Homework 3

1. (a)

As instructed, suppose, the number of points are odd, and suppose the points are in order that is

$$x_1 \leq x_2 \leq \cdots \leq x_{n-1} \leq x_n$$

And the median point of these datapoints is $x_k, s.t.\ k = \frac{n+1}{2}$

Also, we can obtain x_k is median by the definition.

We have the function

$$\mathrm{L(v)} = \sum_{i=1}^{n} |x_i - v|$$

Suppose, there is ∂ points left to the point v ($0 \leq \partial \leq$ n), so obviously there is (n-∂) points on the right of v.

Every time, the v moves ε to the left, the ∂ points are each getting ε closer while (n-∂) points are getting ε further; on the other hand, if v goes ε to the right, the situation is exactly opposite.

Based on that, if v is left to median, which means $\partial < \mathrm{n} - \partial$, if v goes ε to the right, the $\mathrm{L(v} + \varepsilon) = \mathrm{L(v)} - (\mathrm{n} - 2\,\partial)\varepsilon < \mathrm{L(v)}$; on the contrary, if v is right to median, which means $\partial > \mathrm{n} - \partial$, if v goes ε to the left, the $\mathrm{L(v} - \varepsilon) = \mathrm{L(v)} + (\mathrm{n} - 2\,\partial)\varepsilon > \mathrm{L(v)}$.

Therefore, the v can be minimizer of L(v) if and only if v satisfies that $\partial = \mathrm{n} - \partial$. In other words, such v is median.

We hence conclude that median is a value that minimize the function.

(b)

$$\mathrm{L(v)} = \sum_{i=1}^{n} (x_i - v)^2$$

The function is shown above, and in the following, the conclusion that the mean is the minimizer of this function will be shown.

Let $\mu = \frac{1}{\mathrm{n}} \sum_{i=1}^{n} (x_i)$.

The function can be rewritten as

$$\mathrm{L(v)} = \sum_{i=1}^{n} (x_i - v)^2 = \sum_{i=1}^{n} ((x_i - \mu) + (\mu - v))^2$$

$$= \sum_{i=1}^{n} (x_i - \mu)^2 + (\mu - v)^2 + 2(\mu - v)(x_i - \mu)$$

$$= \sum_{i=1}^{n} (\mu - v)^2 + \sum_{i=1}^{n} (x_i - \mu)^2 + 2(\mu - v) \sum_{i=1}^{n} (x_i - \mu)$$

$$= \sum_{i=1}^{n} (x_i - \mu)^2 + n(\mu - v)^2 + 2(\mu - v) \left(\sum_{i=1}^{n} (x_i) - n\mu \right)$$

$$= \sum_{i=1}^{n} (x_i - \mu)^2 + n(\mu - v)^2 + 2(\mu - v) \cdot 0$$

$$= \sum_{i=1}^{n} (x_i - \mu)^2 + n(\mu - v)^2$$

Since $\sum_{i=1}^{n} (x_i - \mu)^2$ is settled when x_i and μ are settled, therefore,

$$\sum_{i=1}^{n} (x_i - \mu)^2 + n(\mu - v)^2 \geq \sum_{i=1}^{n} (x_i - \mu)^2$$

Hence minimizer of L is mean. Therefore, mean is value that minimizes the function.

(c)

Similarly, as for median, the function is

$$\mathrm{L(v)} = \sum_{i=1}^{n} ||x_i - v||_2$$

Just as what we did in (a), we suppose the number of points is odd, and in this case, we use a hyperplane to divide the points into two parts where there is ∂ points on one side ($0 \leq \partial \leq$ n), (n-∂) points on the other side of hyperplane H. Similar to (a), when $\partial = \mathrm{n} - \partial$, there would be optimal. For every hyperplane, that divide the points into equal half, and their union is the optimal v which by the definition is median.

As for mean,

$$\mathrm{L(v)} = \sum_{i=1}^{n} ||x_i - v||_2^2$$

It can be rewritten similarly to what we've done in (b), that is

$$\mathrm{L(v)} = \sum_{i=1}^{n} ||x_i - \mu||_2^2 + n||\mathrm{v} - \mu||_2^2$$

We can also conclude that the optimizer of function is v equal to mean. Therefore, from above, we general our conclusions from (a), (b) to high dimension.

2.

For all questions we have definition for mean and cost below

$$\text{mean(C)} = \frac{1}{|C|}\sum_{x\in C} x$$

$$\text{cost(C)} = \sum_{x\in C} ||x - mean(C)||^2$$

(a)

Based on formula and information, we easily have

$$\mu_1 = \frac{1}{m_1}\sum_{x\in S_1} x$$

$$\mu_2 = \frac{1}{m_2}\sum_{x\in S_2} x$$

Therefore, since $\text{S} = S_1 \cup S_2$, we can obtain

$$\mu = \frac{1}{|S|}\sum_{x\in S} x$$

$$= \frac{1}{m_1 + m_2}\left(\sum_{x\in S_1} x + \sum_{x\in S_2} x\right)$$

$$= \frac{1}{m_1 + m_2}(m_1\mu_1 + m_2\mu_2)$$

(b)

Using the cost formula, we can have

$$\text{cost(S)} = \sum_{x\in S} ||x - \mu||^2$$

$$\text{cost}(S_1) = \sum_{x\in S_1} ||x - \mu_1||^2$$

$$\text{cost}(S_2) = \sum_{x\in S_2} ||x - \mu_2||^2$$

using the formula provided as follow

$$\sum_{x\in C} ||x - z||^2 = \sum_{x\in C} ||x - mean(C)||^2 + |C|||z - mean(C)||^2$$

we can obtain that

$$\text{cost}(S_1) = \sum_{x \in S_1} ||x - \mu_1||^2 = \sum_{x \in S_1} ||x - \mu||^2 - m_1||\mu - \mu_1||^2$$

$$\text{cost}(S_2) = \sum_{x \in S_2} ||x - \mu_2||^2 = \sum_{x \in S_2} ||x - \mu||^2 - m_2||\mu - \mu_2||^2$$

thus, we can have

$$\text{cost(S)} - \big(\text{cost}(S_1) + \text{cost}(S_2)\big)$$

$$= \sum_{x \in S} \big||x - \mu|\big|^2 - \sum_{x \in S_1} \big||x - \mu|\big|^2 - \sum_{x \in S_2} \big||x - \mu|\big|^2$$

$$+ m_1\big||\mu - \mu_1|\big|^2 + m_2\big||\mu - \mu_2|\big|^2$$

$$= m_1\big||\mu - \mu_1|\big|^2 + m_2\big||\mu - \mu_2|\big|^2$$

(c) from (a) $\mu = \frac{1}{m_1 + m_2}(m_1\mu_1 + m_2\mu_2)$ and (b) $\text{cost(S)} - \big(\text{cost}(S_1) + \text{cost}(S_2)\big) = m_1\big||\mu - \mu_1|\big|^2 + m_2\big||\mu - \mu_2|\big|^2$, we can obtain

$$\text{cost(S)} - \big(\text{cost}(S_1) + \text{cost}(S_2)\big)$$

$$= m_1 \left|\left|\frac{1}{m_1 + m_2}(m_1\mu_1 + m_2\mu_2) - \mu_1\right|\right|^2$$

