Xinxin Wu

V00950236

CSC421_Assignment1

Background

For all three algorithms, download and read the data set of CIFAR-10. CIFAR's data dictionary contains 50000 images, each image is a three-channel color(red, green, black) image of 32x32.

So the training set of CIFAR-10 is composed of 50000 vectors of 32x32x3=3072. The matrix of X_train(50000,3072) constitutes the training picture, and the training set contains 50000 labels. The test set(X_test) is 10,000 images, having 10,000 labels.

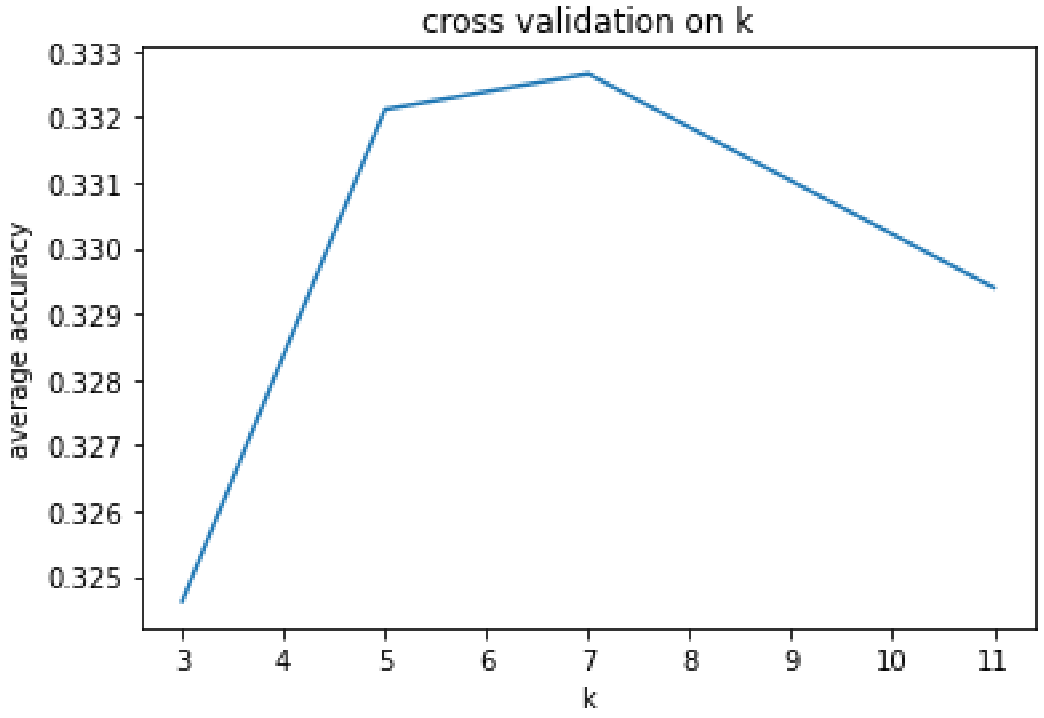
Implementation and result of KNN:

```
import torch
import torchvision
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from torchvision.transforms import ToTensor, Compose
import matplotlib.pyplot as plt
import numpy as np

def load_CIFAR10():
    train_dataset = datasets.CIFAR10(root='data/', download=True,
train=True, transform=ToTensor())
    test_dataset = datasets.CIFAR10(root='data/', download=True,
train=False, transform=ToTensor())

    np.set_printoptions(threshold=np.inf)

    X_train = train_dataset.data
    X_test = test_dataset.data

    y_train = np.array(train_dataset.targets)
    y_test = np.array(test_dataset.targets)

    # X_train = np.reshape(X_train, (X_train.shape[0], -1)).T
    # X_test = np.reshape(X_test, (X_test.shape[0], -1)).T
    X_train = X_train.reshape((50000, 32 * 32 * 3)).T
    X_test = X_test.reshape((10000, 32 * 32 * 3)).T

    y_train = y_train.reshape(1, -1)
    y_test = y_test.reshape(1, -1)

    #X_train = X_train[:, :10000]
    #X_test = X_test[:, :1000]

    #y_train = y_train[:10000].reshape(1, -1)
    #y_test = y_test[:1000].reshape(1, -1)

    X_train = X_train.astype("float")
    X_test = X_test.astype("float")

    return X_train, y_train, X_test, y_test

def sorted_distance(X_train, X_test):
    distances = np.zeros((X_test.shape[1], X_train.shape[1]))

    # (X_test - X_train)*(X_test - X_train)
    # = -2X_test*X_train + X_test*X_test + X_train*X_train
    scalar = -2
    d1 = np.multiply(np.dot(X_test.T, X_train), scalar)  # -2 * X_test
* X_train
    d2 = np.sum(np.square(X_train), axis=0, keepdims=True)  # X_train
* X_train
    d3 = np.sum(np.square(X_test.T), axis=1, keepdims=True)  # X_test
* X_test

    distances = np.sqrt(d1 + d2 + d3)

    for i in range(X_test.shape[1]):  # sort the distances row by row
        distances[i] = np.argsort(distances[i])

    return distances

def predict(X_test, X_train, Y_train, k, distances):
    Y_prediction = np.zeros(X_test.shape[1])

    for i in range(X_test.shape[1]):
        nearest_k = distances[i][:k]  # select k points which have the
nearest distance
        nearest_k = nearest_k.astype(np.int64)

        classes_of_k = Y_train[0, nearest_k]  # find the classes of
each k points
        Y_prediction[i] = np.argmax(np.bincount(classes_of_k))  #
return the class with highest frequency
```

```
    return Y_prediction

def knn(X_test, Y_test, X_train, Y_train, k, distances,
verbose=False):
    numOftest = Y_test.shape[1]

    Y_prediction = predict(X_test, X_train, Y_train, k, distances)
    num_correct = np.sum(Y_prediction == Y_test)
    accuracy = num_correct / numOftest

    if verbose:
        print('Correct %d/%d: The test accuracy of best k is: %f' %
(num_correct, numOftest, accuracy))
    return accuracy

def cross_validation(X_train, y_train, k_set):
