
        results[(lr, r)] = (train_accuracy, val_accuracy)
        if val_accuracy > best_val:
            best_val = val_accuracy
            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)
```

```
/Users/caleblee/Desktop/CS280/CS280-Spring23-Assignment1/Homework1_partB/cs231n/
classifiers/linear_svm.py:88: RuntimeWarning: overflow encountered in scalar
multiply
  loss += reg * np.sum(W * W)
/Users/caleblee/Library/Python/3.9/lib/python/site-
packages/numpy/core/fromnumeric.py:86: RuntimeWarning: overflow encountered in
reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/Users/caleblee/Desktop/CS280/CS280-Spring23-Assignment1/Homework1_partB/cs231n/
classifiers/linear_svm.py:88: RuntimeWarning: overflow encountered in multiply
  loss += reg * np.sum(W * W)
/Users/caleblee/Desktop/CS280/CS280-Spring23-Assignment1/Homework1_partB/cs231n/
classifiers/linear_svm.py:106: RuntimeWarning: overflow encountered in multiply
  dW += reg * 2 * W
/Users/caleblee/Desktop/CS280/CS280-Spring23-Assignment1/Homework1_partB/cs231n/
classifiers/linear_svm.py:83: RuntimeWarning: invalid value encountered in
matmul
  loss = X @ W
/Users/caleblee/Desktop/CS280/CS280-Spring23-Assignment1/Homework1_partB/cs231n/
classifiers/linear_classifier.py:77: RuntimeWarning: invalid value encountered
in subtract
  self.W -= learning_rate * grad
```

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.366571 val accuracy: 0.375000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.358327 val accuracy: 0.371000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.043796 val accuracy: 0.049000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.375000
```

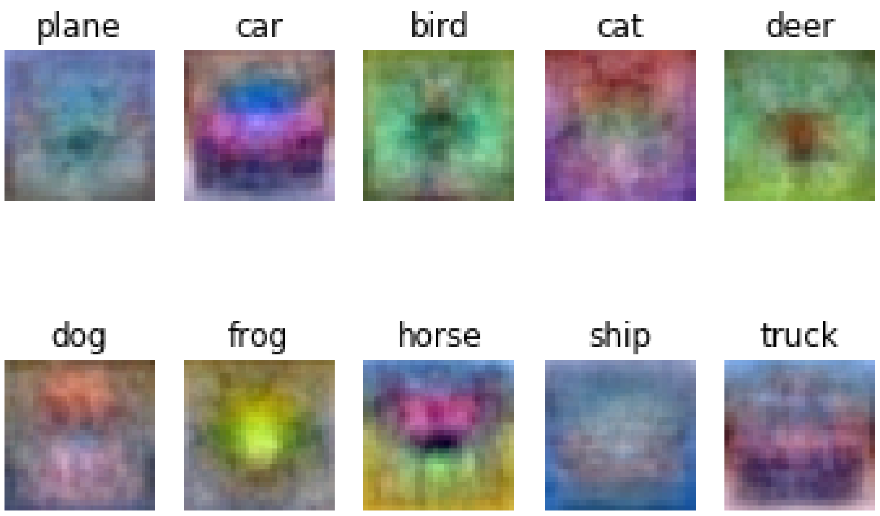
[16]:
```
# Visualize the cross-validation results
import math
import pdb

# 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()
```