$$+ m_2 \left|\left|\frac{1}{m_1 + m_2}(m_1\mu_1 + m_2\mu_2) - \mu_2\right|\right|^2$$

$$= m_1 \left|\left|\frac{m_2(\mu_2 - \mu_1)}{m_1 + m_2}\right|\right|^2 + m_2 \left|\left|\frac{m_1(\mu_1 - \mu_2)}{m_1 + m_2}\right|\right|^2$$

$$= \frac{m_1 m_2}{m_1 + m_2}\big||\mu_1 - \mu_2|\big|^2$$

And we notice that Ward's method for computing the distance is

$$\text{dist}(\text{C}, C') = \frac{|C| \cdot |C'|}{|C| + |C'|} \left|\left|\text{mean}(\text{C}) - \text{mean}(C')\right|\right|^2$$

if we compute the distance between S_1 and S_2, it should be

$$\text{dist}(S_1, S_2) = \frac{|S_1| \cdot |S_2|}{|S_1| + |S_2|} \left|\left|\text{mean}(S_1) - \text{mean}(S_2)\right|\right|^2$$

$$= \frac{m_1 m_2}{m_1 + m_2} \left|\left|\mu_1 - \mu_2\right|\right|^2$$

$$= \text{cost}(\text{S}) - \left(\text{cost}(S_1) + \text{cost}(S_2)\right)$$

Therefore, they're equal, it means that we divide set into two sets, the cost that saved is the distance from such two sets, that's how the cost is related to Ward's method.

(d)

From (c), we can have this equation

$$\text{cost}(S_1) + \text{cost}(S_2) = \text{cost}(\text{S}) - \text{dist}(S_1, S_2)$$

therefore, we can know that if dividing into two clusters, to better minimize the overall cost, it's equal to maximize the distance.

So a greedy top-down algorithm could be

(i) see the whole datapoints as one cluster at first

(ii) compute the distance using Ward's method in the cluster

(iii) select the largest distance pair as new centers and divide the cluster into two smaller clusters

(iv) repeat (ii) and (iii) until the until the desired number of

clusters is obtained.

Above is what I can think for a clustering algorithm using the k-means cost.

3.

(a)

Based on the information, we can compute the matrix

$$\mathrm{M} = [\mathrm{u}_1, \mathrm{u}_2] \begin{bmatrix} \lambda_1 & \\ & \lambda_2 \end{bmatrix} [\mathrm{u}_1, \mathrm{u}_2]^T$$

$$= \frac{1}{5} \begin{bmatrix} 2 & -1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ -1 & 2 \end{bmatrix}$$

$$= \begin{bmatrix} 7 & 6 \\ 6 & -2 \end{bmatrix}$$

(b)

The new eigenvalues should be

$$\lambda_1{}' = \lambda_1 + 2 = 4, \lambda_2{}' = \lambda_2 + 2 = 1$$

(c)

The new eigenvalues should be

$$\lambda_1{}'' = \lambda_1{}^2 = 4, \lambda_2{}'' = \lambda_2{}^2 = 1$$

4.

(a)

Let $v_1 = (1,1,1)$, $v_2 = (-1,-1,1)$ and $v_3 = (2,4,5)$

Then the normal vector for subspace is

$$\mathrm{n} = v_1 \times v_2 = (2, -2, 0)$$

then we have target vector projection on n that is

$$p' = \frac{{v_3}^T n}{||\mathrm{n}||_2^2} n = (-1,1,0)$$

Hence projection onto subspace is

$$\mathrm{p} = v_3 - p' = (3,3,5)$$

(b)

Let $\mathrm{v} = v_1 - v_2 = (2,2,0)$, then the unit direction vector should be

$$\mathrm{u} = \frac{v}{||v||} = \frac{1}{\sqrt{2}}(1,1,0)$$

and we know that affine space pass through the point v_1

hence the affine space can be descripted as

$$\mathrm{S} = \{(1 + \mathrm{t}, 1 + \mathrm{t}, 1) | \mathrm{t} \in \mathrm{R}\}$$

5. For this question, we have following properties

$$\mathrm{U} = \left[u_1, u_2 \dots u_p\right], \mathrm{V} = \left[v_1, v_2 \dots v_p\right]^T$$

And we have

$$\mathrm{U}^T U = I, \mathrm{V}^T V = I$$

Here I is Identity matrix

Also,

$${u_i}^T u_j = \begin{cases} 0, i \neq j \\ 1, i = j \end{cases}$$

$${v_i}^T v_j = \begin{cases} 0, i \neq j \\ 1, i = j \end{cases}$$

And the symmetrical for matrix

$$\Lambda = \begin{pmatrix} \sigma_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma_p \end{pmatrix} = \Lambda^T$$

$$\Rightarrow M^T = (U\Lambda \mathrm{V}^T)^T = V\Lambda^T \mathrm{U}^T = V\Lambda \mathrm{U}^T$$

And we can use them for following expression

(a)

$$Mv_i = U\Lambda \mathrm{V}^T v_i = U\big(\Lambda(\mathrm{V}^T v_i)\big)$$

$$= \big[u_1, u_2 \dots u_p\big]\left(\begin{pmatrix} \sigma_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma_p \end{pmatrix}\left(\big[v_1, v_2 \dots v_p\big]^T v_i\right)\right)$$

$$= \big[u_1, u_2 \dots u_p\big]\left(\begin{pmatrix} \sigma_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma_p \end{pmatrix}([0,0,\dots,1,\dots 0]^T)\right) \ (1 \ at \ position \ i)$$

$$= \big[u_1, u_2 \dots u_p\big][0,0,\dots,\sigma_i,\dots 0]^T = \sigma_i u_i$$

Hence

$$Mv_i = \sigma_i u_i$$

(b)

Similarly, using properties have been showed at the first place

$$M^T u_i = V\Lambda \mathrm{U}^T u_i$$

$$= \big[v_1, v_2 \dots v_p\big]\left(\begin{pmatrix} \sigma_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma_p \end{pmatrix}\left(\big[u_1, u_2 \dots u_p\big]^T u_i\right)\right)$$

$$= \sigma_i v_i$$

(c)

Still, using the properties

$$M^T M = V\Lambda(\mathrm{U}^T U)\Lambda\mathrm{V}^T = V\Lambda^2\mathrm{V}^T$$

$$MM^T = U\Lambda(\mathrm{V}^T V)\Lambda\mathrm{U}^T = U\Lambda^2\mathrm{U}^T$$

So we can obtain

$$M^T M v_i = V\Lambda^2\mathrm{V}^T v_i$$

$$= \left[v_1, v_2 \dots v_p\right]\left(\begin{pmatrix} {\sigma_1}^2 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & {\sigma_p}^2 \end{pmatrix}\left(\left[v_1, v_2 \dots v_p\right]^T v_i\right)\right)$$

$$= {\sigma_i}^2 v_i$$

$$MM^T u_i = U\Lambda^2\mathrm{U}^T u_i$$

$$= \left[u_1, u_2 \dots u_p\right]\left(\begin{pmatrix} {\sigma_1}^2 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & {\sigma_p}^2 \end{pmatrix}\left(\left[u_1, u_2 \dots u_p\right]^T u_i\right)\right)$$

$$= {\sigma_i}^2 u_i$$

From above, $M^T M v_i = {\sigma_i}^2 v_i$ and $MM^T u_i = {\sigma_i}^2 u_i$

(d)

