# Artificial intelligence for VR/AR

# Report for Lab1 Session

Xuefeng Wei
Institut Polytechnique de Paris
xuefeng.wei@ip-paris.fr

## Part I. The Nearest Neighbor Classifier and K-Nearest Neighbor Classifier

In the Nearest Neighbor Classifier algorithm section, the Cifar-10 dataset was chosen to test the performance of the algorithm, in order to understand the dataset, the authors identified specific information about the dataset. As shown in Figure 1, the sizes of x_train, y_train, x_test and y_test were determined. The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. The ten classes are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Figure 2 shows the first 10 images in the testing dataset.

```
x_train's size: 153600000 x_train's shape:  (50000, 32, 32, 3)
y_train's size: 50000 y_train's shape:  (50000, 1)
x_test's size: 30720000 x_test's shape:  (10000, 32, 32, 3)
y_test's size: 10000 y_test's shape:  (10000, 1)
```

Fig1. Size of the four vectors (x_train, y_train, x_test, y_test)

To determine the similarity of two images, a distance algorithm is defined to measure the similarity of two images, where the L1 distance is defined as:

$$d_1(I_1, I_2) = \sum_p \left| I_1^p - I_2^p \right|$$

and the L2 distance is defined as:

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$

The L1 distance was first used to test the performance capability of the NN algorithm, and the performance on the 200 test set data is shown in Figure 3, where test accuracy was used to measure the performance of the algorithm, and also the test accuracy for NN in 200 test samples by using L2 distance. From the experimental results, the L1 distance is more suitable for the NN classifier on this dataset

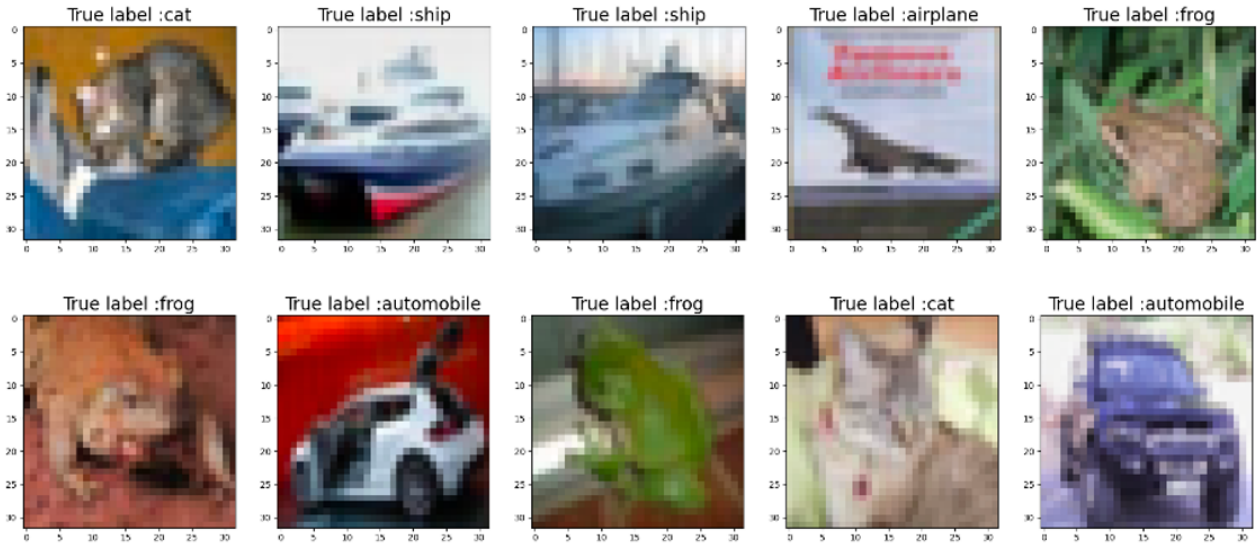



Fig2. First 10 images from the testing dataset

```
Make a prediction for image 198
Make a prediction for image 199
System accuracy = 36.5%

Process finished with exit code 0
```

```
Make a prediction for image 198
Make a prediction for image 199
System accuracy = 35.5%

Process finished with exit code 0
```

Fig3. Testing Accuracy for NN by using L1 distance(left) and L2 distance(right)

Next the KNN classifier is introduced to test its ability to perform on the cifar-10 dataset. Unlike the NN classifier, KNN selects the k closest images and then selects multiple classes to determine the predicted labels, Similarly, L1 and L2 distances are introduced to test the performance, the difference is that for KNN classifier, we need to define a hyperparameter, that is, K value, different K values achieve different test accuracy, the authors chose 3, 5, 10, 20, 50 as K values to observe the results. The performance of the KNN classifier at L1 distance is shown in Table 1, and the results shown in PyCharm console are shown in Figure 4. The performance of the KNN classifier at L2 distance is shown in Table 2, and the results shown in PyCharm console are shown in Figure 5. The overall results show that the system has the greatest testing accuracy when K is set to 10. When K=1, it is observed that the exact value and the NN result are the same, which means that the KNN classifier becomes an NN classifier when K=1.

