HANOI UNIVERSITY OF SCIENCE AND TECHONOLOGY

SCHOOL OF INFORMATION COMMUNICATION TECHNOLOGY

Capstone project report:

Hand-written digits recognition system

Supervised by:

Associate Professor Than Quang Khoat

Group members:

Dao Vu Tien Dat - 20225479

Vu Nhu Duc - 20225485

Ta Tuan Hai - 20225442

Vo Thanh Vinh - 20226074

Hanoi - Vietnam

2023

Contents

1 Brief Introduction

1.1 About the problem

The development in the field of Artificial Intelligence in recent years has proved that A.I is very popular in the world nowadays. Since the introduction of "ChatGPT" back in 2022, this field has received more and more attention from the public. Lots of A.I technology has been developed, and applied in various fields such as healthcare, automobiles, housing, stock exchanges, ..etc... so many fields that people can sense the presence of A.I everywhere now.

As a group of students studying "Introduction to Artificial Intelligence - IT3160E", we understand the importance of this subject and the fundamental building blocks of it to study further advanced topics. And lying in a simple thing there exists a fundamental yet very important problem, that is designing a system that can recognize digits from an input image. This system has many potential applications, such as reading license plates when you enter a shopping mall, or reading license plates to recognize potential traffic rules violations on camera.... Due to the complexity of processing from the environment, for the context of this course, we will design the system to recognize digits from a drawing canvas. The fundamental blocks of this system can be developed further so that It will be able to solve and assist us with real-world problems.

1.2 The dataset

1.2.1 The MNIST dataset

The dataset that we used in this project is a simple dataset called MNIST (An acronym for Modified National Institute of Standards and Technology database) [11], the dataset consists of 70000 images of hand-written digits, and each image is represented in a black and white image. They are commonly used for training various image processing systems. The images have been normalized, processed to fit into a 28x28 pixel bounding box, and anti-aliased. This is the dataset that we can try to implement some of our algorithms to design the system.

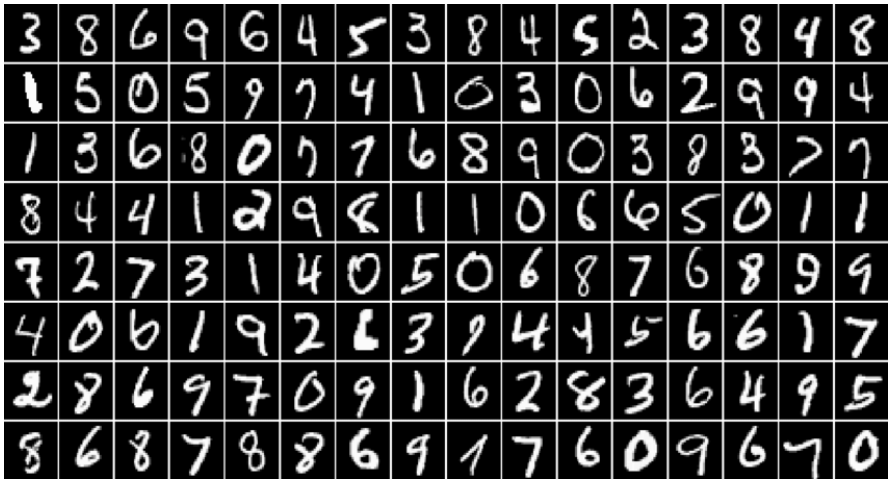


Figure 1: MNIST Dataset

1.3 Libraries / packages to be used

1.3.1 Programming Language

Python is an interpreted, high-level, general-purpose programming language. According to "Popularity of Programming Language" [3] , Python is the most popular programming language in the world. Python supports a lot of useful libraries that assist in building intelligent systems such as "Numpy", "Matplotlib", "Pandas" and so much more. Thus we will use this programming language for this project.

1.3.2 Numpy

Numpy [12] is a strong library that allows us to do fast calculations, manipulate on 1d, array, a matrix, and multi-dimensional arrays (also known as tensors) which can take a very long time to do in a normal Python list, the libraries also proved to be extremely powerful when it comes to data analysis and has been the fundamental building blocks for other packages/modules such as matplotlib, pandas, seaborn, scikit-learn. Numpy is also known for its inheritance of simplicity. So this library will help us do faster operations to create and help our system "learn".