From (c),

$$MM^T = U\Lambda^2\mathrm{U}^T = \left[u_1, u_2 \dots u_p\right]\begin{pmatrix} {\sigma_1}^2 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & {\sigma_p}^2 \end{pmatrix}\left[u_1, u_2 \dots u_p\right]^T$$

Therefore, eigenvalues are ${\sigma_1}^2$, ${\sigma_2}^2$,…, ${\sigma_p}^2$ and eigenvectors are columns of U $(u_1, u_2 \dots u_p)$

(e)

From (c),

$$M^T M = V\Lambda^2 \mathrm{V}^T = \left[v_1, v_2 \dots v_p\right] \begin{pmatrix} {\sigma_1}^2 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & {\sigma_p}^2 \end{pmatrix} \left[v_1, v_2 \dots v_p\right]^T$$

Therefore, eigenvalues are ${\sigma_1}^2$, ${\sigma_2}^2$,…, ${\sigma_p}^2$ and eigenvectors are columns of V $(v_1, v_2 \dots v_p)$

We easily notice that the eigenvalues of $M^T M$ and MM^T are the same. As for eigenvectors, eigenvectors of $M^T M$ are columns of V while eigenvectors of MM^T are columns of U, they are related to SVD of M.

(f)

If rank(M)=k, that would mean that

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_k > \sigma_{K+1} = \cdots = \sigma_p = 0$$

There would be k positive singular values and (p-k) zero singular values.

6.

As for low-rank approximation, and in this case, rank=2, we select the top 2 largest singular values, that is

$$\widehat{M} = [u_1, u_2] \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} [v_1, v_2]^T$$

$$= \begin{pmatrix} 10 \\ 00 \\ 00 \\ 01 \end{pmatrix} \begin{pmatrix} 4 & 0 \\ 0 & 3 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \end{pmatrix}$$

Programming Assignment: Image Representations

In this problem, we will study how different representations of images can affect the performance of a nearest neighbor classifier.

We will experiment with CIFAR-10 data set, which has 50,000 training images and 10,000 test images, with ten different classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck). The images are in color, of size 32 × 32.

We will investigate the following represention:

1. Raw pixel space
2. Histogram-of-gradients (HOG) features
3. Convolutional Neural Network (CNN) features
    - Intermediate-level and high-level features extracted from a **pretrained** CNN
    - Intermediate-level and high-level features extracted from a **randomly initialized** CNN

Instructions

1. Install the required libraries
    - Install the CPU version of PyTorch and torchvision
        - A GPU is **not required** for this assigment.
        - For more specific intructions related to your machine, please refer to: https://pytorch.org/get-started/locally/ (https://pytorch.org/get-started/locally/).
    - Install scikit-learn **1.0.2**
        - Note that in version 1.2.2, KNN classifier with L2 distance behaves weirdly. For example, testing training samples will cause an error. Query results vary a lot over different runs. Classification accurary is close to a random guess. We would recommend using version 1.0.2 instead.
    - Install scikit-image, numpy, os, matplotlib, tqdm, torchinfo, etc.
        - Use `pip install` to install any other required library.
2. We have provided you with a number of functions and scripts in the hopes of alleviating some tedious or error-prone sections of the implementation. You are free to modify them if necessary. The provided files include:
    - `dataset.py`
        - Functions used to download, load and visualize a dataset
    - `extract_feature.py`
        - Functions to extract a variety of features, including raw pixels, HoG features and CNN features.
    - `vgg_network.py`
        - Class defining the VGG-11 architecture, functions to load a pretrained VGG model and test it.
    - `path.py`
        - Paths of directories to save the dataset, features, models, and figures. Feel free to reconfigure it.
3. Submission
    - **This notebook includes various things for you to do.**

In [1]:
```
from torchinfo import summary
from vgg_network import vgg11_bn
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
```

1. Load the data

Download the data set

We will be using the CIFAR-10 (https://www.cs.toronto.edu/~kriz/cifar.html) data set for our experiments. Before getting started, run the following block to download the data set. The data set will be saved to directory `/datasets`. You can change the default download path in `path.py`.

In [2]:
```
from dataset import download_cifar10_dataset
dataset = download_cifar10_dataset()
```

```
Dataset Downloading ...
Downloaded CIFAR-10 dataset to  /Users/zhongziyi/datasets
```

Load in the data set

Run the following block to load the training images, training labels, test images, and test labels as `x_train`, `y_train`, `x_test`, `y_test` respectively. The code will print out their shapes for you.

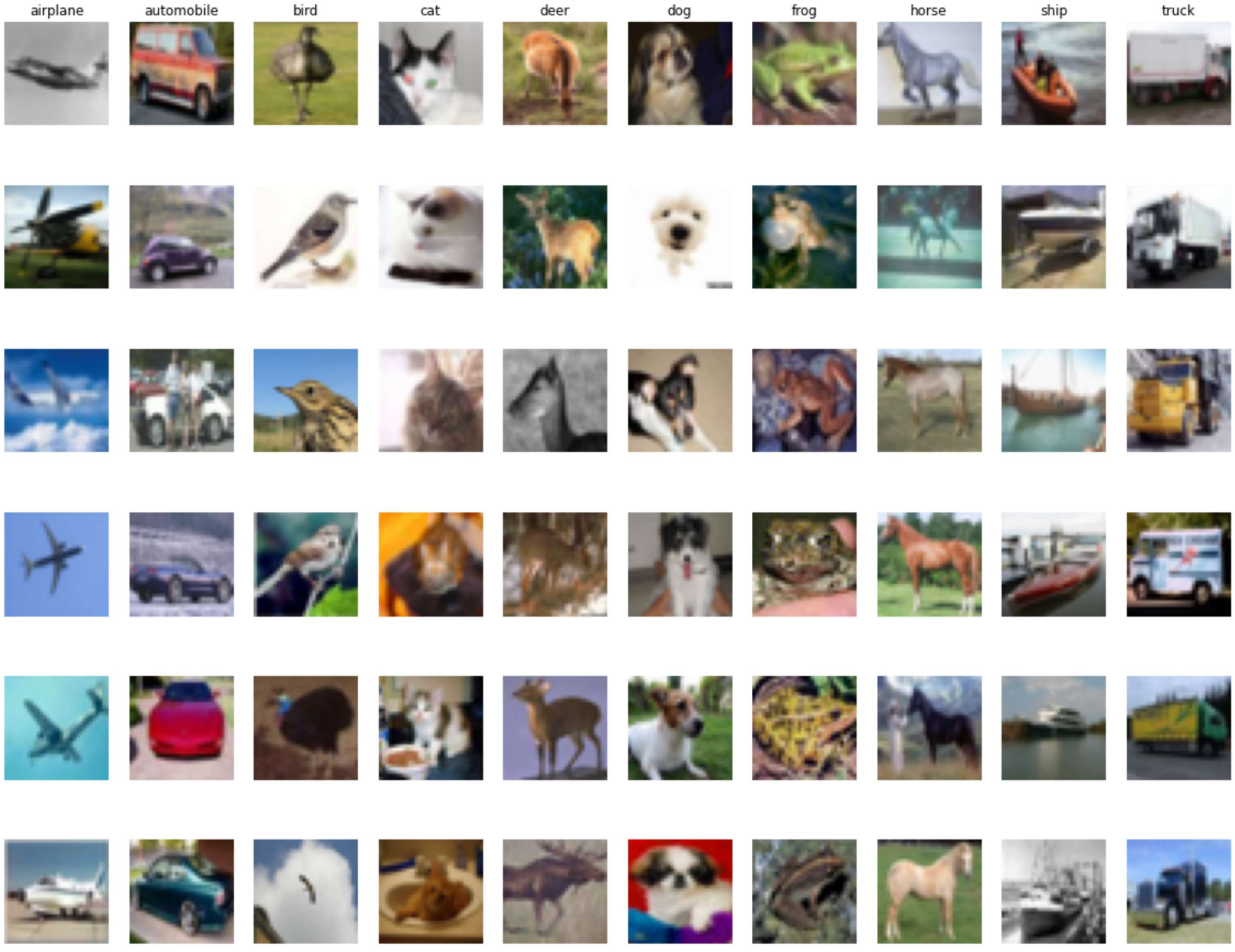
In [3]:
```
from dataset import load_dataset_splits
x_train, y_train, x_test, y_test = load_dataset_splits()
```

```
======> CIFAR-10 dataset loaded
Training set data shape:  (50000, 3, 32, 32)
Training set label shape:  (50000,)
Test set data shape:  (10000, 3, 32, 32)
Test set label shape:  (10000,)
```