[17]:
```python
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.369000
```

[18]:
```python
# 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.

$Your Answer:$

Those pictures look like the "average" pictures of each class. This is because the angle between w_k and x_i affects the value of $w_k^T x_i$. Thus, the smaller the angle is, the more similar w_k is to x_i and the higher the probability that x_i si classified into class k.

[18]:
```

```

softmax

February 24, 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.

[3]:
```
# 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: 2.370673
sanity check: 2.302585
```

Inline Question 1

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

$YourAnswer:$

Since the weight W is randomly initialized, then the probability p_i should be close to 1/{number of class}=0.1 for each class i. Thus, the value of loss function can be approximated as follows:

$$L = \frac{1}{n}\sum_{i=1}^{n} -\log p_i = -\log(\prod_{i=1}^{n} p_i)^{\frac{1}{n}} \approx -\log((0.1)^n)^{(1/n)} = -log 0.1$$

[4]:
```python
# 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: 1.052860 analytic: 1.052860, relative error: 2.201441e-08
numerical: -1.549930 analytic: -1.549930, relative error: 1.647477e-08
numerical: 0.119256 analytic: 0.119256, relative error: 3.456103e-07
numerical: -0.219556 analytic: -0.219556, relative error: 8.991791e-08
numerical: 0.083568 analytic: 0.083568, relative error: 5.460927e-07
numerical: 1.443529 analytic: 1.443529, relative error: 8.874940e-09
numerical: 0.796921 analytic: 0.796921, relative error: 4.137497e-08
numerical: -4.262104 analytic: -4.262104, relative error: 4.176475e-09
numerical: -1.373423 analytic: -1.373423, relative error: 8.834602e-10
numerical: 0.489926 analytic: 0.489926, relative error: 2.588278e-09
numerical: 1.725037 analytic: 1.725037, relative error: 1.961117e-08
numerical: -0.927769 analytic: -0.927769, relative error: 1.667269e-08
numerical: -0.186378 analytic: -0.186378, relative error: 1.390873e-07
numerical: 1.496357 analytic: 1.496357, relative error: 5.722065e-08
numerical: 0.658420 analytic: 0.658420, relative error: 6.588213e-08
numerical: 0.621139 analytic: 0.621139, relative error: 7.778886e-08
numerical: 1.582019 analytic: 1.582019, relative error: 4.703156e-08
numerical: -0.340721 analytic: -0.340721, relative error: 1.002202e-07
numerical: 0.169129 analytic: 0.169129, relative error: 5.742597e-08
numerical: -0.260551 analytic: -0.260551, relative error: 2.454967e-07
```

[5]:
```python
# 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: 2.370673e+00 computed in 0.037366s
vectorized loss: 2.370673e+00 computed in 0.005350s
Loss difference: 0.000000
Gradient difference: 0.000000
```

[6]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of over 0.35 on the validation set.

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

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

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

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

for lr in learning_rates:
    for r in regularization_strengths:
        softmax = Softmax()
        loss_hist = softmax.train(X_train, y_train, learning_rate=lr, reg=r,
                      num_iters=1500, verbose=False)
        y_train_pred = softmax.predict(X_train)
        y_val_pred = softmax.predict(X_val)
        train_accuracy = np.mean(y_train == y_train_pred)
        val_accuracy = np.mean(y_val == y_val_pred)
        results[(lr, r)] = (train_accuracy, val_accuracy)
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_softmax = softmax

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

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

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

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.328388 val accuracy: 0.340000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.303184 val accuracy: 0.318000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.326327 val accuracy: 0.331000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.298510 val accuracy: 0.311000
best validation accuracy achieved during cross-validation: 0.340000
```

[7]:
```
# 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: 0.344000
```

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.

$\color{blue}{\textit{Your Answer :}}$

Yes, it’s possible.

YourExplanation :

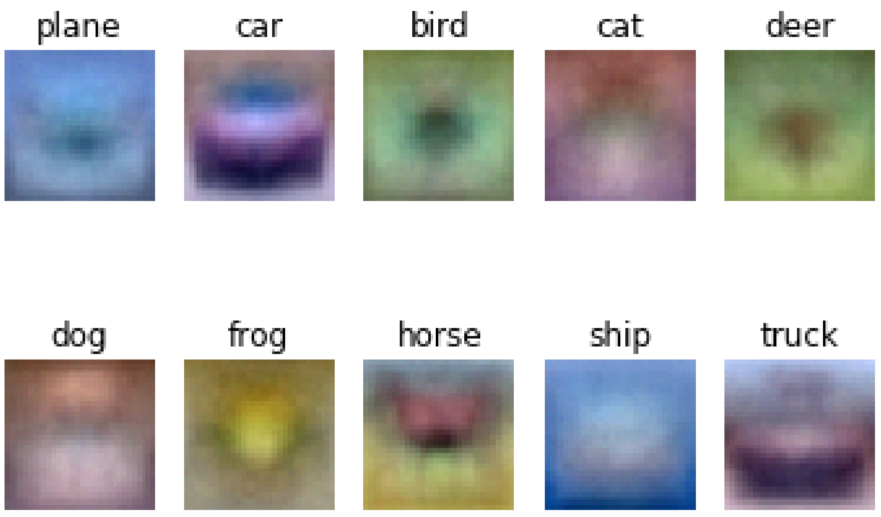
Given that the loss function of SVM is as follows:

$$L = \frac{1}{n}\sum_{i=1}^{n}\sum_{j\neq y_i} max(0, 1-(w_j - w_{y_i})x_i)$$

if the new added point x_i has the property that $(w_j - w_{y_i})x_i < 1$ for all j then the loss value will not change

However, for softmax, the loss function takes every training points into consideration. Thus, the value will change if we adda new point x_i into training set.

[8]:
```
# 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])
```



[8]:

```
```

two_layer_net

February 24, 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.

[16]:
```
# A bit of setup

import numpy as np
import matplotlib.pyplot as plt

from cs231n.classifiers.neural_net import TwoLayerNet

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

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

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

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

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

[17]:
```
# 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.