1.3.3 Pandas

In computer programming, pandas is a software library written for the Python programming language for data manipulation and analysis [16] [18]. This is an extremely powerful tool, especially for tabular data manipulation. In this project, pandas will be used specifically

to help us store images from additional datasets into tabular data, each column will represent the pixel of the image, in which there will be 784 columns in total.

1.3.4 Matplotlib

Matplotlib [13] is a comprehensive library for creating static, animated, and interactive visualizations in Python. Seaborn [21] is a Python data visualization library based on Matplotlib, they are integrated and have many features and useful tools. They will help us visualize the multiple images while training to verify whether the system is learning correctly or not.

1.3.5 Scikit-learn

Scikit-learn [9] is a free machine-learning library for the Python programming language, the library gives us tools to implement fast Machine Learning with lots of useful methods to help us process the images before feeding them into the system. Not just processing methods, it also has many attributes that we will use to experiment on which algorithms work well for the problem.

1.3.6 OpenCV

OpenCV (Open Source Computer Vision Library) [8] is a library of programming functions mainly for real-time computer vision. Despite having many methods, functions, and utilities for computer vision, for this project, we will use only some functions of this library to draw the bounding box with the prediction on the canvas.

1.3.7 Tensorflow

TensorFlow [6] is a free and open-source software library for machine learning and artificial intelligence. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks. Due to the complexity of programming and the time taken for each training process. We will implement a Convolution Neural Network using the TensorFlow library, which assists us in using GPU to help with the speed of training.

1.3.8 Pillow

Pillow [10] [20] is a Python Imaging Library which is a free and open-source additional library for the Python programming language that adds support for opening, manipulating,

and saving many different image file formats. For the context of this problem, we use this library to retrieve images from the canvas so that we can process them, feed them through the model, and finally return the predictions.

1.3.9 Tkinter

Tkinter [15] [1] is a standard Python GUI library. And this is how we will be able to build/deploy our models/solutions. The canvas will consist of a canvas where we can draw the numbers, 5 buttons of which 3 will be used for prediction using our 3 proposed algorithms, 1 button to show the bounding box of every digit drawn on the screen, and 1 button to clear the canvas.

2 Preprocessing Data

2.1 Preprocessing the training data

Most algorithms that we use for this problem will rely mainly on a vector being fed into a model and it will calculate and print out the prediction. So it's reasonable for our team to take the first step to preprocess the data by reshaping the 28 x 28 images into a vector of size (784,), this process is called flattening the image, by putting all rows of the image next to each other on a single row. The dataset, which it's size is (70000, 28, 28) now be transformed into a 2D array of size (70000, 784).

Then to avoid calculating large numbers, and large gradients (for neural networks) as well as to reduce the effect of illumination on the image, we will scale the picture from range [0, 255] down to [0, 1] by dividing the pixel value by 255.

For special implementations such as neural network (MLP) and Convolution neural network (CNN), we will have to one-hot encode the label, which is changing the label of each instance into a probability distribution, which we will mention about later. But for now, the label y of the dataset will transform into a probability distribution. Which is the following:

$$P(y = i|X) = \begin{cases} 1 & \text{if } y = i \\ 0 & \text{otherwise} \end{cases}$$

Below is the function where we process the MNIST dataset. dividing by 255 so each value of pixel will be in range[0, 1] and the label becomes a probability distribution.

```
'''
The dataset has shape (70000, 28, 28)
The label has shape (70000, 1)
After processing, the shape of the label should be
(70000, 10)

'''
#The current shape of y is (n, 1) -> transform it into (n, 10)
def oneHotEncoder(Y):
    y_ = np.zeros((Y.shape[0], 10))
    n, m = Y.shape
    for i in range(n):
        for j in range(m):
            y_[i, j] = 0
        y_[i, Y[i]] = 1
    return y_

#preprocess data for the implementation
def preprocessData(train_X, train_y):
    train_X_new = train_X / 255.
    train_y_new = oneHotEncoder(train_y)
    return train_X_new, train_y_new
```