Visualize the data

Run the following block to display several randomly-sampled images from each of the ten classes.

In [4]:
```
from dataset import visualize_cifar_data
visualize_cifar_data(images=x_train.transpose(0, 2, 3, 1), labels=y_train, samples_per_class=6)
```



2. Nearest neighbor classification on CIFAR-10

In this part, we will investigate the classification problem by training and testing a nearest neighbor classifier on CIFAR-10 dataset with **Euclidean (L2) distance**.

Function `run_nearest_neighbor`, shown in the next cell, takes a labeled training set (`x_train`, `y_train`) and test set (`x_test`, `y_test`), and applies 1-nearest neighbor classification to the test points, using `KNeighborsClassifier` from `sklearn`. It prints out the accuracy on the test set and returns the classifier.

In [5]:
```
from sklearn.neighbors import KNeighborsClassifier

def run_nearest_neighbor(x_train, y_train, x_test, y_test):
    # create classifier
    nn_classifier = KNeighborsClassifier(n_neighbors=1, algorithm='auto')

    # train
    nn_classifier.fit(x_train, y_train)

    # test and report accuracy
    test_acc = nn_classifier.score(x_test, y_test)
    print("Nearest neighbor accuracy on the test set: %f"%test_acc)

    return nn_classifier
```

Nearest neighbor on raw pixels

Now we do nearest neighbor classification in the raw pixel space.

We have provided you with a function `compute_or_load_features` in `extract_feature.py` to extract the features from the training and test images. You can specify the feature extraction method by setting parameter `feature_type` to one of the options in `['raw_pixel', 'hog', 'pretrained_cnn', 'random_cnn']`. When the features are extracted by a CNN, you can further specify at which layer you are extracting the features

by setting parameter `layer` to one of the options in `['last_conv', 'last_fc']`. In practice, extracting features from a large data set could be time consuming. To avoid repeated computation, this function will only compute the features once and store them in directory `/features` (you can change this in `path.py`). If being called later, the function will search for the existing feature files and directly load them into memory.

Call function `compute_or_load_features` to extract the raw pixels as features and call function `run_knn` to train and test nearest neighbor classifier in this feature space. To do: Report the test accuracy you get.

In [6]:
```
from extract_feature import compute_or_load_features

# compute or load features
raw_pixel_train_features, raw_pixel_test_features = compute_or_load_features(x_train, x_test, "raw_pixel")

# run knn
raw_pixel_knn_classifier = run_nearest_neighbor(raw_pixel_train_features, y_train, raw_pixel_test_features, y_test)
```

```
======> Loaded train and test features from  /Users/zhongziyi/features/raw_pixel.pkl
Training feature shape:  (50000, 3072)
Test feature shape:  (10000, 3072)
Nearest neighbor accuracy on the test set: 0.353900
```

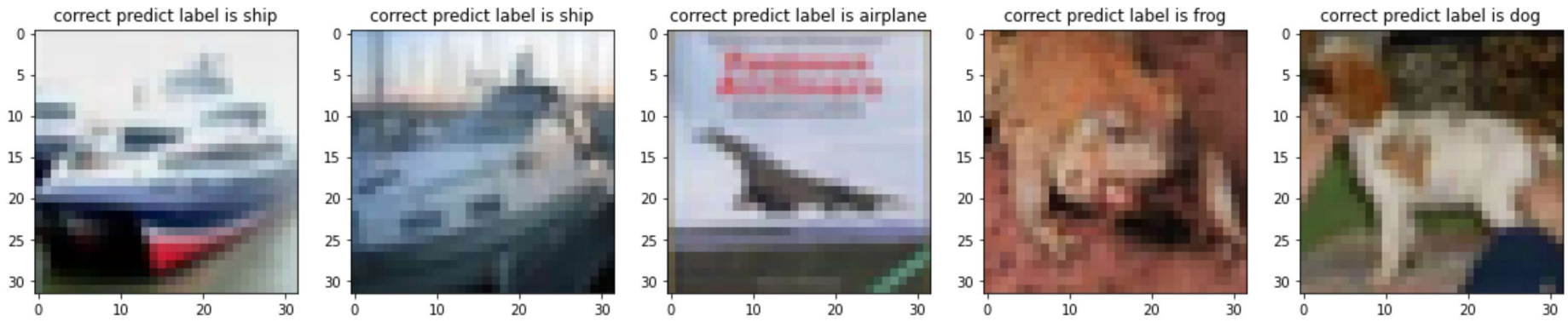
To get a sense of how the images are distributed locally in the pixel space, we can look at their nearest neighbors.

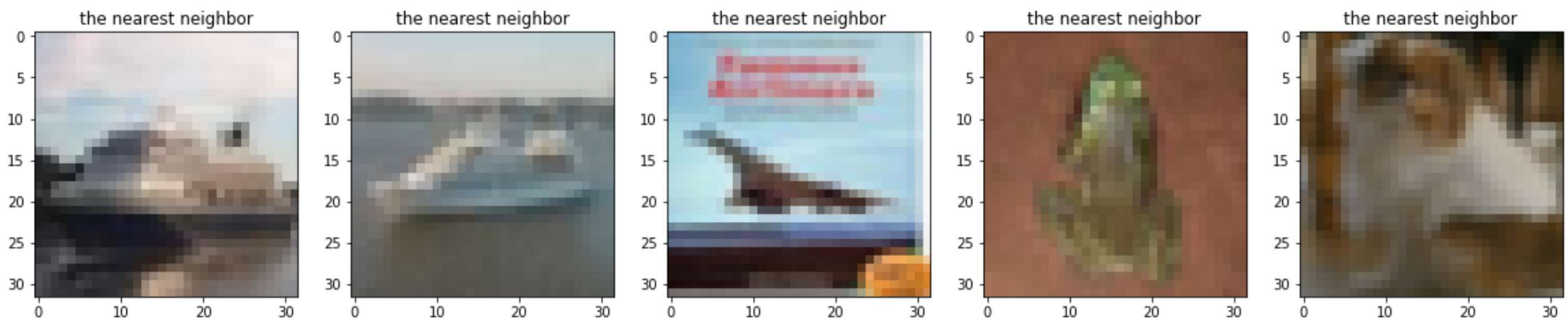
To do: Write code in the following block to do the following. Make your code modular so that it can be re-used for the representations we will consider.