    print("-------------------cross validation
starts-------------------")
    num_folds = 5    # k-fold value = 5
    best_accuracy = -1
    best_k = 0

    # divide both X_train and y_train into 5 parts,
    # 4 of them are train set and the left one is the test set
    X_train_folds = np.array_split(X_train.T, num_folds)
    y_train_folds = np.array_split(y_train.T, num_folds)

    accuracys = np.zeros(len(k_set), dtype=np.float64)

    # Take 4 of them for training, 1 of them for verification by using
loop
    for i in range(num_folds):
        X_tr = np.reshape(np.array(X_train_folds[:i] + X_train_folds[i
+ 1:]),
                          (int(X_train.shape[1] * (num_folds - 1) /
num_folds), -1))
        y_tr = np.reshape(y_train_folds[:i] + y_train_folds[i + 1:],
                          (int(y_train.shape[1] * (num_folds - 1) /
num_folds), -1))
        X_te = X_train_folds[i]
        y_te = y_train_folds[i]

        distances = sorted_distance(X_tr.T, X_te.T)

        for j in range(len(k_set)):
            accuracys[j] += knn(X_te.T, y_te.T, X_tr.T, y_tr.T,
k_set[j], distances, verbose=False)
            # find the best k
            if(accuracys[j] > best_accuracy):
                best_accuracy = accuracys[j]
                best_k = k_set[j]
    # Take the average of all 5 verification results as the accuracy
    accuracys /= num_folds


    for j in range(len(k_set)):
        print("k =", k_set[j])
        print("the average accuracy of %d train folds is %f" %
(num_folds, accuracys[j]))
        print('\n' +
"--------------------------------------------------------" + '\n')

    # plot the image to show the relationship between k and average
accuracy
    plt.plot(k_set, accuracys)
    plt.ylabel('average accuracy')
    plt.xlabel('k')
    plt.title("cross validation on k")
    plt.show()
    print("The best k with highest average accuracy is", best_k)
    return best_k

def test_data_validation(X_test, y_test, X_train, y_train, k):
    print("\n-------------------test data validation
starts-------------------")

    # Get the sorted_distance, avoid redundant calculation
    distances = sorted_distance(X_train, X_test)
    print()
    print("The best k we get from the cross validation is", k)
    accuracy = knn(X_test, y_test, X_train, y_train, k, distances,
verbose=True)

X_train, y_train, X_test, y_test = load_CIFAR10()
k_set = [3, 5, 7, 11]

best_k = cross_validation(X_train, y_train, k_set)
test_data_validation(X_test, y_test, X_train, y_train, best_k)
```

```
Files already downloaded and verified
Files already downloaded and verified
-------------------cross validation starts-------------------
k = 3
the average accuracy of 5 train folds is 0.324620

-------------------------------------------------------

k = 5
the average accuracy of 5 train folds is 0.332120
```

```
-------------------------------------------------------

k = 7
the average accuracy of 5 train folds is 0.332660

-------------------------------------------------------

k = 11
the average accuracy of 5 train folds is 0.329400

-------------------------------------------------------
```



```
The best k with highest average accuracy is 7

-------------------test data validation starts------------------

The best k we get from the cross validation is 7
Correct 3358/10000: The test accuracy of best k is: 0.335800
```