2.2 Preprocessing the testing data

To fit the image to the model to be predicted, we need to fit the testing images into MNIST format. It is simply a better solution since the MNIST dataset is the first dataset that we use to create a model to recognize the hand-written digits, so later on when we augment more data into the system, the system should also be able to recognize hand-written digits that are in MNIST format. which is resizing the image down to 28x28, but before that, some crucial steps of preprocessing are required:

Firstly we will need to dilate the image, this means that the thickness of the digit will increase, so that when we scale it down to 28x28, it will not be blurry and thin. Then we will separate each digit into separate images, then each image can be predicted and combined for a result showing on a new window. The separation can be done by finding the bounding box of each digit. The process of finding the bounding box of each digit is to locate the upper left corner and lower right corner of the bounding box. This step is done easily by Depth-First search (DFS) on the canvas. From the core, how the computer sees an image is a "3d-tensor" of numbers, each tensor layer represents the color channel Red, Green, and Blue in every pixel of the image, and, the number which will indicates the intensity of "red", "green", "blue" of each pixel in that image. ranging from 0 to 255 (16 squared). But for black-and-white images. The image is presented by only a 2D tensor (or we familiarly call it a matrix) of pixels. The number on that matrix will indicate the intensity of the "whiteness" of that pixel. 0 will indicate and 255 will indicate white. So the DFS algorithm will work as follows.

For every non-zero non-visited pixel found while scanning through the matrix, we will perform DFS search on it, finding every non-zero non-visited pixel that belongs to that connected component of the matrix. Then while searching, we will locate the minimum x_{min}, y_{min}, and maximum x_{max}, y_{max} coordinates of the connected components. The resulting coordinates will be a bounding box of the digit.

After locating the bounding box for every digit in the canvas. The separated digits image will then be padded with columns and rows of 0 to make the bounding box square so that when we resize the picture down to 28x28 pixel images, the image scale will not be changed/distorted. Finally, we will flatten the newly processed image into a 784-sized vector, so that it can be used to be predicted.

The code below is the implementation of using DFS to find the bounding box of each digit in the new image:

```
1 def fit(self, img):
2
3         #Initialize variables before performing DFS
4         m, n = img.shape
5         self.vst = np.zeros(img.shape)
```

```
        self.num_label = 0
        self.contours = []
        #Perform DFS search on canvas
        for i in range(m):
            for j in range(n):
                if img[i, j] != 0 and self.vst[i, j] == 0:
                    self.min_x, self.min_y = float('inf'), float('inf')
                    self.max_x, self.max_y = 0, 0
                    self.num_label += 1

                    self.dfs(img, m, n, i, j, self.num_label)
                    self.contours.append((self.min_x, self.min_y,
                                          self.max_x, self.max_y))

        #Adding bounding boxes of each digit to contour
        data, img_data = self.digit_segmentation(img, self.num_label, self.contours)
        self.processed_img, self.unflattened_img = data, img_data
```

The method *digit_segmentation* is used to get the sub-image of each digit out of the contour, fixing it into a square, then resizing it into 28x28 size images. After processing, we will have an array of (784, 1) size vectors and an array of (28, 28, 1) images.