- Show the first five images in the test set whose label is correctly predicted by 1-NN, and show the nearest neighbor (in the training set) of each of these images.
- Show the first five images in the test set whose label is incorrectly predicted by 1-NN, and show the nearest neighbor (in the training set) of each of the images.

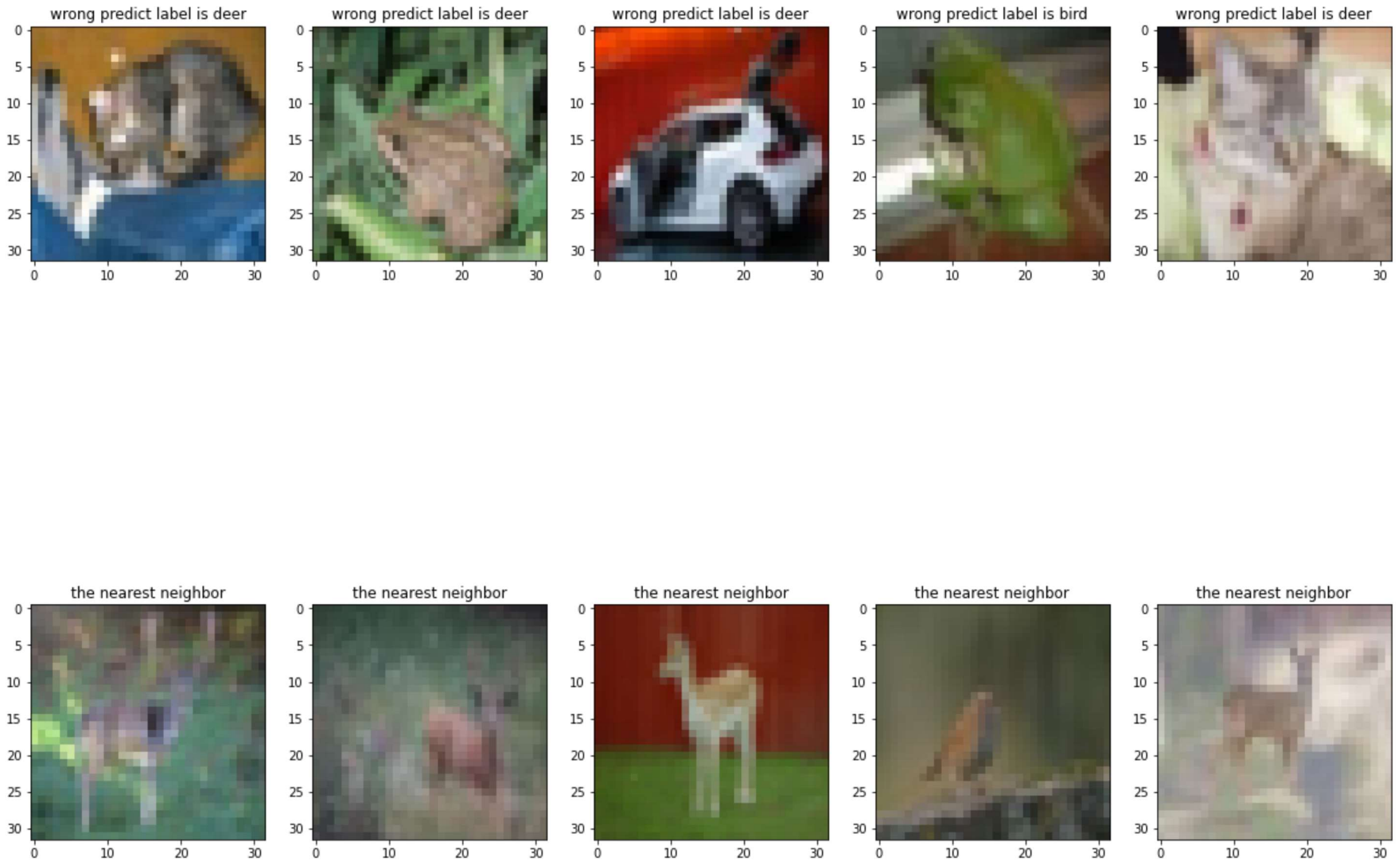
In [7]:
```
### Your code here
import numpy as np
import matplotlib.pyplot as plt
cifar_classes = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
def correct(model,train_features,test_features,raw_train_features,raw_test_features,cifar_classes):
    cnt=0
    x_correct,y_correct=[],[]
    y_pred=model.predict(test_features)
    for i in range(len(y_pred)):
        if y_pred[i]==y_test[i]:
            cnt+=1
            x_correct.append(i)
            y_correct.append(y_pred[i])

        if cnt==5:
            break


    x_correct=np.array(x_correct)
    y_correct=np.array(y_correct)
    correct_neighbors=model.kneighbors(test_features[x_correct,:],return_distance=False)
    correct_neighbors_t=raw_train_features[correct_neighbors,:]
    correct_neighbors_fig=raw_test_features[x_correct,:]
    for i in range(5):
        plt.subplot(2,5,i+1)
        plt.title('correct predict label is {}'.format(cifar_classes[y_pred[x_correct[i]]]))
        fig=correct_neighbors_fig[i].reshape(3,32,32)
        fig=fig.transpose(1,2,0)
        plt.imshow(fig)
        plt.subplot(2,5,i+6)
        plt.title('the nearest neighbor')
        fig=correct_neighbors_t[i].reshape(3,32,32)
        fig=fig.transpose(1,2,0)
        plt.imshow(fig)


def wrong(model,train_features,test_features,raw_train_features,raw_test_features,cifar_classes):
    cnt=0
    x_wrong,y_wrong=[],[]
    y_pred=model.predict(test_features)
    for i in range(len(y_pred)):
        if y_pred[i]!=y_test[i]:
            cnt+=1
            x_wrong.append(i)
            y_wrong.append(y_pred[i])
            #plt.imshow(x_test[i].transpose())
        if cnt==5:
            break


    x_wrong=np.array(x_wrong)
    y_wrong=np.array(y_wrong)
    wrong_neighbors=model.kneighbors(test_features[x_wrong,:],return_distance=False)
    wrong_neighbors_t=raw_train_features[wrong_neighbors,:]
    wrong_neighbors_fig=raw_test_features[x_wrong,:]
    for i in range(5):
        plt.subplot(2,5,i+1)
        plt.title('wrong predict label is {}'.format(cifar_classes[y_pred[x_wrong[i]]]))
        fig=wrong_neighbors_fig[i].reshape(3,32,32)
        fig=fig.transpose(1,2,0)
        plt.imshow(fig)
        plt.subplot(2,5,i+6)
        plt.title('the nearest neighbor')
        fig=wrong_neighbors_t[i].reshape(3,32,32)
        fig=fig.transpose(1,2,0)
        plt.imshow(fig)
```

In [8]:
```
correct(raw_pixel_knn_classifier,raw_pixel_train_features,raw_pixel_test_features,
        raw_pixel_train_features,raw_pixel_test_features,cifar_classes)
```