| **No of neighbors** | **1** | **3** | **5** | **10** | **20** | **50** |
|---|---|---|---|---|---|---|
| **System accuracy** | **36.5** | **35.5** | **38.0** | **39.0** | **38.5** | **35.0** |
| **Time Costing(s)** | **143.99** | **150.03** | **142.43** | **150.94** | **147.92** | **155.10** |

Table 1. Testing Accuracy for KNN by using L1 distance for various numbers of neighbors

| **No of neighbors** | **1** | **3** | **5** | **10** | **20** | **50** |
|---|---|---|---|---|---|---|
| **System accuracy** | **35.5** | **34.5** | **36.0** | **38.0** | **35.5** | **35.5** |
| **Time Costing(s)** | **871.86** | **907.72** | **920.46** | **909.72** | **909.91** | **860.91** |

Table 2. Testing Accuracy for KNN by using L2 distance for various numbers of neighbors

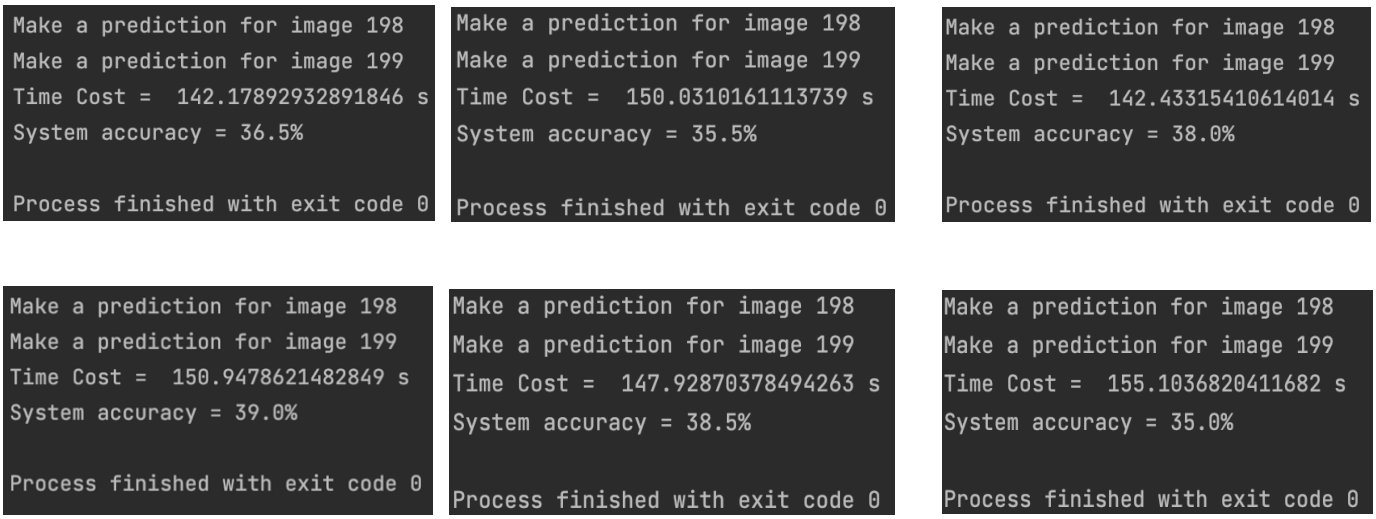



Figure4. Results shown in PyCharm console for KNN by using L1 distance

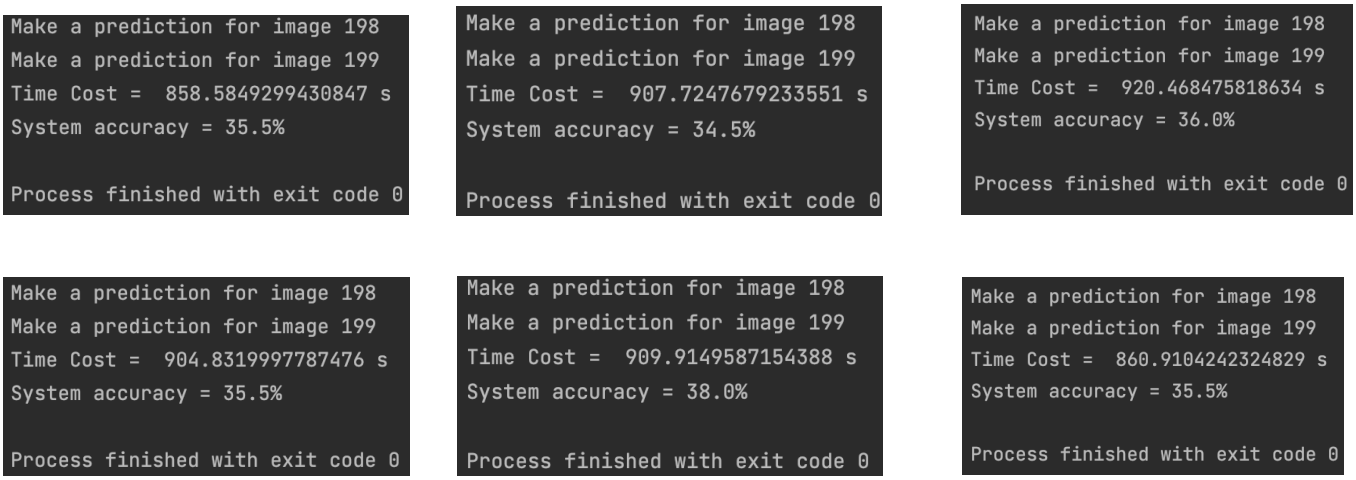



Figure5. Results shown in PyCharm console for KNN by using L2 distance

From the above results, for KNN classifier on this data, using L1 distance is more suitable than using L2 distance because KNN classification performance using L1 distance is better for all different values of K. For the NN classifier, using the L1 distance is also more suitable than using the L2 distance because the test accuracy is higher.

## Part II. Linear Classifier with SVM Loss

In this section, an SVM loss-based linear classifier is designed, which is a linear classifier capable of adjusting the weights according to the training situation. First, a relatively simple set of data is loaded to test the performance of the classifier. The data in the training set are six sets, namely [1, 5, 1, 4], [2, 4, 0, 3], [2, 1, 3, 3], [2, 0, 4, 2], [5, 1, 0, 2], [4, 2, 1, 1, 1], which are labeled [0, 0 ,1 ,1 2, 2]. The data of the test set are three sets, they are [1, 5, 2, 4], [2, 1, 2, 3], [ 4, 1, 0, 1] and their labels are [0, 1, 2]. After the training process starts,