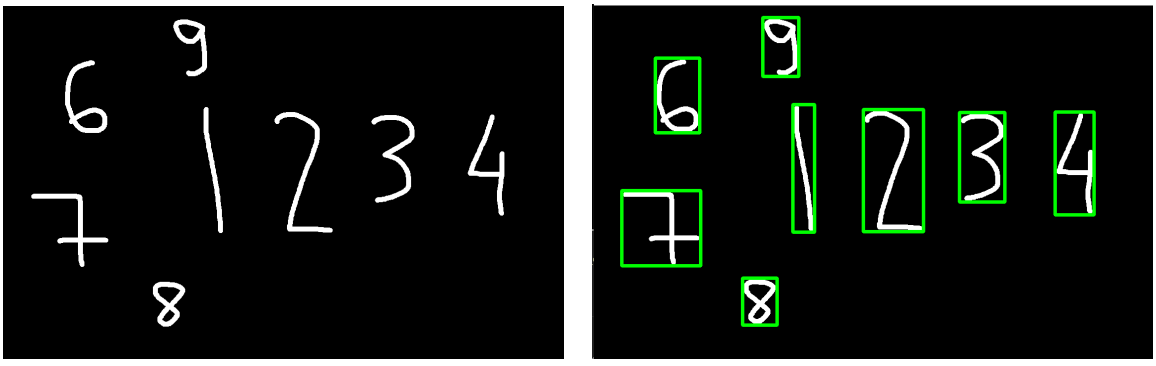


Figure 2: Images before and after plotting bounding box

3 Proposed algorithms

3.1 K-Nearest Neighbors

3.1.1 Intuitive Idea

K-nearest neighbor (KNN) [17] is a supervised learning algorithm in Machine Learning. It is used for classification and regression by picking k closest training examples in the dataset to predict the output (label) of new data. For classification problems, the class of the data is decided based on "major votes". This means that around K nearest data (in this example vectors) around them based on geometry or different metric distances, which class has the most appearance will be chosen to be the class of the testing data. This is the first algorithm that we will propose to design a system. Regarding picking k nearest neighbors, we will consider 2 main ways to choose those vectors.

The first way is by using Euclidean distance. This way is more popular than the second way because of its efficiency. In this case, we find k nearest neighbors by computing the square root of the sum of the deviations to the power of 2. A deviation is the distance between a point in the training vector with a point at that position of the test vector. Then we need to sort the calculated distances in ascending order and keep the K smallest distances.

The Euclidean distance between two vector p, q is:

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i=1}^{N} (p_i - q_i)^2}$$

The second way to compute is the Manhattan distance. We just need to calculate the sum of the deviations instead of computing the square root and then do the same as the first way.

The Manhattan distance between two vector p, q is:

$$d(\mathbf{p}, \mathbf{q}) = \sum_{i=1}^{N} |p_i - q_i|$$

For KNN, we are making every nearest neighbor's weight equal, which means that every neighbor in that K-nearest neighbors set has the same weight, even the K-th closest one and the closest one to the testing vector. In practice it may have some bad effects, resulting in bad/inaccurate predictions. So we need to assign weight to them. It means that the nearest vector to the testing vector has more "weight" when we consider the label of the testing vector. The closer the vector is to the testing one, it will have more influence on the output of the data than the further ones.

Our goal is to find out the best distance measure, a suitable number of K, and the best weight option so that KNN has the highest accuracy and can be used to predict real handwritten numbers. For this problem, we will implement KNN by using the "Sklearn" library and "Matplotlib" library to help us visualize our experiments. We will split the dataset into 2 different sets, one for training and one for testing. We will split the MNIST dataset into 60000 samples for training size, 10000 for test

3.1.2 New Findings / Insights

First, we will implement KNN with Euclidean distance, with different weights. We will test two popular "kernel methods" for weighting, the "distance" and "Gaussian", The "uniform" kernel does not use any kernel functions and all nearest neighbors have the same weight.

The Gaussian weight is of the following:

$$w_i = e^{-\frac{d_i^2}{2\sigma^2}}$$

- where d_i is the distance between the neighbor i and the testing instance.
- σ is a fixed constant, and can be modified to adjust the influence of the weights.

By implementing the KNN algorithm on the MNIST dataset by testing on different "nearest neighbors, we have found some "interesting results"

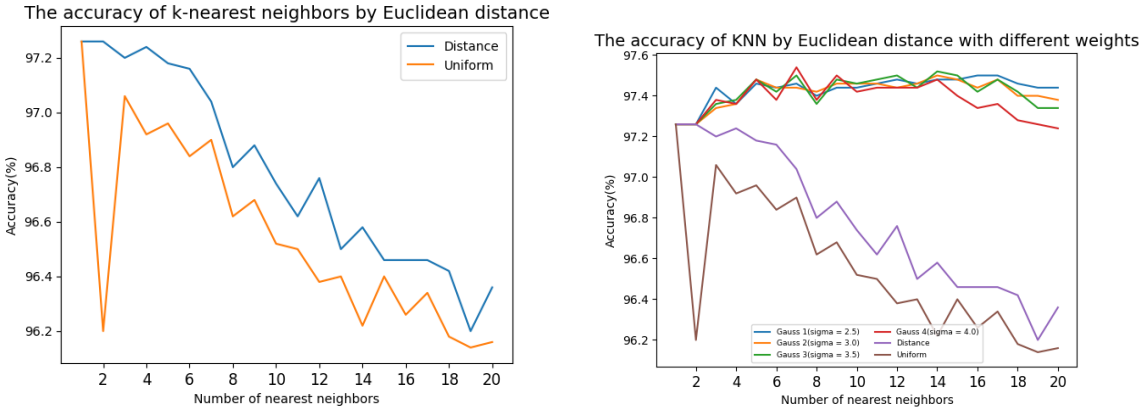


Figure 3: plots of KNN with Euclidean distance

From the above line graph, we can conclude the following:

- For larger numbers of K, the accuracy, although still very high, tends to decrease with "uniform" and "distance" weighting functions.