In [9]:
```
wrong(raw_pixel_knn_classifier,raw_pixel_train_features,raw_pixel_test_features,
      raw_pixel_train_features,raw_pixel_test_features,cifar_classes)
```



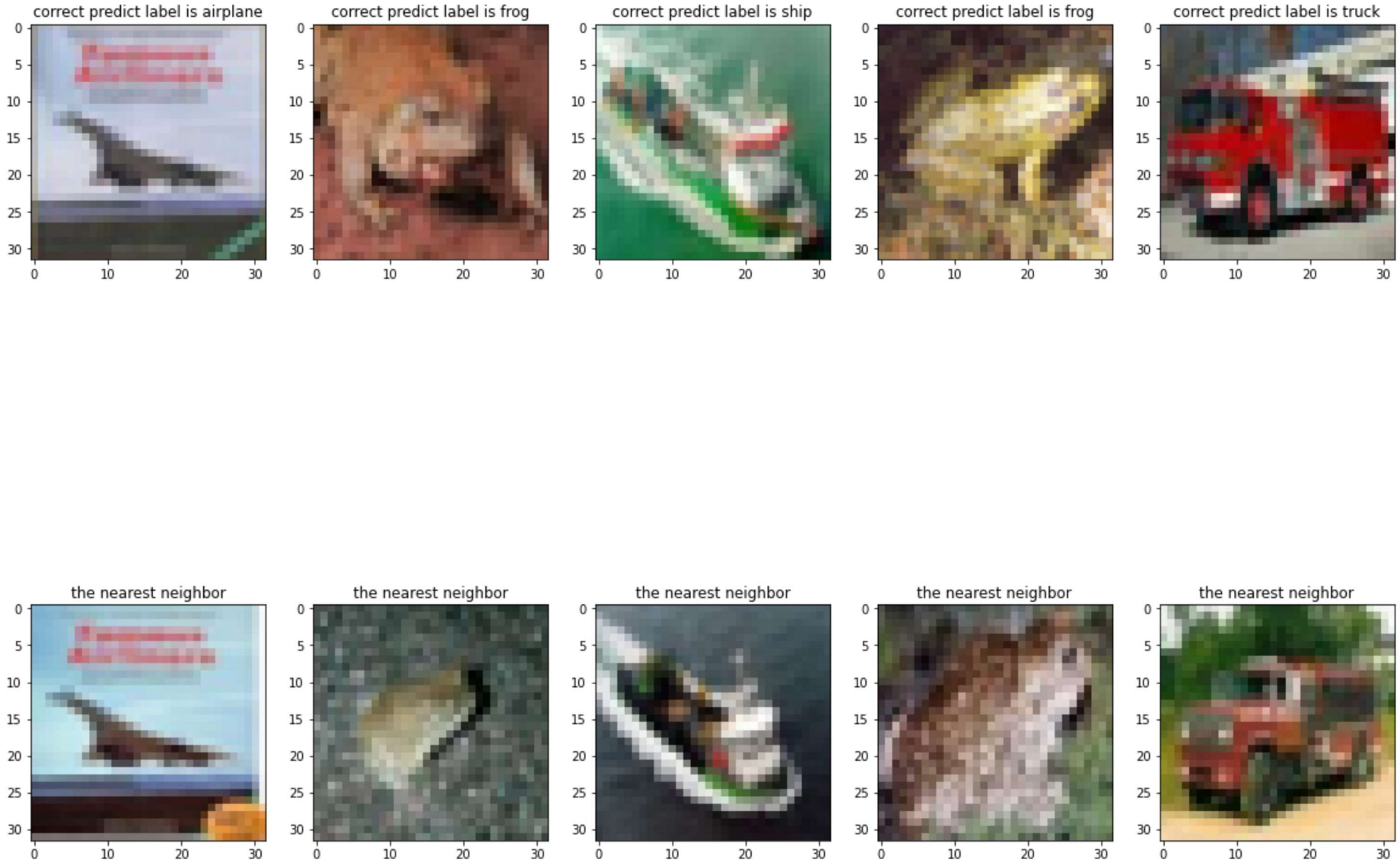
Nearest neighbor on HOG features

The HOG (Histogram of Oriented Gradients) descriptor computes local statistics of gradients in an image and uses them as feature respresentations. Train and test a nearest neighbor classifier on HOG features. To do: Report the test accuracy. As with the raw pixel representation, show the first five correctly classified images (and their nearest neighbors) and the five first incorrectly classified.

In [10]:
```
from extract_feature import compute_or_load_features

# compute or load features
hog_train_features, hog_test_features = compute_or_load_features(x_train, x_test, "hog")

# run knn
hog_knn_classifier = run_nearest_neighbor(hog_train_features, y_train, hog_test_features, y_test)
```

```
======> Loaded train and test features from  /Users/zhongziyi/features/hog.pkl
Training feature shape:  (50000, 512)
Test feature shape:  (10000, 512)
Nearest neighbor accuracy on the test set: 0.365700
```

In [11]:
```
correct(hog_knn_classifier,hog_train_features,hog_test_features,
        raw_pixel_train_features,raw_pixel_test_features,cifar_classes)
```



In [12]:
```
wrong(hog_knn_classifier,hog_train_features,hog_test_features,
    raw_pixel_train_features,raw_pixel_test_features,cifar_classes)
```

Nearest neighbor on CNN features

Over the past decade, deep *convolutional neural networks* (CNNs) have become building blocks in a wide range of computer vision tasks. A CNN trained on a large scale image classification task learns to extract spatial hierarchies of features from edges to object parts [2].

In this problem, we will explore the representations learned by CNNs on different layers. We will be using VGG (https://arxiv.org/abs/1409.1556) as our convolutional network architecture. A VGG11 model pretrained on CIFAR-10 can be found at `/models/vgg11_bn.pt` , so you are **NOT REQUIRED** to train the neural network. And since we only need to do forward pass through the network (no backpropagation), a CPU is enough for our purpose.

A VGG network is composed of a sequence of convolutional layers, pooling layers, and fully connected layers. To clearly understand its architecture, we provide you with a helper function visualizing the layers and input/output shapes. The following code feeds a batch of 32 × 32 × 3 images into VGG11 network and do a forward pass. Run the code and check the summary.