in order for this classifier to converge, once the loss on the training set is below 0.001, the classifier converges, at which point the number of training steps equals 159, and when we test the test set data with the new trained weights W, 100% test accuracy is obtained, as shown in Figure 6 which shows the execution results in PyCharm Console. Therefore, the system accuracy is 100%.

```
Epoch =  148 Loss = 0.12439493229361977
Epoch =  149 Loss = 0.10648753625835654
Epoch =  150 Loss = 0.09776084181559776
Epoch =  151 Loss = 0.07935118237579857
Epoch =  152 Loss = 0.06474103833212173
Epoch =  153 Loss = 0.05249547977960765
Epoch =  154 Loss = 0.03347790886175296
Epoch =  155 Loss = 0.023094385309996374
Epoch =  156 Loss = 0.007221156687733166
Epoch =  157 Loss = 0.004011278320294837
Epoch =  158 Loss = 0.0019476161410827853
Epoch =  159 Loss = 0.0003246026901804642
When the number of steps =  159 The loss variation is inferior to 0.001
Accuracy for test = 100.0%

Process finished with exit code 0
```

Figure6. Number of steps necessary and system accuracy in PyCharm Console

Next, more complex data was loaded to test the performance of the classifier. The dataset is The Iris Flowers Dataset, which contains data features such as sepal length, sepal width, petal length, metal width and the corresponding labels (Iris Setosa, Iris Versicolour, Iris Virginica). A total of 150 samples were included, and a portion of the data is shown in Figure 7.

| SepalLengthCm | SepalWidthCm | PetalLengthCm | PetalWidthCm | Class |
|---|---|---|---|---|
| **5.1** | 3.5 | 1.4 | 0.2 | Iris-setosa |
| **4.9** | 3 | 1.4 | 0.2 | Iris-setosa |
| **4.7** | 3.2 | 1.3 | 0.2 | Iris-setosa |
| **4.6** | 3.1 | 1.5 | 0.2 | Iris-setosa |
| **5** | 3.6 | 1.4 | 0.2 | Iris-setosa |
| **5.4** | 3.9 | 1.7 | 0.4 | Iris-setosa |
| **4.6** | 3.4 | 1.4 | 0.3 | Iris-setosa |
| **5** | 3.4 | 1.5 | 0.2 | Iris-setosa |
| **4.4** | 2.9 | 1.4 | 0.2 | Iris-setosa |
| **4.9** | 3.1 | 1.5 | 0.1 | Iris-setosa |
| **5.4** | 3.7 | 1.5 | 0.2 | Iris-setosa |
| **4.8** | 3.4 | 1.6 | 0.2 | Iris-setosa |
| **4.8** | 3 | 1.4 | 0.1 | Iris-setosa |
| **4.3** | 3 | 1.1 | 0.1 | Iris-setosa |
| **5.8** | 4 | 1.2 | 0.2 | Iris-setosa |