[18]:
```
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
  [-0.81233741, -1.27654624, -0.70335995],
  [-0.17129677, -1.18803311, -0.47310444],
  [-0.51590475, -1.01354314, -0.8504215 ],
  [-0.15419291, -0.48629638, -0.52901952],
  [-0.00618733, -0.12435261, -0.15226949]])
print(correct_scores)
print()

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

```
Your scores:
[[-0.81233741 -1.27654624 -0.70335995]
```

```
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

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

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

3 Forward pass: compute loss

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

[23]:
```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.30378789133

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

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

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:

[24]:
```
from cs231n.gradient_check import eval_numerical_gradient

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

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

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

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

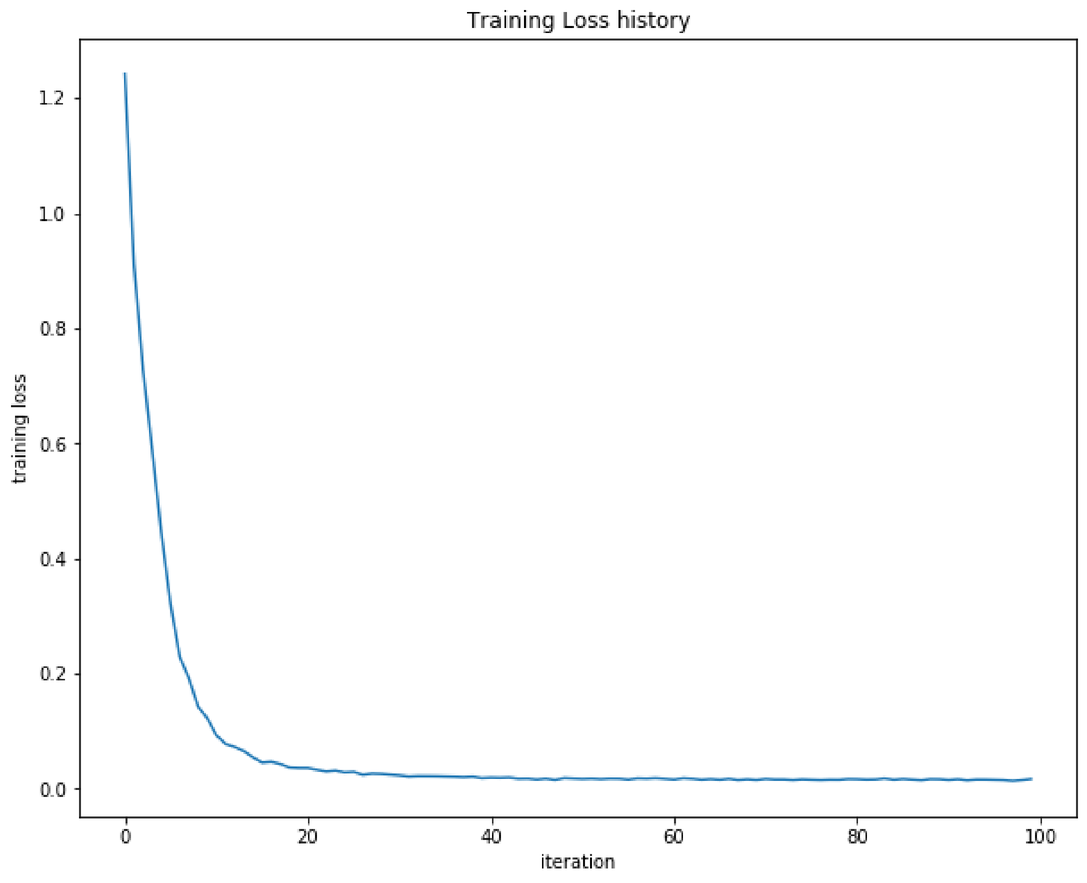
5 Train the network

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.

[25]:
```
net = init_toy_model()
stats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, verbose=False)

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

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

```
Final training loss:  0.017149607938732048
```



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.

[26]:
```
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.

[27]:
```python
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

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

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