In [13]:
```
from torchinfo import summary
from vgg_network import vgg11_bn
vgg_model = vgg11_bn(pretrained=False)
summary(vgg_model, input_size=(16, 3, 32, 32))
```

Out[13]:
```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
VGG                                      [16, 10]                  --
├─Sequential: 1-1                        [16, 512, 1, 1]           --
│    └─Conv2d: 2-1                       [16, 64, 32, 32]          1,792
│    └─BatchNorm2d: 2-2                  [16, 64, 32, 32]          128
│    └─ReLU: 2-3                         [16, 64, 32, 32]          --
│    └─MaxPool2d: 2-4                    [16, 64, 16, 16]          --
│    └─Conv2d: 2-5                       [16, 128, 16, 16]         73,856
│    └─BatchNorm2d: 2-6                  [16, 128, 16, 16]         256
│    └─ReLU: 2-7                         [16, 128, 16, 16]         --
│    └─MaxPool2d: 2-8                    [16, 128, 8, 8]           --
│    └─Conv2d: 2-9                       [16, 256, 8, 8]           295,168
│    └─BatchNorm2d: 2-10                 [16, 256, 8, 8]           512
│    └─ReLU: 2-11                        [16, 256, 8, 8]           --
│    └─Conv2d: 2-12                      [16, 256, 8, 8]           590,080
│    └─BatchNorm2d: 2-13                 [16, 256, 8, 8]           512
│    └─ReLU: 2-14                        [16, 256, 8, 8]           --
│    └─MaxPool2d: 2-15                   [16, 256, 4, 4]           --
│    └─Conv2d: 2-16                      [16, 512, 4, 4]           1,180,160
│    └─BatchNorm2d: 2-17                 [16, 512, 4, 4]           1,024
│    └─ReLU: 2-18                        [16, 512, 4, 4]           --
│    └─Conv2d: 2-19                      [16, 512, 4, 4]           2,359,808
│    └─BatchNorm2d: 2-20                 [16, 512, 4, 4]           1,024
│    └─ReLU: 2-21                        [16, 512, 4, 4]           --
│    └─MaxPool2d: 2-22                   [16, 512, 2, 2]           --
│    └─Conv2d: 2-23                      [16, 512, 2, 2]           2,359,808
│    └─BatchNorm2d: 2-24                 [16, 512, 2, 2]           1,024
│    └─ReLU: 2-25                        [16, 512, 2, 2]           --
│    └─Conv2d: 2-26                      [16, 512, 2, 2]           2,359,808
│    └─BatchNorm2d: 2-27                 [16, 512, 2, 2]           1,024
│    └─ReLU: 2-28                        [16, 512, 2, 2]           --
│    └─MaxPool2d: 2-29                   [16, 512, 1, 1]           --
├─AdaptiveAvgPool2d: 1-2                 [16, 512, 1, 1]           --
├─Sequential: 1-3                        [16, 10]                  --
│    └─Linear: 2-30                      [16, 4096]                2,101,248
│    └─ReLU: 2-31                        [16, 4096]                --
│    └─Dropout: 2-32                     [16, 4096]                --
│    └─Linear: 2-33                      [16, 4096]                16,781,312
│    └─ReLU: 2-34                        [16, 4096]                --
│    └─Dropout: 2-35                     [16, 4096]                --
│    └─Linear: 2-36                      [16, 10]                  40,970
==========================================================================================
Total params: 28,149,514
Trainable params: 28,149,514
Non-trainable params: 0
Total mult-adds (G): 2.75
==========================================================================================
Input size (MB): 0.20
Forward/backward pass size (MB): 39.85
Params size (MB): 112.60
Estimated Total Size (MB): 152.64
==========================================================================================
```

Nearest neighbor on pretrained CNN features

As shown in the summary above, a VGG net consists of three components:

1. A sequence of convolutional blocks (1-1 and 1-2)
2. Two fully connected blocks (2-30 to 2-35)
3. A single fully connected layer (2-36)

The first two components together act as a feature extractor. The convolutional blocks extract the low-level and intermediate-level features, and the fully connected blocks extract the high-level features. The third component is a single linear layer mapping the feature vectors to the classes therefore can be viewed as a linear classifier.

In this problem, we will experiment with features extracted by a pretrained VGG net at two specific layers:

1. `last_conv` : AdaptiveAvgPool2d (1-2)
2. `last_fc` : ReLU (2-34)

Train and test a nearest neighbor classifier on pretrained VGG features at these two layers respectively (Set parameter `layer` of function `compute_or_load_features` to `last_conv` or `last_fc`). Report the test accuracies with these two representations. For `last_fc` , show the first five correctly classified images (with nearest neighbors) and the first five incorrectly classified.

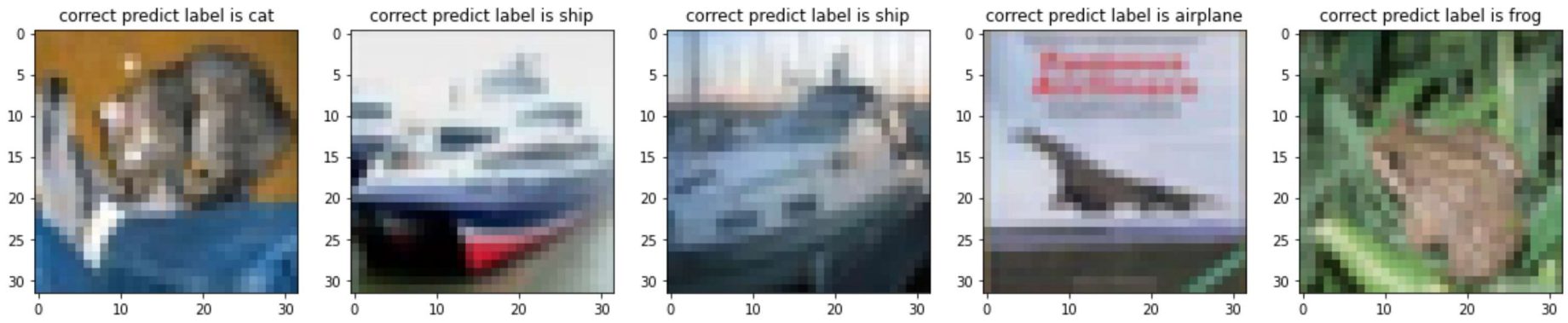
In [14]:
```
from extract_feature import compute_or_load_features

# compute or load features
pretrained_cnn_last_conv_train_features, pretrained_cnn_last_conv_test_features = compute_or_load_features(x_train, x_test, "pretraine

# run knn
pretrained_cnn_last_conv_knn_classifier = run_nearest_neighbor(pretrained_cnn_last_conv_train_features, y_train, pretrained_cnn_last_c
```

```
======> Loaded train and test features from  /Users/zhongziyi/features/pretrained_cnn_last_conv.pkl
Training feature shape:  (50000, 512)
Test feature shape:  (10000, 512)
Nearest neighbor accuracy on the test set: 0.920000
```