Figure7. A portion of the data of Iris Dataset

Then the data were first shuffled and then divided into 4 parts, which are x_train, y_train, x_test and y_test. The size of training set and test set are 120 and 30 respectively, and the ratio is 0.8 and 0.2. To test the classifier, the optimal value of the training step needs to be found, since the learning rate has a large effect on the classifier, the authors tested the classifier on six different learning rates with the values of 1e-1,1e-2,1e-3,1e-4,1e-5,1e-6, and the maximum number of steps set to 10000. The results are shown in Table3 and Figure 8.

| Learning Rate | 1e-1 | 1e-2 | 1e-3 | 1e-4 | 1e-5 | 1e-6 |
|---|---|---|---|---|---|---|
| Number of Steps | 355 | 470 | 4725 | 4248 | 1 | 1 |
| Maximum Testing Accuracy (%) | 100 | 100 | 100 | 66.66 | 23.33 | 23.33 |

Table 3. Optimal value for the weight's adjustment step for maximum testing accuracy

```
/Users/sid/Desktop/AI_VAR/lab0/venv/bin/python "/Users/sid/Desktop/AI_VAR/lab1/Iris Classifier.py"
When step size = 0.1 Epoch = 355 Maximum testing Accuracy =100.0%
When step size = 0.01 Epoch = 470 Maximum testing Accuracy =100.0%
When step size = 0.001 Epoch = 4725 Maximum testing Accuracy =100.0%
When step size = 0.0001 Epoch = 4248 Maximum testing Accuracy =66.66666666666666%
When step size = 1e-05 Epoch = 1 Maximum testing Accuracy =23.333333333333332%
When step size = 1e-06 Epoch = 1 Maximum testing Accuracy =23.333333333333332%

Process finished with exit code 0
```

Figure8. Optimal value for the weight's adjustment step for maximum testing accuracy

From the results in Table 3, when the learning rate is set to 0.1, the performance of this classifier, From the results in Table III, the classifier performs best when the learning rate is set to 0.1, because it gets the best test accuracy with the smallest number of epochs. The next test is to get the number of training steps needed for this classifier to get over 90% test accuracy. Figure 9 shows this result. The minimum number of steps necessary to train the system to obtain an accuracy superior to 90% is 312.

```
/Users/sid/Desktop/AI_VAR/lab0/venv/bin/python "/Users/sid/Desktop/AI_VAR/lab1/Iris Classifier.py"
When step size set as 0.1, and Epoch = 312 Testing Accuracy superior to 90% Testing Accuracy is 93.33333333333333%

Process finished with exit code 0
```

Figure 9. The minimum number of steps for obtaining an accuracy superior to 90%

The effect of random initialization of the weight matrix on the performance of the classifier is also discussed, firstly the learning rate is set to 0.01, because a lower learning rate allows to observe more clearly the effect of different initial weights on the training, the maximum number of learning steps are set to 200, 5 different initial random matrices are set, where the values range from 0 to 1. Figure 10 shows the performance of the

system for five different random initialized weight matrices. Figure 11 shows their respective accuracy curves and Figure 12 shows their respective loss curves, The impact of random initialization on training can be seen more visually through the curves.

```
/Users/sid/Desktop/AI_VAR/lab0/venv/bin/python "/Users/sid/Desktop/AI_VAR/lab1/Iris Classifier.py"
Round 0 completed... Highest Test Accuracy =76.66666666666667 with Train Accuracy = 79.16666666666666 Train Loss = [0.73045864] Test Loss = [0.02434862]
Round 1 completed... Highest Test Accuracy =90.0 with Train Accuracy = 76.66666666666667 Train Loss = [0.89509961] Test Loss = [0.02983665]
Round 2 completed... Highest Test Accuracy =63.33333333333333 with Train Accuracy = 66.66666666666666 Train Loss = [1.3338855] Test Loss = [0.04446285]
Round 3 completed... Highest Test Accuracy =63.33333333333333 with Train Accuracy = 70.0 Train Loss = [0.78874084] Test Loss = [0.02629136]
Round 4 completed... Highest Test Accuracy =100.0 with Train Accuracy = 89.16666666666667 Train Loss = [0.84897663] Test Loss = [0.02829922]

Process finished with exit code 0
```

Figure 10. The effect of the random initialization of the weight matrix to the system