- The accuracy of "my weight" (the Gaussian weighted kernel functions) appears to be more consistent.
- We discovered that a different sigma value, specifically $\sigma = 2.5$, seems to be the most optimal for KNN with a higher number of K. Smaller sigma values inversely increase the influence of the nearest neighbors on the testing set in the Gaussian weighted kernel function.
- Each of these processes takes a considerable amount of time to run. Each cell in our Python notebook takes approximately 5 minutes to run, which is quite slow. Thus, it might not be practical to apply this algorithm in real-life scenarios.

Secondly, we implement KNN with "Manhattan" Distance, with different weights, applying the same "kernel methods" for weighting as the given experiment, and the result is shown below

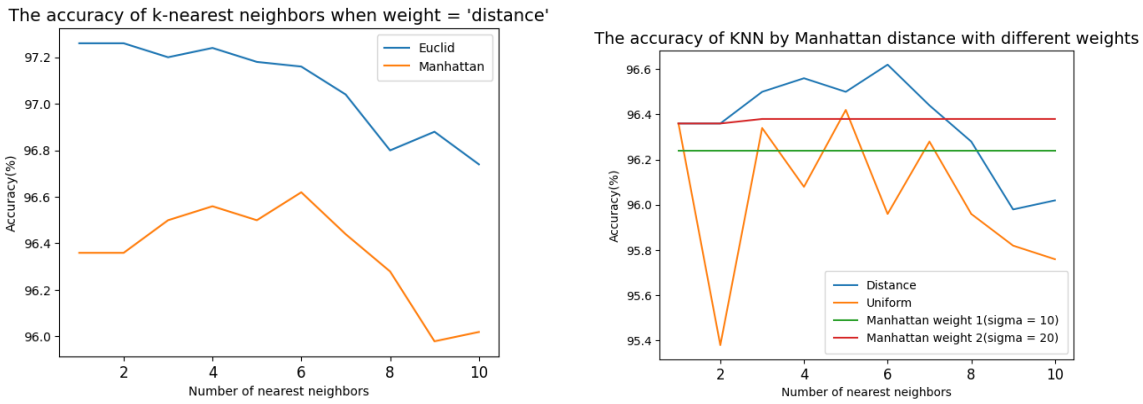


Figure 4: plots of KNN with Manhattan distance

Here are some of our conclusions:

- Surprisingly, the accuracy remains relatively consistent for large numbers of $\sigma = 10, 20$.
- While conducting some experiments, we've found out that for every value of $\sigma < 10$, the accuracy of the model remains consistent while changing the number of K.
- As for uniform weight (where all nearest neighbors have the same weight of 1), the accuracy fluctuates and tends to decrease as the number of K increases.
- The time taken to run each variation of KNN was extremely long. Each model took around 10 minutes of runtime. The training process was nearly instantaneous, but the testing process took a very long time.

- The reason behind this is that KNN doesn't really "learn" anything; it simply stores the training data. When the testing data are fed into the model, it calculates the distance from the testing data to every vector in the training set, and then sorts to find the K-nearest ones.

From the given experiments we can conclude that KNN has really good performance, and took zero training time but the testing phase took very long, so it's not suitable for real-life applications. But for demonstration, we will use it for the demo.

Conclusion: Through all the plots, we have concluded that KNN with Euclidean distance with Gaussian Kernel of $\sigma = 4$ and value k = 7 is the model that will be deployed.