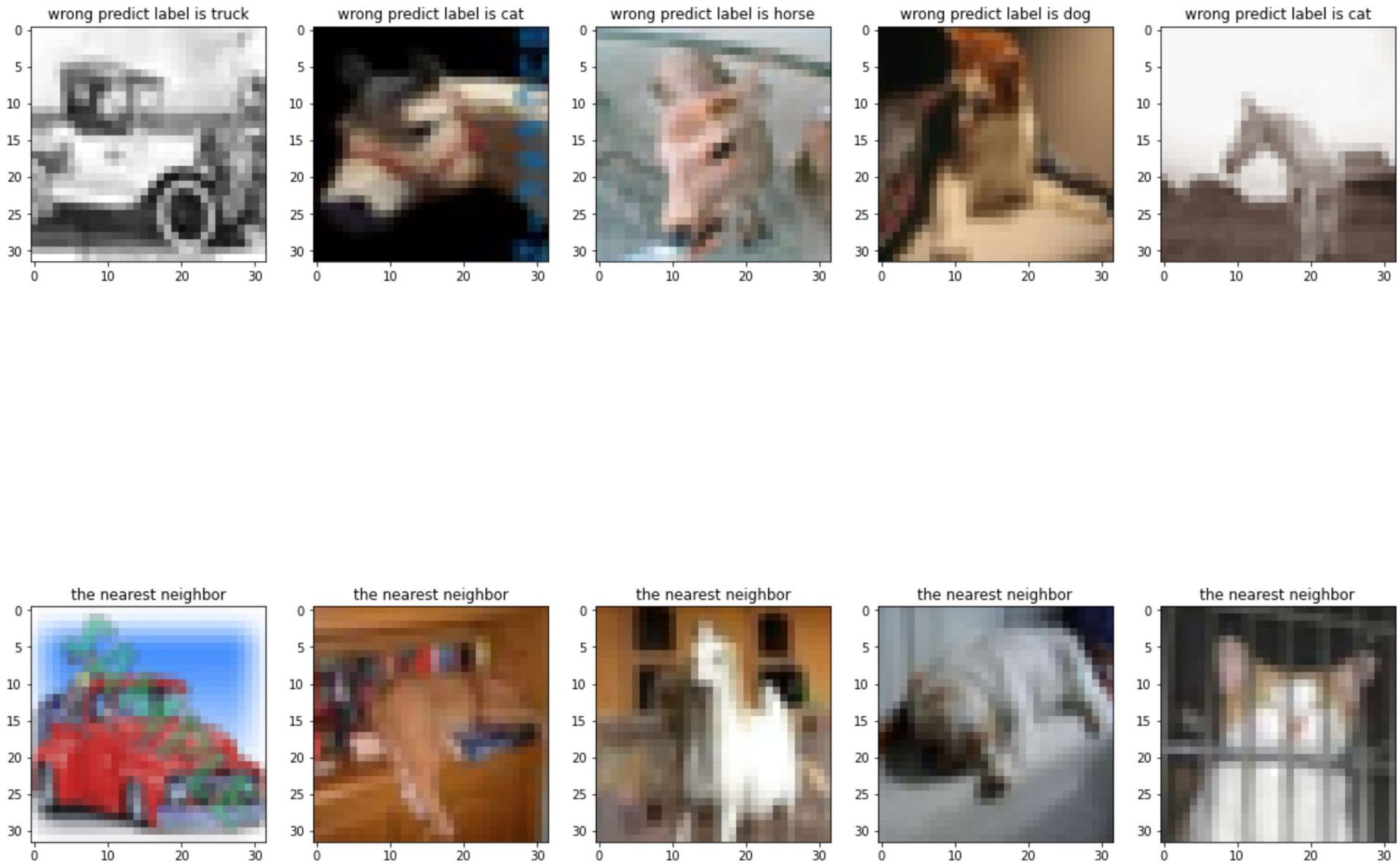
In [15]:
```
from extract_feature import compute_or_load_features

# compute or load features
pretrained_cnn_last_fc_train_features, pretrained_cnn_last_fc_test_features = compute_or_load_features(x_train, x_test, "pretrained_cn

# run knn
pretrained_cnn_last_fc_knn_classifier = run_nearest_neighbor(pretrained_cnn_last_fc_train_features, y_train, pretrained_cnn_last_fc_te
```

```
======> Loaded train and test features from  /Users/zhongziyi/features/pretrained_cnn_last_fc.pkl
Training feature shape:  (50000, 4096)
Test feature shape:  (10000, 4096)
Nearest neighbor accuracy on the test set: 0.920700
```

In [16]:
```
correct(pretrained_cnn_last_fc_knn_classifier,pretrained_cnn_last_fc_train_features,
        pretrained_cnn_last_fc_test_features,
        raw_pixel_train_features,raw_pixel_test_features,cifar_classes)
```



In [17]:
```
wrong(pretrained_cnn_last_fc_knn_classifier,pretrained_cnn_last_fc_train_features,
      pretrained_cnn_last_fc_test_features,
      raw_pixel_train_features,raw_pixel_test_features,cifar_classes)
```



Alignment between nearest neighbor classifier and pretrained CNN model

In the following block, call function `test_pretrained_vgg` from `vgg_network.py`. The function will test the pretrained VGG net on the test images of CIFAR-10 and return the test accuracy. Report this accuracy and compare it with the test accuracy of nearest neighbor classifier in pretrained `last conv` and `last fc` feature space. In which space does the nearest neighbor classifier behave the most similarly to a pretrained CNN model?

In [18]:
```
from vgg_network import test_pretrained_vgg

test_acc = test_pretrained_vgg(x_test, y_test)
print('Accuracy on the test images: ', test_acc)
```

```
Accuracy on the test images:  0.9225
```

Nearest neighbor on random CNN features

The excellent feature extraction ability of ConvNets enables them to solve computer vision problems in a universal way. One may think that this is because ConvNets are generally trained on large datasets of images. But is this the only reason? To answer this question, a recent work Deep Image Prior [1] shows that the structure of a CNN is sufficient to capture a great deal of low-level image statistics prior to any learning. A randomly initialized fully-convolutional neural network is already able to achieve good results in standard image reconstruction problems such as denoising, super-resolution, and inpainting.

In this problem, we will investigate whether the inductive bias captured by the ConvNets also helps for image classification problems.

You will train and test the nearest neighbor classifier on the features extracted by a randomly initialized VGG network. To do: Experiment with two feature spaces `last conv` and `last fc`, and report the test accuracies.

In [19]:
```
from extract_feature import compute_or_load_features

# compute or load features
random_cnn_last_conv_train_features, random_cnn_last_conv_test_features = compute_or_load_features(x_train, x_test, "random_cnn", "las

# run knn
random_cnn_last_conv_knn_classifier = run_nearest_neighbor(random_cnn_last_conv_train_features, y_train, random_cnn_last_conv_test_fea
```

```
======> Loaded train and test features from  /Users/zhongziyi/features/random_cnn_last_conv.pkl
Training feature shape:  (50000, 512)
Test feature shape:  (10000, 512)
Nearest neighbor accuracy on the test set: 0.404700
```

In [20]:
```
from extract_feature import compute_or_load_features

# compute or load features
random_cnn_last_fc_train_features, random_cnn_last_fc_test_features = compute_or_load_features(x_train, x_test, "random_cnn", "last_fc

# run knn
random_cnn_last_fc_knn_classifier = run_nearest_neighbor(random_cnn_last_fc_train_features, y_train, random_cnn_last_fc_test_features,
```

```
======> Loaded train and test features from  /Users/zhongziyi/features/random_cnn_last_fc.pkl
Training feature shape:  (50000, 4096)
Test feature shape:  (10000, 4096)
Nearest neighbor accuracy on the test set: 0.393400
```

Compare the test accuracies of nearest neighbor classifier on random `last conv` features and HOG features. In which representation space does it perform better?

---

my answer is the last conv is better since accuracy on test set for HOG is only 0.365700 while the last conv is 0.4047. But 0.4 and 0.37 are close, so basically they are not very different since they both are not too good(under 0.5)

Compare the test accuracies of nearest neighbor classifier on random `last conv` features and random `last fc` features. In which representation space does it perform better?

---

my answer is that the last conv is a litte better since last conv got 0.4047 on test set while last fc got 0.3934. To be honest, no much difference since they are really close.

References

1. D. Ulyanov, A. Vedaldi, and V. Lempitsky, Deep Image Prior (https://arxiv.org/pdf/1711.10925.pdf), CVPR 2018.
2. M. Zeiler and R. Fergus, Visualizing and Understanding Deep Neural Networks (https://arxiv.org/pdf/1311.2901.pdf), ECCV 2014.