```
iteration 0 / 1000: loss 2.302954
iteration 100 / 1000: loss 2.302550
iteration 200 / 1000: loss 2.297648
iteration 300 / 1000: loss 2.259602
iteration 400 / 1000: loss 2.204170
iteration 500 / 1000: loss 2.118565
iteration 600 / 1000: loss 2.051535
iteration 700 / 1000: loss 1.988466
iteration 800 / 1000: loss 2.006591
iteration 900 / 1000: loss 1.951473
Validation accuracy:  0.287
```

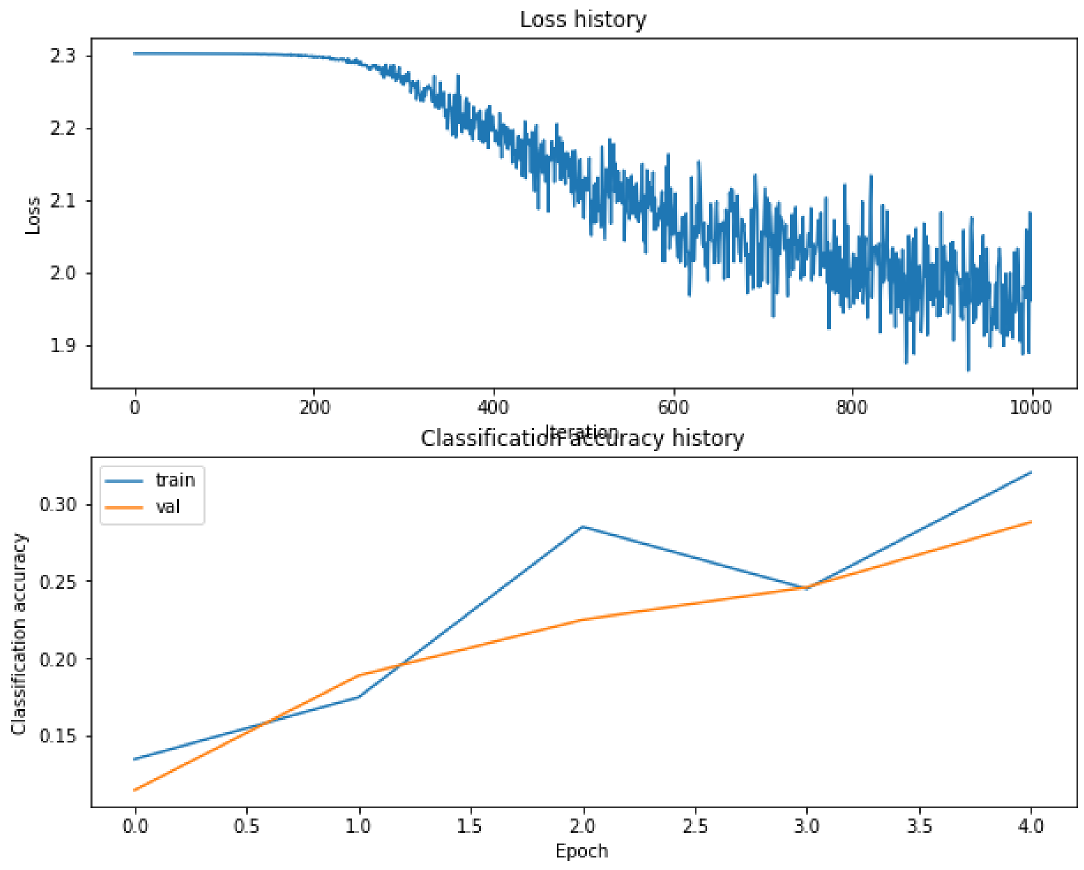
8 Debug the training

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.

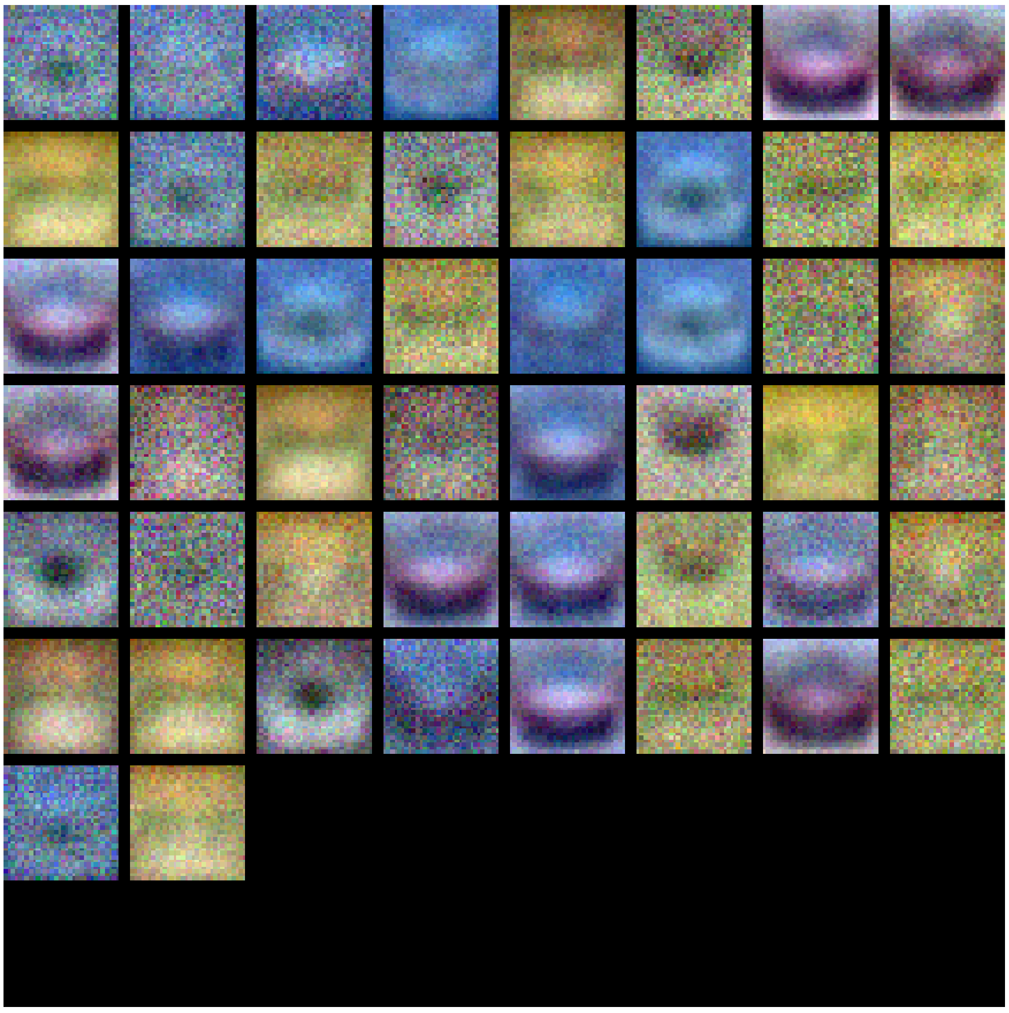
Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

[28]:
```
# 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()
```



[29]:

```python
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.

$Your Answer :$

I experimented with different hyperparameters combinations, in detail, I choose two different learning_rates: 5e-4 and 1e-3 and two different batch_size: 200 and 400.

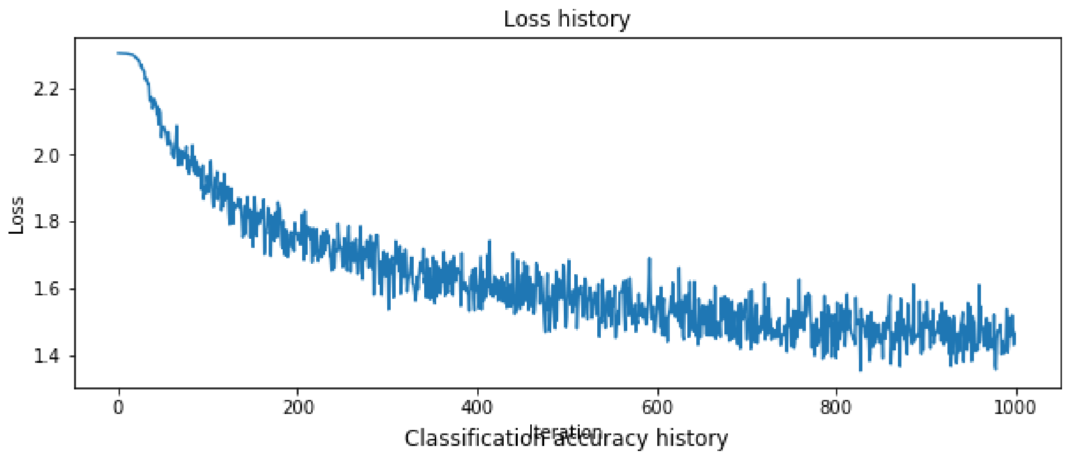
The reason why I choose to tune learning_rate is that learning rate is the plot above shows that the loss fluctuated heavily during the training process, then we can increase the learning rate to by increasing 5e-4 each time. As for batch_size, increase batch size will help the model to get more accurate gradient during training, so we compare the result of original(200) parameter and new (400) parameter.

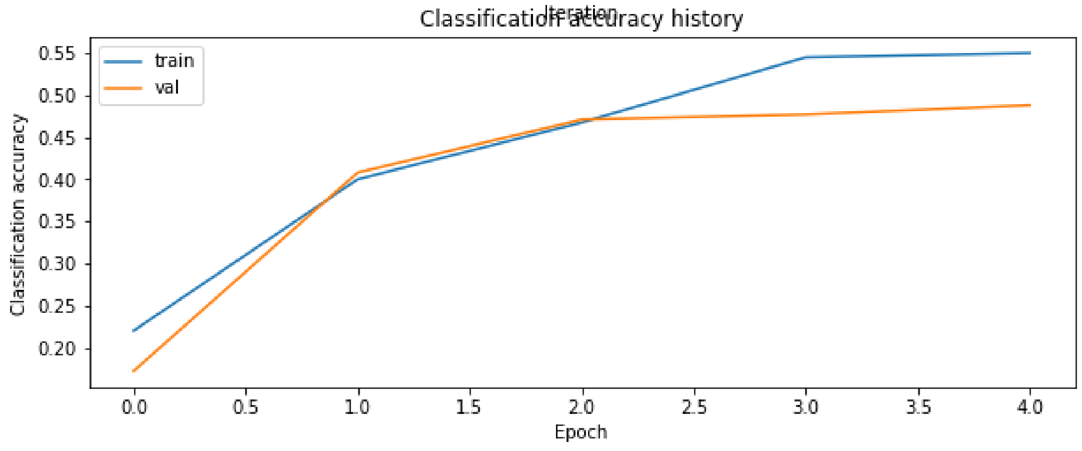
I traversed through all possible combinations of hyperparameters and save the model with the highest validation accuracy as the best model.