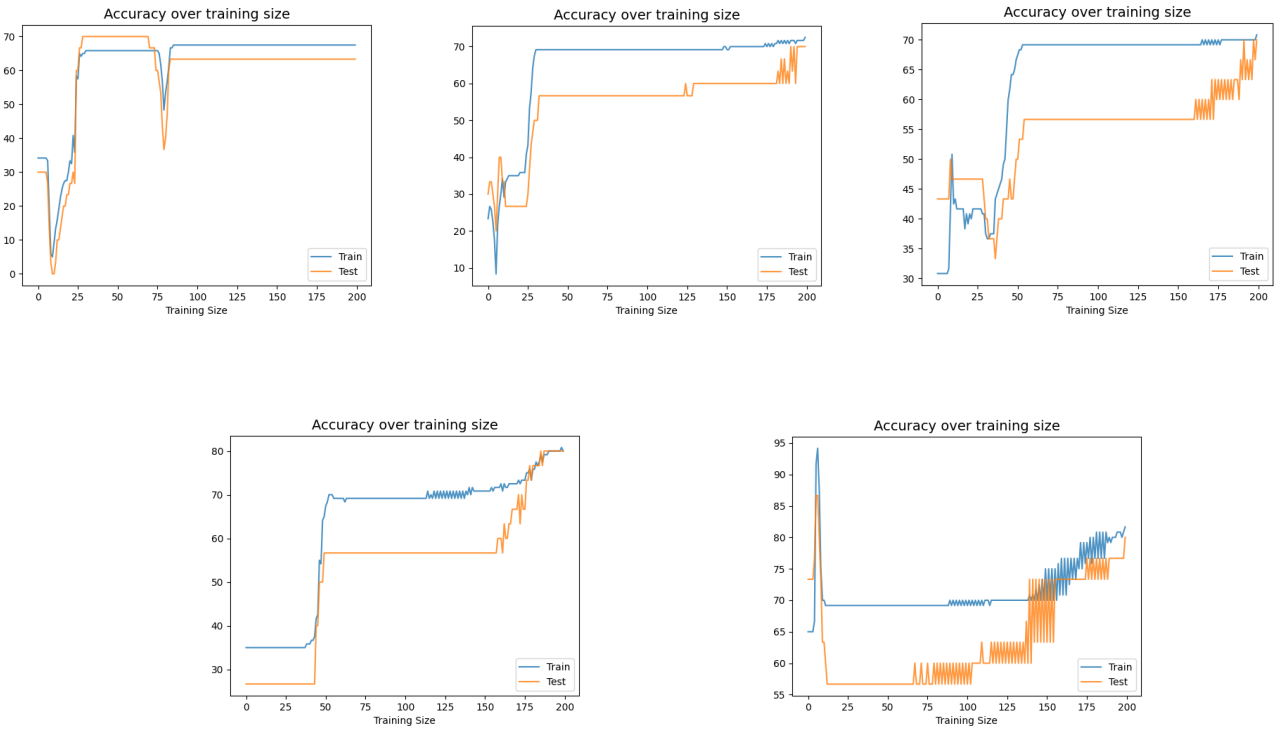
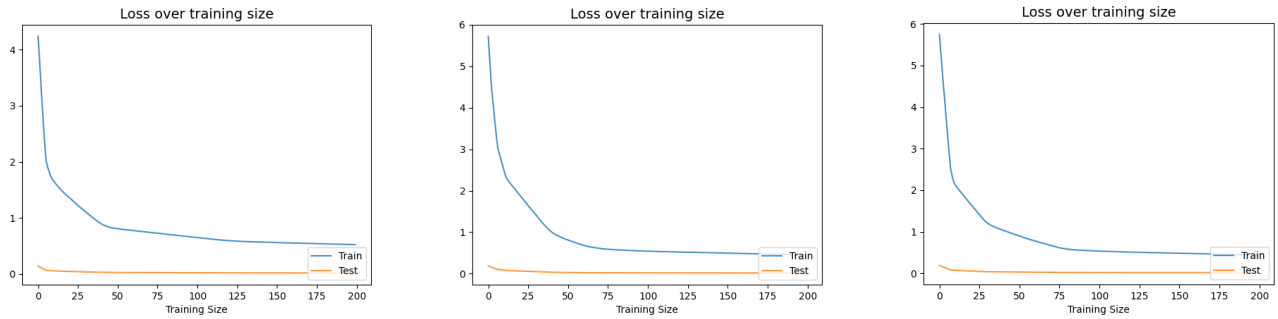



Figure 11. Accuracy Curve for the system with different random initialization of the weights matrix



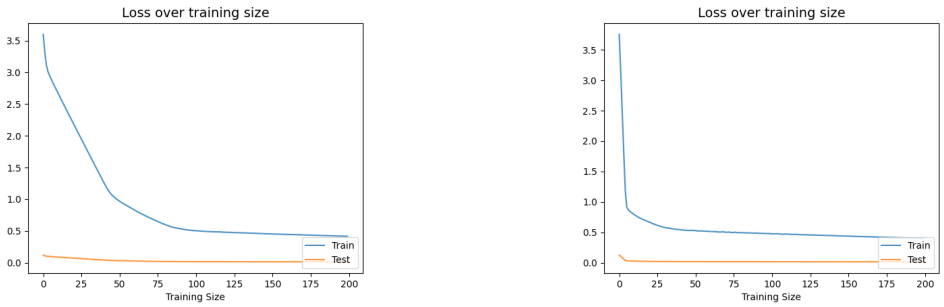



Figure 11. Loss Curve for the system with different random initialization of the weights matrix

From the results in Fig. 10, different initial weight matrices make the system get different optimal training accuracy and test accuracy, and different training loss and test loss, from Fig. 11 and Fig. 12, different initial weight matrices affect the learning process of the system, good initial weights can make the system learn faster and converge more stable, while bad ones will make the learning process more volatile. Therefore, the system influenced by the random initialization of the weights matrix.