Report of KNN:

Divide the training set into five batches, of which four-fifths were taken as the training set and the remaining one-fifth as the test set. The average accuracy was obtained by substituting K =3, 5, 7 and 11 into the calculation respectively. Select the k value with the highest accuracy as k of X_test.

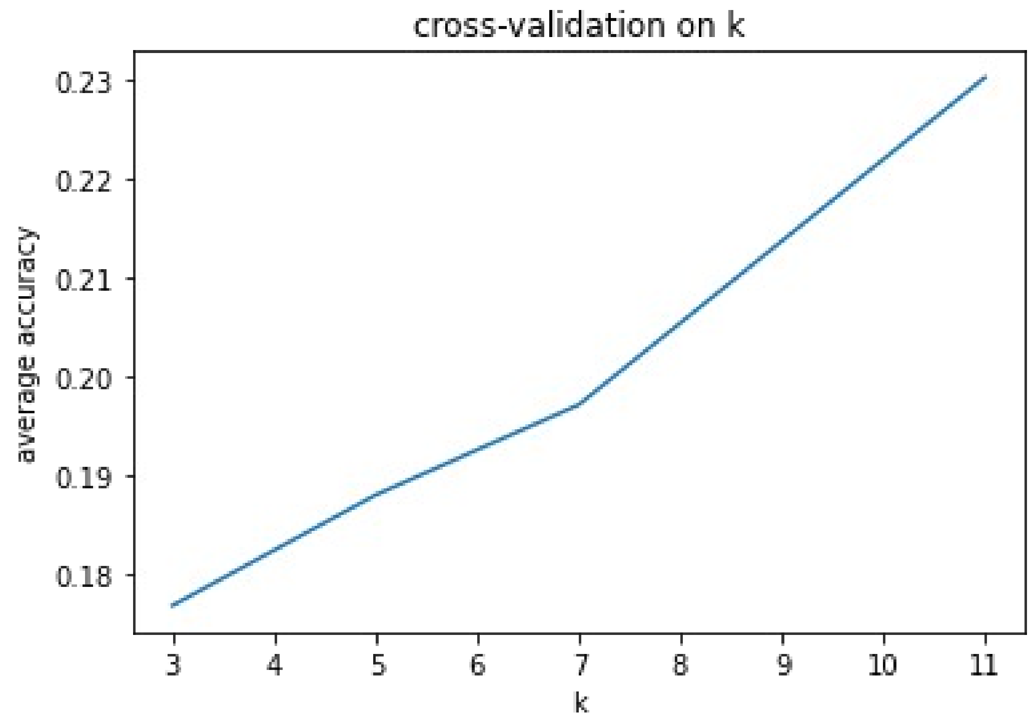
The core part of the algorithm is as follows: calculate the distance between the images to be classified and the images of the training set through Euclidean distance using matrix, sort the distance from small to large, and find the first K labels with the most occurrences as the prediction classification labels. Then all the predicted classification labels are compared with the real classification labels in the test set to calculate the accuracy.

