

# Efficient Multiclass k-NN Classification on MNIST with PyTorch-based Rapid Vectorized Calculation

KE Siyun. kesiyun@hanyang.ac.kr<sup>1</sup>

## *Abstract*

This report is prepared for **answering the questions provided in Assignment #1** of the 2023 Hanyang Summer Course AIX0005. The objective of this report is to investigate the execution of a multiclass classification task on the MNIST dataset utilizing the k-Nearest Neighbors algorithm. Furthermore, this report briefly elucidates the author's conceptualizations for the project, particularly concerning the endeavors to incorporate matrix calculations as a replacement for time-complex iteration algorithms, along with various other supplementary methodologies.

## **Introduction**

The project implemented a classification algorithm on the handwritten digit dataset MNIST. Specifically, the chosen algorithm was the k-Nearest Neighbors (k-NN) algorithm, known for its straightforward conceptualization and ease of implementation. A basic implementation of k-NN may involve computing distances between each data entry through iterative processes and selecting the k-closest entries.

Nevertheless, the original k-NN algorithm with iterative calculations possesses a time complexity of  $O(n^3)$ , which becomes a critical concern under circumstances with large datasets. When considering the entire MNIST dataset, consisting of 60,000 training images (each with 28x28 pixels), the computational demands become prohibitively high, resulting in unacceptably long processing times (e.g., approximately 41 days for validation using 6,000 samples).

To address this challenge, the project explores the integration of vectorized calculations utilizing tensor operations. Meanwhile, broadcast features included in PyTorch or TensorFlow may significantly enhance the efficiency of matrix calculations, thus alleviating the excessive computational burden compared to the original iterative approach.

---

<sup>1</sup> garfield.ke@connect.um.edu.mo

# Implementation

## Data input

MNIST dataset was randomly shuffled and the first 6000 (10%) entries as validation set were taken out from the total 60000 entries of the training set. The datasets were reshaped to an  $n \times 784$  matrix for calculation ( $n$  = the size of the dataset;  $784 = 28 \times 28$ ).

## Original approach $O(n^3)$ and some exploration on distance $O(n^2)$

The author first implemented an incomplete solution for the k-NN task. This is a step-by-step repeat of the k-NN pseudocode. The main function and a function to calculate the distance formed a triple iteration. This approach was soon proven to be unrealistic on 60000 data entries.

```
def fit_and_validate_model(k):
    min_k=np.zeros(shape=(1,k))[0]#k
    min_k_indices=np.zeros(shape=(1,k))[0]#k
    left_vacancies=k

    for i in range(0,val_count): #i-th image in val
        print(i)
        print(min_k_indices)
        for j in range(i+1,train_count):
            #print(i, " ",j)
            #print(min_k)
            #print(min_k_indices)

            #if(i<=j): continue

            current_max=min_k.max()
            current_max_index=min_k.argmax()
            current_min=min_k.min()
            current_min_index=min_k.argmin()
            current_kth=0
            if left_vacancies==0: current_kth=current_max

            this_dist=Ecu_distance_TCH(val_data[i], train_data[j], current_kth)
            #if(this_dist==-2): print("continued")
            if left_vacancies>0:
                min_k[current_min_index]=this_dist #To replace 0s
                min_k_indices[current_min_index]=j
                left_vacancies=left_vacancies-1
            else:

def Ecu_distance(tensorline1, tensorline2, current_kth):
    Shape1=tensorline1.shape[0]
    Shape2=tensorline2.shape[0]
    if Shape1!=Shape2: return -1 #error
    d_sq=0
    for i in range(0,Shape2):
        d_sq=d_sq+(tensorline1[i]-tensorline2[i])**2
        if (d_sq>current_kth and (current_kth!=0)): return -2 #not necessary to continue
    return float(d_sq)
```

Some alternative methods to calculate distance were tested and significantly reduced the time needed for training. They may reduce the time complexity to  $O(n^2)$ , but approximately the whole algorithm will take around four days to complete the prediction (on Google Colab Engines)

```
def Ecu_distance_NN(tensorline1, tensorline2, current_kth):
    #Shape1=tensorline1.shape[0]
    #Shape2=tensorline2.shape[0]
    #if Shape1!=Shape2: return -1 #error
    d_sq=F.pairwise_distance(tensorline1, tensorline2, p=2)**2 #ecu->p==2

    return float(d_sq)

def Ecu_distance_TCH(tensorline1, tensorline2, current_kth):
    #Shape1=tensorline1.shape[0]
    #Shape2=tensorline2.shape[0]
    #if Shape1!=Shape2: return -1 #error
    d_sq=torch.sum((tensorline1-tensorline2)**2)

    return float(d_sq)
```

## Final structure with distance matrix $O(n)$

The final solution in this project directly computes a distance matrix between two sets. The documents from well-known libraries like scikit-learn reveal they utilized the broadcast features to speed up the computation. Some resources<sup>2</sup> suggest an equation (shown below) and the distance matrix can be formed in this way. This is a self-implemented method for experimental purposes, alternative methods with more flexible features (e.g. torch.cdist) can be referred to if needed. (Important while calculating with other distance approaches)