3.1.3 Implementation

Below is the part of the code that we used to plot out some experiment result of Euclidean distance, the code of other weights are nearly the same, so we will show the part where we test the accuracy on $\sigma = 4$

```
def my_weight_4(distance):
    sigma_square = 4
    return np.exp(-(distance**2) / sigma_square)

acc_4 = []
for i in range(1, 21):
    print("Running on {} Neighbors".format(i))
    knn_4 = KNeighborsClassifier(n_neighbors = i, weights = my_weight_4)
    knn_4.fit(x_train, y_train)
    acc_4.append(knn_4.score(x_test[1:5000], y_test[1:5000]))
```

The rest of the code and the experiment can be found at the end of this report.

3.2 Multilayer perceptron / Artificial Neural Network

3.2.1 Intuitive idea

Neural networks were first proposed in 1944 by Warren McCullough and Walter Pitts, two University of Chicago researchers. The idea is to stimulate how neurons in our brain might work. and in 1957, a layered network of perceptrons, consisting of an input layer, a

hidden layer with randomized weights that did not learn, and an output layer with learning connections, was introduced by Frank Rosenblatt in his book Perceptron [19]. And this is our second algorithm to recognize hand-written digits.

Multilayer Perceptron (MLP) is another name for a modern feedforward artificial neural network. Which consists of fully connected neurons with some non-linear activation function. It is known to learn and distinguish complex data that are not linearly separable. With given data, it can be used to recognize and make accurate predictions about non-linear structure, which in this case is our hand-written digits.

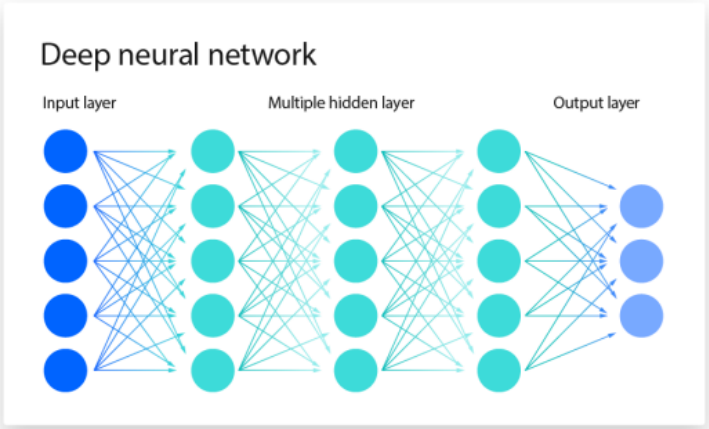


Figure 5: MLP structure (source: IBM)

MLP consists of at least three layers, one input layer, one (or more hidden layers), and one output layer. The more hidden layers we add, the better the MLP can learn complex structures. It will learn through an algorithm called Back Propagation, which uses using "Gradient descent" method, trying to minimize the loss function by calculating the gradient of the loss function with respect to each of the weights, then the weights of the neural network will be updated.

The loss function that MLP uses to learn is the cross-entropy loss function, which is cross entropy. Cross entropy is a measure of the difference between two probability distributions for a given random variable or set of events. The formula of cross entropy is below:

$$\text{Cross-Entropy Loss} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij} \cdot \log(p_{ij})$$

Where y_{ij} represents the true probability distribution over class j for the ith instance (in

one-hot encoded vector), p_{ij} represents the predicted probability distribution over class j for the ith instance.

The output layer of the neural network comes in 10 units, which can be considered as a vector of size 10. Which v[i] will represent the probability $\mathrm{P}(y_{predicted} = i \mid X = \text{given image})$, the true label of the image which is considered to be a probability distribution as follows:

$$P(y = i|X) = \begin{cases} 1 & \text{if } y = i \\ 0 & \text{otherwise} \end{cases}$$

Our team's neural network will have an input layer of 784 units, each unit representing a pixel in a flattened 28 x 28 image. The output layer has 10 units/nodes, which if a number is a 0, 1, ... 9 then that node will be "switched" on. Moreover, the Softmax activation function will be used for the output layer. It is typically a useful function for multi-class classification problems. In this case, it is classifying each digit to predict whether they are 0, 1, ..., 9.

$$\text{Softmax}(\mathbf{z_i}) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$$

Where Softmax($\mathbf{z_i}$) represent the probability $\mathrm{P}(y_{predicted} = i \mid X = \text{given image})$

For activation functions in the hidden layers, we will use "relu" (Rectified Linear unit) for our activation function. It is easy, fast to compute, non-linear, and is a very robust choice for activation functions for neural networks nowadays.