Because the Euclidean distance is calculated using a matrix method rather than loops, the algorithm is much faster and more efficient(just a few minutes). As we can see from the output, when this method is applied, all k are about 32% accurate. When K=7, the average prediction accuracy is the highest. Therefore, K=7 will be regarded as the best k value to predict X_test, and finally we got about 33.58% accuracy.

Implementation and result of K Means:

```
import torch
import torchvision
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from torchvision.transforms import ToTensor, Compose
import matplotlib.pyplot as plt
import numpy as np

def load_CIFAR10():
    train_dataset = datasets.CIFAR10(root='data/', download=True,
train=True, transform=ToTensor())
    test_dataset = datasets.CIFAR10(root='data/', download=True,
train=False, transform=ToTensor())

    np.set_printoptions(threshold=np.inf)

    X_train = train_dataset.data
    X_test = test_dataset.data

    y_train = np.array(train_dataset.targets)
    y_test = np.array(test_dataset.targets)

    X_train = X_train.reshape(50000, 32 * 32 * 3)
    X_test = X_test.reshape(10000, 32 * 32 * 3)

    '''
    X_train = X_train[:10000]
    X_test = X_test[:1000]

    y_train = y_train[:10000]
    y_test = y_test[:1000]
    '''
    X_train = X_train.astype("float")
    X_test = X_test.astype("float")

    return X_train, y_train, X_test, y_test

def distance(data_set, cores, k):
    distances = np.zeros((data_set.shape[0], k))

    d1 = np.multiply(np.dot(data_set, cores.T), -2)
    d2 = np.sum(np.square(cores.T), axis=0, keepdims=True)
    d3 = np.sum(np.square(data_set), axis=1, keepdims=True)

    distances = np.sqrt(d1 + d2 + d3)
    return distances

def k_means(X_train, y_train, k):
    # m = the number of train set
    # n = the number of all pixels of the image in train set
    m, n = X_train.shape

    # Clustering results for m samples,
    # each of which corresponds to a clustering ordinal number
    result = np.empty(m, dtype=np.int32)

    # Random initialization of k centroids
    cores = X_train[np.random.choice(np.arange(m), k, replace=False)]

    # k labels corresponding to the K clusters
    label_of_clusters = np.empty(k, dtype=np.int32)

    while True:
        distances = distance(X_train, cores, k)

        # The nearest centroid index ordinal number for each sample
        index_min = np.argmin(distances, axis=1)

        # If the sample clustering does not change, return cluster
centroid data and cluster labels
        if (index_min == result).all():
            for i in range(k):
                label_of_clusters[i] =
np.argmax(np.bincount(y_train[result == i]))
            break

        # Reclassify
        result[:] = index_min
        # Traverse the centroid set
        for i in range(k):
            # Find the subset corresponding to the current centroid
            items = X_train[result == i]
            # Takes the mean of the subset as the position of the
current centroid
            cores[i] = np.mean(items, axis=0)

    return cores, label_of_clusters

def predict(X_test, cores, label_of_clusters):
    k = label_of_clusters.shape[0]
    m, n = X_test.shape
    # m predictive labels for test samples
    prediction = np.zeros(m)

    distances = distance(X_test, cores, k)

    # The nearest centroid index ordinal number for each sample
    index_min = np.argmin(distances, axis=1)
    prediction = label_of_clusters[index_min]

    return prediction

def calculate_accuracy(y_test, prediction, verbose=False):
    num_test = y_test.shape[0]
    num_correct = np.sum(prediction == y_test)
    accuracy = num_correct / num_test
    if verbose:
        print('Correct %d/%d: The test accuracy: %f' % (num_correct,
num_test, accuracy))
    return accuracy

def cross_validation(X_train, y_train, k_set):
    print("------------------cross validation
starts------------------")
    num_folds = 5    # k-fold value = 5

    best_accuracy = -1
    best_k = 0

    # divide both X_train and y_train into 5 parts,
    # 4 of them are train set and the left one is the test set
    X_train_folds = np.array_split(X_train, num_folds)
    y_train_folds = np.array_split(y_train, num_folds)

    accuracys = np.zeros(len(k_set), dtype=np.float64)

    # Take 4 of them for training, 1 of them for verification by using
loop
    for i in range(num_folds):
        X_tr = np.reshape(np.array(X_train_folds[:i] + X_train_folds[i
+ 1:]),
                          (int(X_train.shape[0] * (num_folds - 1) /
num_folds), -1))
        y_tr = np.reshape(y_train_folds[:i] + y_train_folds[i + 1:],
                          int(y_train.shape[0] * (num_folds - 1) /
num_folds))
        X_te = X_train_folds[i]
        y_te = y_train_folds[i]

        for j in range(len(k_set)):
            cores, label_of_clusters = k_means(X_tr, y_tr, k_set[j])
            y_prediction = predict(X_te, cores, label_of_clusters)
            accuracys[j] += calculate_accuracy(y_te, y_prediction,
verbose=False)

            if(accuracys[j] > best_accuracy):
                best_accuracy = accuracys[j]
                best_k = k_set[j]

    accuracys /= num_folds
    for j in range(len(k_set)):
        print("k =", k_set[j])
        print("the average accuracy of %d train folds is %f" %
(num_folds, accuracys[j]))
        print('\n' +
"--------------------------------------------------------" + '\n')
    plt.plot(k_set, accuracys)
    plt.ylabel('average accuracy')
    plt.xlabel('k')
    plt.title("cross-validation on k")
    plt.show()

    print("The best k with highest average accuracy is", best_k)
    print()
    return best_k

def test_data_validation(X_test, y_test, X_train, y_train, k):
    print("-------------------test data validation
starts-------------------")

    print()
    print("The best k we get from the cross validation is", k)

    cores, label_of_clusters = k_means(X_train, y_train, k)
    y_prediction = predict(X_test, cores, label_of_clusters)
    accuracy = calculate_accuracy(y_test, y_prediction, verbose=True)

X_train, y_train, X_test, y_test = load_CIFAR10()
k_set = [3, 5, 7, 11]
best_k = cross_validation(X_train, y_train, k_set)
test_data_validation(X_test, y_test, X_train, y_train, best_k)
```

```
Files already downloaded and verified
Files already downloaded and verified
-------------------cross validation starts-------------------
k = 3
the average accuracy of 5 train folds is 0.176880

-------------------------------------------------------

k = 5
the average accuracy of 5 train folds is 0.188040

-------------------------------------------------------

k = 7
the average accuracy of 5 train folds is 0.197140

-------------------------------------------------------

k = 11
the average accuracy of 5 train folds is 0.230160

-------------------------------------------------------
```



```
The best k with highest average accuracy is 11

-------------------test data validation starts-------------------
```

```
The best k we get from the cross validation is 11
Correct 2226/10000: The test accuracy: 0.222600
```