Finally, the results in Table III show that the system achieves 100% accuracy on the test set for learning rates of 1e-1,1e-2,1e-3

# Appendix

## Code for Exercise 1

```
print("x_train's size:",x_train.size,"x_train's shape: ",x_train.shape)
print("y_train's size:",y_train.size,"y_train's shape: ",y_train.shape)
print("x_test's size:", x_test.size, "x_test's shape: ", x_test.shape)
print("y_test's size:", y_test.size, "y_test's shape: ", y_test.shape)
```

Comment: The size and shape of the array can be known by using the *. shape and *.size functions, and the results are shown in Figure 1.

## Code for Exercise 2

```
index = 0
n_rows = 2
n_cols = 5
fig, axis = plt.subplots(n_rows, n_cols, figsize=[25, 25])
for row in range(n_rows):
    for col in range(n_cols):
        if index < 10:
            axis[row, col].imshow(x_test[index])
            axis[row, col].set_title(
                "True label :{}".format(dict_classes[index]), fontsize=20)
            index += 1
plt.show()'"
```

Comment:  Visualization of images in a dataset can be achieved by double looping and controlling the values of rows and columns and using the matplotlib.pyplot library, and the results are shown in Figure2.

## Code for Exercise 3

```
def predictLabelNN(x_train_flatten, y_train, img):
    predictedLabel = -1
    scoreMin = 100000000
    for idx, imgT in enumerate(x_train_flatten):
        difference = np.power((img-imgT), 2)
        score = np.sqrt(np.sum(difference))
        if score < scoreMin:
```

```
            scoreMin = score
            predictedLabel = y_train[idx]
    return predictedLabel


def main():
    for idx, img in enumerate(x_test_flatten[0:200]):
        print("Make a prediction for image {}".format(idx))
        predictedLabel = predictLabelKNN(x_train_flatten, y_train, img)
        if predictedLabel == y_test[idx]:
            numberOfCorrectPredictedImages  = numberOfCorrectPredictedImages+1

    accuracy = numberOfCorrectPredictedImages/200*100
```

Comment: The function predictLabelNN() implements the calculation of L2 distance, first calculate the difference between the test image and the training image and then square it, and finally square it, then you can calculate the L2 distance of one image and other images, and finally use the minimum distance as the predicted value, and then find the corresponding label, Finally, in the main function, the total number of correct predictions is obtained by comparing the predicted labels with the true labels, and then the total number of test samples is applied to get the accurate value.

---

## Code for Exercise 4

---

```
def most_frequent(list):
    list = [x[0] for x in list]
    return [max(set(list), key = list.count)]

def predictLabelKNN(x_train_flatten, y_train, img, K):
    predictedLabel = -1
    predictions = []  # list to save the scores and associated labels as pairs  (score, label)
    for idx, imgT in enumerate(x_train_flatten):
        difference = abs(img-imgT)
        score = np.sum(difference)
        predictions.append((score, y_train[idx]))
    predictions = sorted(predictions, key=lambda x :x[0])
    predictions_K  = predictions[0:K]
    predLabels: List[Any] = []
    for element in predictions_K:
        predLabels.append(element[1])
    predictedLabel = most_frequent(predLabels)

    return predictedLabel
```

Comment: KNN algorithm using L1 distance, unlike the NN algorithm, KNN only goes to the K points with the smallest distance, and then uses the highest frequency label as the prediction label, The results are shown in Table 1 and Figure 4.

---

## Code for Exercise 5

---

```
def most_frequent(list):
    list = [x[0] for x in list]
    return [max(set(list), key = list.count)]

def predictLabelKNN(x_train_flatten, y_train, img, K):
    predictedLabel = -1
    predictions = []    # list to save the scores and associated labels as pairs    (score, label)
    for idx, imgT in enumerate(x_train_flatten):
        difference = np.power((img-imgT), 2)
        score = np.sqrt(np.sum(difference))
        predictions.append((score, y_train[idx]))
    predictions = sorted(predictions, key=lambda x :x[0])
    predictions_K    = predictions[0:K]
    predLabels: List[Any] = []
    for element in predictions_K:
        predLabels.append(element[1])
    predictedLabel = most_frequent(predLabels)

    return predictedLabel
```

Comment: Like the KNN algorithm using L1 distance but using L2 distance to measure the distance of feature points. The results are shown in Table 2 and Figure 5.