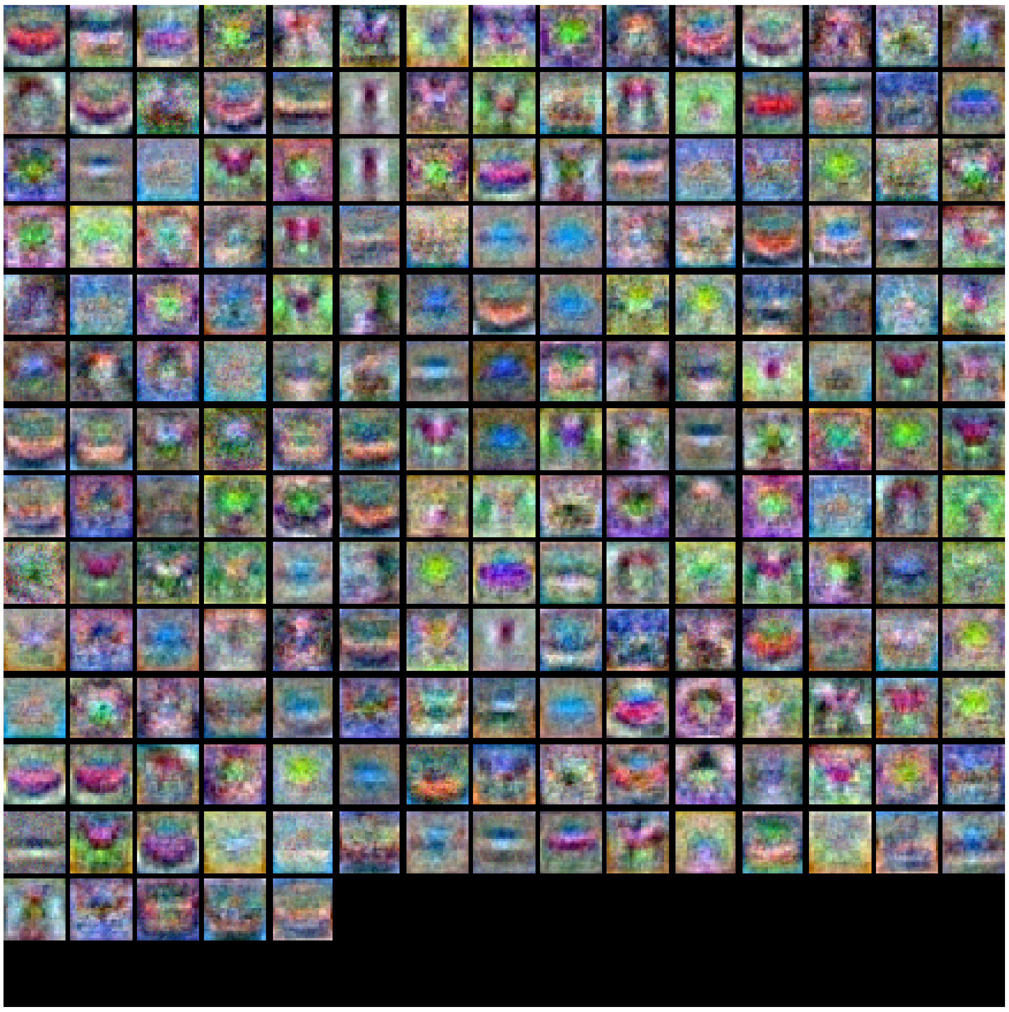
[42]:
```
best_net = None # store the best model into this

#################################################################################
# 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)*****
learning_rates = [5e-4, 1e-3]
batch_size = [200, 400]
results = {}
best_val = -1
best_stats = None

for lr in learning_rates:
    for b in batch_size:
        net = TwoLayerNet(input_size, 200, num_classes)
        stats = net.train(X_train, y_train, X_val, y_val,
                    num_iters=1000, batch_size=b,
                    learning_rate=lr, learning_rate_decay=0.95,
                    reg=0.25, verbose=False)
        train_acc = (net.predict(X_train) == y_train).mean()
        val_acc = (net.predict(X_val) == y_val).mean()
        results[(lr, b)] = (train_acc, val_acc)
        if val_acc > best_val:
            best_val = val_acc
            best_net = net
            best_stats = stats
        print(f"lr={lr}, batch_size={b}, train_acc={train_acc},␣
↪val_acc={val_acc}")

plt.subplot(2, 1, 1)
plt.plot(best_stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')

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

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