Report of K Means:

Similar to KNN, divide the training set into five batches, of which four-fifths were taken as the training set and the remaining one-fifth as the test set. The average accuracy was obtained by substituting K =3, 5, 7 and 11 into the calculation respectively. Select the k value with the highest accuracy as k of X_test.

Here, the basic idea of the K-Means algorithm is that the K value is given in advance, K cluster centers are randomly given initially, and the sample points to be classified are divided into clusters according to the nearest neighbor principle. The centroids of each cluster are then recalculated by averaging to determine the new cluster centroids. Iterate until the cluster center stops moving.

In detail, after getting the best k value by cross validation, first randomly select k images (or regarded as k points) as the centroids of k clusters, then the first step is to calculate the distance from each image (point) to each of the k centroids, and then Join the cluster closest to you. In this way, after the first step, each image has its own cluster; in the second step, for each cluster, its centroid is recalculated (average of all image coordinates in it). Repeat steps 1 and 2 until the centroid does not change or changes very little. In the end we have k images, representing k centroids, corresponding to k clusters. At this point, we will calculate the distance from each image in the test set to the k centroids, and predict that the label of the image in the test set is the same as the label of the nearest centroid, and finally the accuracy can be calculated.

Because we randomly pick k centroids each time, the results might vary slightly each time. And when we calculate the distance, we can speed up the algorithm and improve the efficiency by using the matrix instead of loops. However, because the iteration time (the number of runs for which the positions of the k centroids do not change any more) is uncertain, the entire algorithm might be very time-consuming when iterating. I think that if the algorithm needs to be improved, we can set the specific number of iterations, which can be roughly calculated through testing. Or, when the change in the position of the centroid is less than a certain parameter, stop the iteration, which will make the algorithm more complete and efficient.