---

## Code for Exercise 6&7

---

```
def predict(xsample, W):
    s = []
    s = np.dot(W,xsample)
    return s

def computeLossForASample(s, labelForSample, delta):
    loss_i = 0
```

```
    syi = s[labelForSample]    # the score for the correct class corresonding to the current input sample based on the label yi
    for i in range(len(s)):
        if i != labelForSample:
            loss_i = loss_i + max(0, s[i]-syi+delta)
return loss_i

def computeLossGradientForASample(W, s, currentDataPoint, labelForSample, delta):

    dW_i = np.zeros(W.shape)    # initialize the matrix of gradients with zero
    syi = s[labelForSample]     # establish the score obtained for the true class
    for j, sj in enumerate(s):     # j=idx sj=content
        #print(j)
        #print(sj)
        dist = sj - syi + delta
        if j == labelForSample:
            continue
        if dist > 0:
            dW_i[j] = currentDataPoint
            dW_i[labelForSample] = dW_i[labelForSample] - currentDataPoint
    return dW_i

def main():
    for i in range(300):
        for idx, xsample in enumerate(x_train):
            s = predict(xsample, W)
            loss_i = computeLossForASample(s, y_train[idx], delta)
            dW_i = computeLossGradientForASample(W, s, x_train[idx], y_train[idx], delta)
            loss_L = loss_L + loss_i
            dW = dW + dW_i

        loss_L = loss_L / len(x_train)
        print("Epoch = ", i+2 ,"Loss = {}".format(loss_L))

        dW = dW / len(x_train)
        W = W - step_size * dW
        if loss_L < 0.001:
            break
    print("When the number of steps = ", i+1,"The loss variation is inferior to 0.001")

    correctPredicted = 0
    for idx, xsample in enumerate(x_test):
        predictedlabel = np.argmax(np.dot(W,xsample))
        #print(predictedlabel)
        if predictedlabel == y_test[idx]:
            correctPredicted=correctPredicted+1


    accuracy = correctPredicted/len(y_test)*100      # Modify this
```

```
    print("Accuracy for test = {}%".format(accuracy))

    return
```

Comment: The gradient of each sample is calculated by calling computeLossForASample() and ComputeLossGradientForASample() and then updating the weight matrix. To determine the number of steps needed, the number of steps is determined by a loop followed by an if condition. Then the final weight matrix can be found by argmax function to predict the label.