```
lr=0.0005, batch_size=200, train_acc=0.4638775510204082, val_acc=0.449
lr=0.0005, batch_size=400, train_acc=0.4653673469387755, val_acc=0.468
lr=0.001, batch_size=200, train_acc=0.5078571428571429, val_acc=0.481
lr=0.001, batch_size=400, train_acc=0.5173673469387755, val_acc=0.497
```



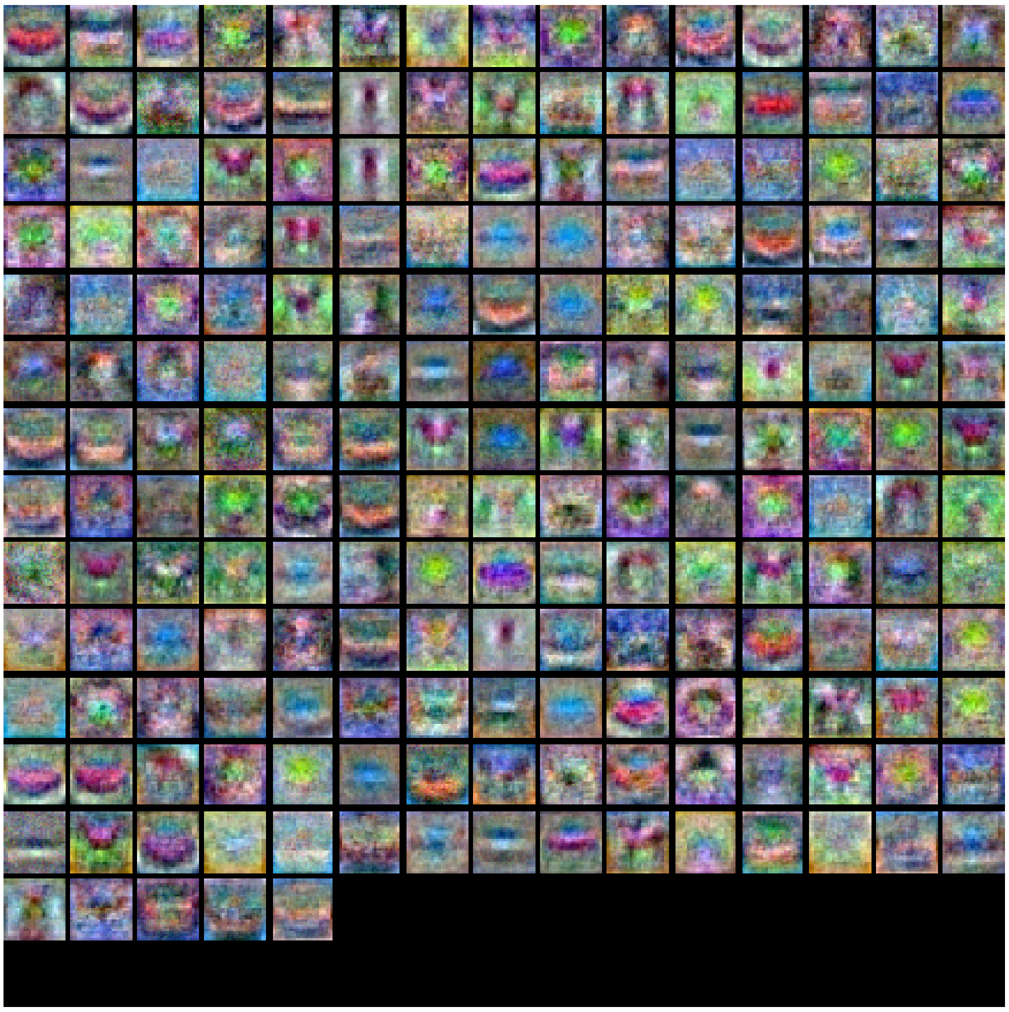




```
[43]: # Print your validation accuracy: this should be above 48%
      val_acc = (best_net.predict(X_val) == y_val).mean()
      print('Validation accuracy: ', val_acc)
```

```
Validation accuracy:  0.497
```

```
[44]: # 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%.

[45]:
```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:  0.502
```

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.

$YourAnswer:$

I choose 1, 2, 3.

$YourExplanation:$

Actually this problem is equivalent to how to avoid overfitting because overfitting means the model nearly remembered the whole training set. For 1, the more data the model meet, the less likely that it remember all of them. For 2, the more hidden units the model has, the harder for training and the model is less likely to remember all the dataset. For 3, regularization constrains the L2 norm of weights won't be extremely large which means the model won't approximation the implicit function in an extremely complex way.

```
[ ]:
```

features

February 24, 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)]
```

```
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.