We found that when k increases, the prediction accuracy also increases, and finally when k = 11, the average prediction accuracy reaches the highest, which is about 23%. It is not difficult to understand that when we select more k centroids at the beginning, we may get more labels, and even all ten labels may be covered, so a reasonable and larger k value can get higher accuracy. For the X_test using the best k=11 value, we get the test accuracy about 22.26%.

Implementation and result of Softmax:

```
import torch
import torchvision
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from torchvision.transforms import ToTensor, Compose
import matplotlib.pyplot as plt
import numpy as np

def load_CIFAR10():
    train_dataset = datasets.CIFAR10(root='data/', download=True, 
train=True, transform=ToTensor())
    test_dataset = datasets.CIFAR10(root='data/', download=True, 
train=False, transform=ToTensor())

    np.set_printoptions(threshold=np.inf)

    X_train = train_dataset.data
    X_test = test_dataset.data

    y_train = np.array(train_dataset.targets)
    y_test = np.array(test_dataset.targets)

    X_train = X_train.reshape(50000, 32 * 32 * 3)
    X_test = X_test.reshape(10000, 32 * 32 * 3)

    '''
    X_train = X_train[:1000]
    X_test = X_test[:500]

    y_train = y_train[:1000]
    y_test = y_test[:500]
    '''

    X_train = X_train.astype("float")
    X_test = X_test.astype("float")

    return X_train, y_train, X_test, y_test

def softmax(f, X_train):
    '''
    -input: a vector of class scores
    -output: probability vector (non-negative, sums to 1)
    '''
    num_train = X_train.shape[0]
    # consider the numeric stability
    # find the largest score and subtract all scores by it in case 
exponential explosion
```

```
    #f_max = np.reshape(np.max(f, axis=1), (num_train, 1))
    f_max = np.max(f, axis=1, keepdims=True)
    prob = np.exp(f - f_max) / np.sum(np.exp(f - f_max), axis=1,
keepdims=True)
    return prob

def softmax_loss(W, b, X_train, y_train, reg):
    # initialization

    num_train = X_train.shape[0]
    num_feature = W.shape[0]
    num_class = W.shape[1]

    dW = np.zeros(W.shape)
    db = np.zeros(b.shape)
    loss = 0.0

    f_scores = X_train.dot(W) + b    # X * W, shape(num_train,
num_class)
    softmax_output = softmax(f_scores, X_train) # shape(num_train,
num_class)

    real_classes = np.zeros(softmax_output.shape)
    real_classes[range(num_train), y_train] = 1.0   # shape(num_train,
num_class)

    # calulate the loss
    loss += -np.sum(real_classes * np.log(softmax_output)) / num_train

    # calculate the gradient descent
    dW += -np.dot(X_train.T, real_classes - softmax_output) /
num_train

    db += np.sum((real_classes - softmax_output),axis=0,keepdims=True)
/ num_train
    return loss, dW, db

def train(W, b, X_train, y_train, lr, reg, num_iters, batch_size):
    # initialization
    num_train = X_train.shape[0]
    num_feature = W.shape[0]

    for i in range(num_iters):

        X_batch = np.zeros((batch_size, num_feature))
        y_batch = np.zeros((batch_size))
        # randomly choose batch_size samples from X_train without
repeatment
        idx_batch =
np.random.choice(num_train,batch_size,replace=False)
        X_batch = X_train[idx_batch]
        y_batch = y_train[idx_batch]

        # get the loss and gradient
        loss, gradient, bias = softmax_loss(W, b, X_batch, y_batch,
reg)

        #loss, gradient, bias = softmax_loss(W, b, X_train, y_train,
reg)

        # grandient descent
        W -= gradient * lr
        b -= bias * lr
    return W,b

def predict_accuracy(W, b, X, y):
    num_train = X.shape[0]
    scores = X.dot(W) + b

    y_prediction = np.zeros(num_train)
    y_prediction = np.argmax(scores, axis=1)

    accuracy = np.mean(y_prediction == y)
    return accuracy

def lr_schedule(X_train, y_train, lrs):
    num_feature = X_train.shape[1]
    num_class = 10

    best_val = -1
    best_lr = 0.0
    reg = 1e4
    accuracys = []

    for lr in lrs:
        W = 0.001 * np.random.randn(num_feature, num_class)
        b = 0.001 * np.random.randn(1, num_class)

        # train the data and get the updated W
        W,b = train(W, b, X_train, y_train, lr, reg, num_iters=3000,
batch_size=200)
        train_accuracy = predict_accuracy(W, b, X_train, y_train)
        accuracys.append(train_accuracy)

        if(train_accuracy > best_val):
            best_val = train_accuracy
            best_lr = lr
            best_W = W
            best_b = b

        print("lr =", lr)
        print("the accuracy of training dataset is", train_accuracy)
        print('\n' +
"--------------------------------------------------------" + '\n')

    plt.plot(lrs, accuracys)
    plt.ylabel('Accuracy of train dataset')
    plt.xlabel('learning rates')
    plt.title("accuracy on learning rate")
    plt.show()

    print("The best learning rate with highest training accuracy is",
best_lr)
    print('\n' +
"--------------------------------------------------------" + '\n')
    return best_lr, best_W, best_b
```