---

## Code for Exercise 8

---

```
def findOptimalLearningRate(x_train,x_test,y_train,y_test,W,dW,delta,loss_L,step_size):
        maxacc = 0
        bestepoch = 0
        for i in range(10000):
            for idx, xsample in enumerate(x_train):
                s = predict(xsample, W)
                loss_i = computeLossForASample(s, y_train[idx], delta)
                dW_i = computeLossGradientForASample(W, s, x_train[idx], y_train[idx], delta)
                loss_L = loss_L + loss_i
                dW = dW + dW_i
            loss_L = loss_L / len(x_train)
            dW = dW / len(x_train)
            W = W - step_size * dW
            correctPredicted = 0
            for idx, xsample in enumerate(x_test):
                predictedlabel = np.argmax(np.dot(W, xsample))
                # print(predictedlabel)
                if predictedlabel == y_test[idx]:
                    correctPredicted = correctPredicted + 1
            accuracy = correctPredicted / len(y_test) * 100
            if accuracy > maxacc:
                maxacc = accuracy
                bestepoch = i+1
            if maxacc == 100:
                break
        print("When step size =", step_size, "Epoch =", bestepoch, "Maximum testing Accuracy
={}%".format(maxacc))

def minimumnumberOfsteps(x_train,x_test,y_train,y_test,step_size):
    W = np.array([[0.82163581, 0.01537692, 0.12706988, 0.5888949],
                  [0.17020233, 0.89392429, 0.2827656, 0.83561792],
                  [0.96253776, 0.49765929, 0.15274233, 0.01755514]])
    dW = np.zeros(W.shape)
    delta = 1    # margin
    loss_L = 0
    epoch = 0
```

```
    ACC=0
    for i in range(10000):
        for idx, xsample in enumerate(x_train):
            s = predict(xsample, W)
            loss_i = computeLossForASample(s, y_train[idx], delta)
            dW_i = computeLossGradientForASample(W, s, x_train[idx], y_train[idx], delta)
            loss_L = loss_L + loss_i
            dW = dW + dW_i

        loss_L = loss_L / len(x_train)
        dW = dW / len(x_train)
        W = W - step_size * dW

        # print("When the number of steps = ", i + 2, "The loss variation is inferior to 0.001")
        correctPredicted = 0
        for idx, xsample in enumerate(x_test):
            predictedlabel = np.argmax(np.dot(W, xsample))
            # print(predictedlabel)
            if predictedlabel == y_test[idx]:
                correctPredicted = correctPredicted + 1

        accuracy = correctPredicted / len(y_test) * 100
        if accuracy > 90:
            epoch = i + 1
            ACC=accuracy
            break
    print("When step size set as 0.1, and Epoch =", epoch, "Testing Accuracy superior to 90%","Testing
Accuracy is {}%".format(ACC))

def RandomWeights(x_train,x_test,y_train,y_test,step_size,round):
    W = np.random.rand(3,4)
    dW = np.zeros(W.shape)
    delta = 1    # margin
    loss_L = 0
    loss_L_test = 0
    epoch = 0
    ACC = 0
    loss_train = []
    loss_test = []
    acc_train = []
    acc_test = []
    best_train_acc = 0
    best_test_acc = 0
    best_train_loss = 0
    best_test_loss = 0
    for i in range(200):
        for idx, xsample in enumerate(x_train):
            s = predict(xsample, W)
            loss_i = computeLossForASample(s, y_train[idx], delta)
```

```
            dW_i = computeLossGradientForASample(W, s, x_train[idx], y_train[idx], delta)
            loss_L = loss_L + loss_i
            dW = dW + dW_i
        loss_L = loss_L / len(x_train)
        dW = dW / len(x_train)
        W = W - step_size * dW

        for idx, xsample in enumerate(x_test):
            s_test = predict(xsample, W)
            loss_i_test = computeLossForASample(s_test, y_test[idx], delta)
            loss_L_test = loss_L_test + loss_i_test
        loss_L_test = loss_L / len(x_test)


        # print("When the number of steps = ", i + 2, "The loss variation is inferior to 0.001")
        correctPredicted_test = 0
        correctPredicted_train = 0
        for idx, xsample in enumerate(x_test):
            predictedlabel = np.argmax(np.dot(W, xsample))
            if predictedlabel == y_test[idx]:
                correctPredicted_test = correctPredicted_test + 1


        for idx, xsample in enumerate(x_train):
            predictedlabel = np.argmax(np.dot(W, xsample))
            if predictedlabel == y_train[idx]:
                correctPredicted_train = correctPredicted_train + 1


        accuracy_test = correctPredicted_test / len(y_test) * 100
        accuracy_train = correctPredicted_train / len(y_train) * 100

        if accuracy_test > best_test_acc:
            best_test_acc = accuracy_test
            best_train_acc = accuracy_train
            best_train_loss = loss_L
            best_test_loss = loss_L_test

        acc_train.append(accuracy_train)
        acc_test.append(accuracy_test)
        loss_train.append(loss_L)
        loss_test.append(loss_L_test)

    plot_curve(loss_train,loss_test,acc_test,acc_train,round)
    print("Round {} completed...".format(round), "Highest Test Accuracy
={}".format(best_test_acc),"with Train Accuracy = {}".format(best_train_acc),"Train Loss =
{}".format(best_train_loss),"Test Loss = {}".format(best_test_loss))
```

```
def main():

    with open('Iris.csv', 'r') as csvfile:
        reader = csv.reader(csvfile)
        rows = [row for row in reader]

    data = np.array(rows)
    data = data[1:]
    np.random.shuffle(data)

    x_train = np.array(data[0:120, :4]).astype('float')
    x_test = np.array(data[120:150, :4]).astype('float')
    y_train = np.array(data[0:120, 4:])
    y_test = np.array(data[120:150, 4:])
    labelcoding(y_train)
    labelcoding(y_test)
    y_train = y_train.astype('int')
    y_test = y_test.astype('int')

    ''' Exercise 8 Step1: Find the optimal value for the weights adjustment step in order to obtain the maximum testing accuracy '''
    for element in [0.1, 0.01, 0.001, 0.0001, 0.00001, 0.000001]:
        W = np.array([[0.82163581, 0.01537692, 0.12706988, 0.5888949],
                      [0.17020233, 0.89392429, 0.2827656, 0.83561792],
                      [0.96253776, 0.49765929, 0.15274233, 0.01755514]])
        # print(W)
        dW = np.zeros(W.shape)
        delta = 1    # margin
        step_size = 1    # weights adjustment ratio , learning rate
        loss_L = 0
        findOptimalLearningRate(x_train, x_test, y_train, y_test, W, dW, delta, loss_L, element)

    ''' Exercise 8 Step2: Find the minimum number of steps necessary to train in order to obtain an accuracy superior to 90%    '''
    minimumnumberOfsteps(x_train, x_test, y_train, y_test, 0.1)

    ''' Exercise 8 Step3: random initialization of the weights matrix    '''
    for element in range(5):
        RandomWeights(x_train, x_test, y_train, y_test, 0.01, element)
```

Comment: The red, blue, and green code segments represent the codes of the first three questions of exercise 8, respectively, in which the fourth question has been solved. The red code segment seeks to find the optimal number of times the weight matrix is more wanted has reached the maximum test accuracy, and the value can be determined by controlling the loop variable through an if statement, while the authors discuss the value at different learning rates, and the results are shown in 8.  The green code shows the minimum number of learning steps to determine the system to reach 90% accuracy, again determined by an if statement, which

the authors find with a learning rate of 0.1, and the result is shown in Figure 9. The last blue code is about the effect of the random weight matrix on the system, by loading five different random weight matrices for training, and then drawing the corresponding learning curve and loss curve in order to observe more intuitively the effect of the random initialization matrix on the model performance, and the results are shown in Figure 10 and Figure 11.