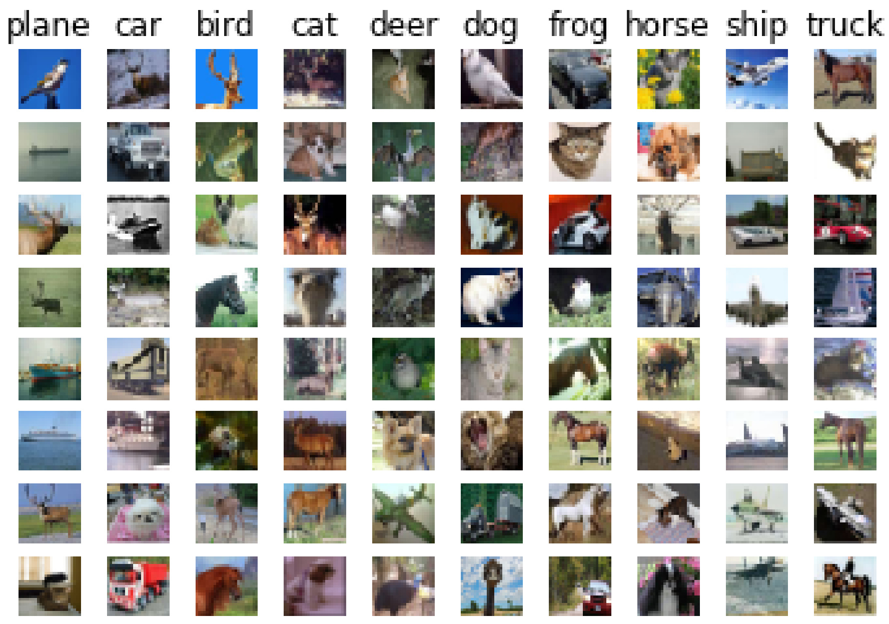
[5]:
```python
# Use the validation set to tune the learning rate and regularization strength

from cs231n.classifiers.linear_classifier import LinearSVM

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

results = {}
best_val = -1
best_svm = None

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained classifer in best_svm. You might also want to play          #
# with different numbers of bins in the color histogram. If you are careful    #
# you should be able to get accuracy of near 0.44 on the validation set.        #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

for lr in learning_rates:
    for r in regularization_strengths:
        svm = LinearSVM()
        svm.train(X_train_feats, y_train, learning_rate=1e-7, reg=2.5e4,␣
 ↪num_iters=1500)
        y_train_pred = svm.predict(X_train_feats)
        y_val_pred = svm.predict(X_val_feats)
        train_accuracy = np.mean(y_train_pred == y_train)
        val_accuracy = np.mean(y_val_pred == y_val)
        results[(lr, r)] = (train_accuracy, val_accuracy)
        if val_accuracy > best_val:
            best_val = val_accuracy
            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)
```

```
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.412286 val accuracy: 0.419000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.411429 val accuracy: 0.416000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.414633 val accuracy: 0.423000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.418000 val accuracy: 0.424000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.415939 val accuracy: 0.420000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.416163 val accuracy: 0.419000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.412673 val accuracy: 0.409000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.411714 val accuracy: 0.413000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.411959 val accuracy: 0.409000
best validation accuracy achieved during cross-validation: 0.424000
```

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

```
0.423
```

[7]:
```
# 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 :

The misclassified images show features of both their corresponding labels and the labels they are classified into. For example, in the first column, most of the images have the streamline shape which is also an important feature of plane. Moreover, the color of images in "dog" column all have grey style, which is also the color of some typical dogs (e.g. husky and alaskan dog). In conclusion, it's reasonable for the model to misclassify.

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.

[8]:
```
# 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)
```

[14]:
```
from cs231n.classifiers.neural_net import TwoLayerNet

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

best_net = None

################################################################################
# TODO: Train a two-layer neural network on image features. You may want to    #
# cross-validate various parameters as in previous sections. Store your best   #
# model in the best_net variable.                                              #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

learning_rates = [0.5, 1]
batch_size = [200, 400]

best_val = -1
```

```
for lr in learning_rates:
    for b in batch_size:
        net = TwoLayerNet(input_dim, hidden_dim, num_classes)
        net.train(X_train_feats, y_train, X_val_feats, y_val, learning_rate=lr,␣
 ↪learning_rate_decay=0.95, reg=2.5e-3, num_iters=1500, batch_size=b)
        y_train_pred = net.predict(X_train_feats)
        y_val_pred = net.predict(X_val_feats)
        train_accuracy = np.mean(y_train_pred == y_train)
        val_accuracy = np.mean(y_val_pred == y_val)
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_net = net
        print(f"lr={lr}, batch_size={b}, train_acc={train_accuracy},␣
 ↪val_acc={val_accuracy}")
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
lr=0.5, batch_size=200, train_acc=0.6035918367346939, val_acc=0.573
lr=0.5, batch_size=400, train_acc=0.6208775510204082, val_acc=0.588
lr=1, batch_size=200, train_acc=0.5911224489795919, val_acc=0.565
lr=1, batch_size=400, train_acc=0.5933265306122449, val_acc=0.536
```

[15]:
```
# 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)
```

```
0.567
```

[ ]:
```

```