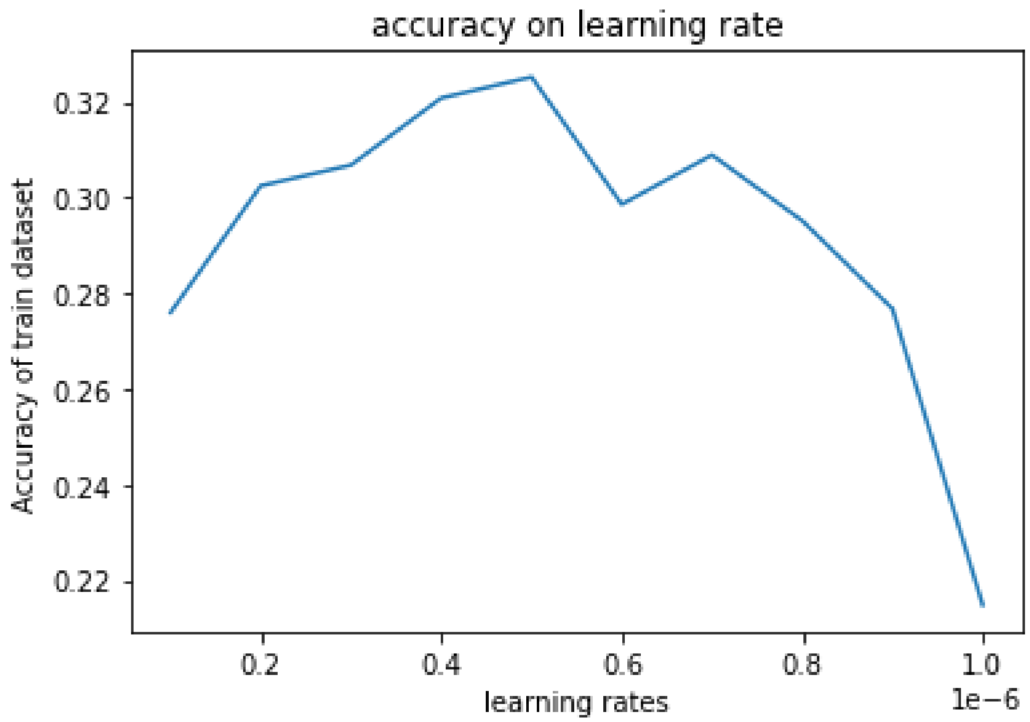
```
X_train, y_train, X_test, y_test = load_CIFAR10()
```

```
Files already downloaded and verified
Files already downloaded and verified
```

```
lrs = [1e-7,2e-7,3e-7,4e-7,5e-7,6e-7,7e-7,8e-7,9e-7,1e-6]
best_lr, W, b = lr_schedule(X_train, y_train, lrs)
test_accuracy = predict_accuracy(W, b, X_test, y_test)

print("By using the best lr, the test accuracy is", test_accuracy)
```

```
lr = 1e-07
the accuracy of training dataset is 0.27602

-------------------------------------------------------


lr = 2e-07
the accuracy of training dataset is 0.30252

-------------------------------------------------------


lr = 3e-07
the accuracy of training dataset is 0.30686

-------------------------------------------------------


lr = 4e-07
the accuracy of training dataset is 0.3209

-------------------------------------------------------


lr = 5e-07
the accuracy of training dataset is 0.32526
```

```
-------------------------------------------------------

lr = 6e-07
the accuracy of training dataset is 0.29864

-------------------------------------------------------

lr = 7e-07
the accuracy of training dataset is 0.30898

-------------------------------------------------------

lr = 8e-07
the accuracy of training dataset is 0.2952

-------------------------------------------------------

lr = 9e-07
the accuracy of training dataset is 0.2768

-------------------------------------------------------

lr = 1e-06
the accuracy of training dataset is 0.2148

-------------------------------------------------------
```



```
The best learning rate with highest training accuracy is 5e-07

-------------------------------------------------------

By using the best lr, the test accuracy is 0.3058
```