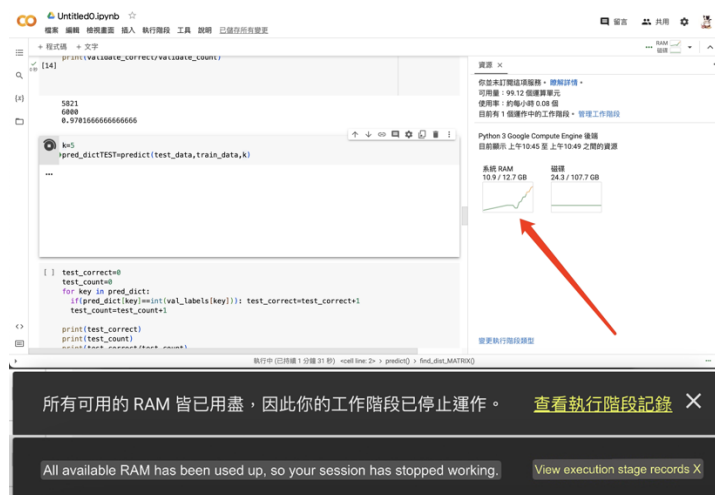
$$dists_{ij} = \sqrt{x_i x_i - 2x_i y_i + y_i y_i}$$

```
#Distance matrix!  
def find_dist_MATRIX(val_data,train_data):  
    #dd=val_data[i]**2+train_data[i]**2-2*val_data[i]*train_data[i]  
  
    m = val_data.size(0)  
    n = train_data.size(0)  
    xx = (val_data**2).sum(dim=1,keepdim=True).expand(m, n)  
    yy = (train_data**2).sum(dim=1, keepdim=True).expand(n, m).transpose(0,1)  
    xy = val_data.matmul(train_data.transpose(0,1))  
    dd = xx-2*xy+yy  
    return dd
```

The final approach using a distance matrix may find out the required k neighbors with only one iteration. The result was usually calculated in several minutes.

## Batch processing to reduce RAM usage

The calculation for the matrix could easily run out of allocated RAM on Colab if 60000 entries in the testing set were predicted in a single batch. Therefore, a measure to split the testing set into batches each with a maximum of 10000 data was designed to address this issue.

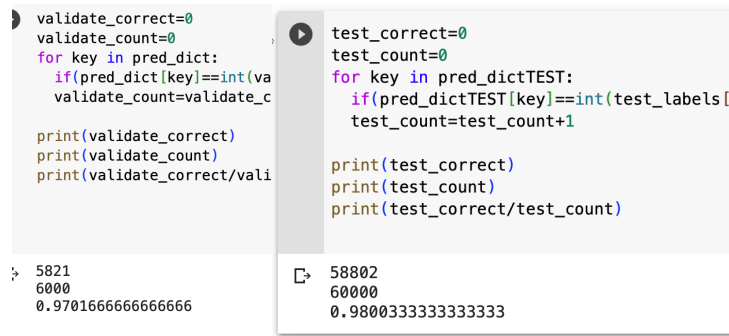


<sup>2</sup> Might be incredible: <https://zhuanlan.zhihu.com/p/428563664>

## Evaluation and answers to assignment questions

### 1- Accuracy (Euclidian distance & $k=5$ )

When using Euclidian distance and considering 5 nearest neighbors, the validation achieved an accuracy of 97% (5821 correct out of 6000). The performance of testing is similar, which is 98%(58802 correct out of 60000)



The screenshot shows two code cells from a Jupyter Notebook. The left cell contains code for validation accuracy, and the right cell contains code for test accuracy. Both cells print the number of correct predictions, the total number of predictions, and the resulting accuracy.

```
validate_correct=0
validate_count=0
for key in pred_dict:
    if(pred_dict[key]==int(va
        validate_count=validate_c

print(validate_correct)
print(validate_count)
print(validate_correct/vali

5821
6000
0.9701666666666666
```

```
test_correct=0
test_count=0
for key in pred_dictTEST:
    if(pred_dictTEST[key]==int(test_labels[
        test_count=test_count+1

print(test_correct)
print(test_count)
print(test_correct/test_count)

58802
60000
0.9800333333333333
```

### 2- Hyperparameters involved in this task

In the implementation provided by the author, the following hyperparameters can be tuned:

- k value.** This is the most obvious one in the k-NN algorithm. A larger value k may have a more precise classification but might also have overfitting issues.
- Weighting algorithm.** The current algorithm simply counts the closest 5 entries and finds the one that appeared the most. An alternative way is to take their distance into account (e.g. Neighbors with smaller distances are more important).
- Distance measuring method. Besides Euclidean distance, other distance methods like Manhattan distance may be tested.

### 3- Performance on alternative hyperparameters

Based on answers in 2-, the following modification on hyperparameters were introduced.

- k value. Both **smaller** and bigger value of k is tested ( $k= 3, 7$ , or  $9$ ) The result is listed here:

k value	Accuracy
<b>3</b>	<b>0.9721666666666666(BEST)</b>
5 [original]	0.9702
7	0.9711666666666666
9	0.9708333333333333

- b. Two distance weighting algorithms were introduced. They specifically treat the closest one or the farthest one as the most important. (changed code base in prediction)

Method	Accuracy
NIL [original]	0.9702
<b>w=1/d (closest)</b>	<b>0.972(BEST)</b>
w=d (farthest)	0.9696666666666667

- c. Two alternative distance algorithms were tested (changed code base in distance calculation -used torch.cdist instead of self-implemented ones)

Distance mode	Accuracy
<b>Euclidean [original]</b>	<b>0.9702 (BEST)</b>
Manhattan	0.9696666666666667
Minkowski	0.967

#### 4- Final performance on TEST images

With the best arrangement provided in the last answer (i.e. k=3, w=1/d, Euclidean), the accuracy of the test set is,

Correct (original)	Total	Accuracy (original)
59109	60000	0.98515
(58802)		(0.9800333333333333)

## Remarks

The Colab share link for verification-

<https://colab.research.google.com/drive/187K3Yom0ZI72jiwneOxojZIQMcYGikcX?usp=sharing>

[End]