Report of Softmax:

The idea of softmax is to use the linear model f=Wx+b to get the score for each class, where W is the learnable weight, x is each image, and b is the learnable bias. After the softmax model is created, the loss function model is used to calculate the cross-entropy loss. And further updates reduce gradient W and bias. By using matrices instead of loops when calculating the loss, we can speed up the operation and improve efficiency.

For each k value, we perform 3000 operations, randomly select 200 images each time to update dW and db, and finally obtain the W and bias corresponding to the k value. Although we tested more than a dozen learning rates, the overall running speed of softmax is still very fast, and we finally return the W and bias corresponding to the best learning rate for testing X_test.

I used ten learning rates to test, and obtained the accuracy of different training data sets at the corresponding learning rates. It can be seen that the accuracy rates obtained by different learning rates are different, and the highest accuracy rate of about 32% corresponds to a learning rate of 5e-7. Therefore, the W and bias corresponding to the optimal learning rate will be used. Further testing on the test set data. As can be seen from the figure, the final test accuracy rate is about 30%.

Conclusion:

Image classification is one of the fundamental research topics in the field of artificial intelligence, and the three methods built in this assignment each have their own merits.

Based on the comparison of the above experimental results, we can see that:

1. KNN is not enough for some specific image classification tasks. The training of the KNN algorithm does not take time (the training process just stores the training set data), but since each test image needs to be compared with all the stored training images, the test takes a lot of time, which is obviously a large disadvantage, because in practical applications, we pay much more attention to the test efficiency than the training efficiency.

   In actual image classification, the KNN algorithm is basically not used. Because images are high-dimensional data (they usually contain many pixels), these high-dimensional data mainly want to express semantic information, not the distance difference between a specific pixel (in an image, the value of a specific pixel is The sum difference basically contains no useful information).

2. The K-means algorithm is a clustering algorithm that is widely used and has a high appearance rate. The idea is simple and the explanatory power is strong. After setting the k value, a specified number of clusters can be output. However, in practical applications, it is also necessary to pay attention to the shortcomings of the K-means algorithm:

   The k value of the K-means algorithm must be determined before clustering, which is often difficult to estimate in the absence of knowledge of the distribution of the dataset, and the optimal k value can only be explored through multiple attempts.

   The k cluster centers in the first iteration of the K-means algorithm have a great influence on the final result of the algorithm, but in the K-means algorithm, the k cluster centers in the first iteration are randomly selected, which gives Algorithmic clustering results bring uncertainty.

   The K-means algorithm gradually optimizes the algorithm in the process of continuous iteration. In each iteration, the distance between each object and the cluster center must be calculated, so when the amount of data is very large, the time overhead is relatively large.

3. In machine learning, especially deep learning, softmax is a very common and important function, especially in multi-classification scenarios. Softmax maps some inputs to real numbers between 0-1, and the normalization guarantees that the sum is 1, so the sum of the probabilities of multi-classification is exactly 1.

   Softmax regression can handle multi-classification problems based on the model itself, but the complex computational problems involved may lead to long running time.