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

But in practice, to avoid the problem called "dead neurons" inside our neural network, which means that the value passing through one neuron always returns zero, implying it is inactive during training, we will use a variant of "relu", which is called "leaky relu". The leaky relu activation function is shown below:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01 & \text{otherwise} \end{cases}$$

Our problem now is to determine the best architecture for our artificial neural networks. Finding out the number of hidden layers, determining the number of units in each hidden layer. So that it can achieve high accuracy without overfitting and can generalize well on unseen data.

3.2.2 New findings/conclusions

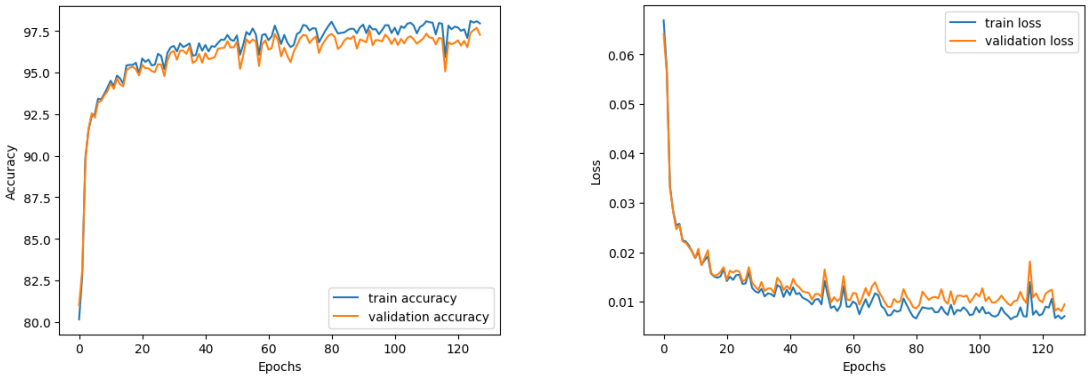
All the number of nodes in a hidden layer are all power of 2, inspired by this article [7]. We will split the MNIST dataset into 3 sets, a training set, which consists of 48000 samples, a validation set which consists of 12000 images to make sure the neural network can generalize well and the test sets for the final testing with 10000 samples. We have done some research on the simple feed-forward neural networks using Adam optimizer, which is an accelerator for the learning rate to help the loss function escape local extrema, and using a batch size of 32. We have come up with interesting results.

Num Epoch	Learning rate	batch size	hidden layer dims	acc
128	1×10^{-3}	32	512 256 128	97.08
128	1×10^{-3}	32	256 256 256	97.07
128	1×10^{-3}	32	512 512 512	97.5
128	1×10^{-3}	32	32	96.76
128	3×10^{-3}	32	512 256 128	96.85
128	1×10^{-3}	32	512 512 256 256 128 64	97.52

Table 1: Experimental result of MLP architecture

- The accuracy of the neural network did not increase significantly when we tried increasing the number of hidden layers as well as their dimensions.
- The more hidden layers we add and the more dimensions we add to it, the longer the training time becomes. For instance, the last architecture took the longest with 50 minutes of training time, while the third architecture also took around 45 minutes for training.
- Furthermore, the more hidden layers we add, the more vulnerable the neural network is to overfitting.

Conclusion: Since the accuracy barely changes for different architectures, We have decided to pick the architecture that has an accuracy that is above 97% and has the least parameters so that the calculating process would be time–effective, which is the first architecture (3 hidden layers with each has size 512, 256 and 128).



(a) Accuracy (b) Loss

Figure 6: Accuracy and loss of selected neural network architecture

3.2.3 Implementation

We can implement this neural network very conveniently using Tensorflow library, but for this capstone project, our team wanted to implement MLP from scratch, taking inspiration from these articles [5], [4]

Firstly we have to define some activation functions that our neural network uses, which are "leaky relu", and "softmax":

```
def leakyRelu(x):
    return np.where(x > 0, x, 0.01 * x)
def softmax(z):
    ez = np.exp(z - np.max(z, axis = 0))
    return ez / np.sum(ez, axis = 0, keepdims = True)

def leakyRelu_derivative(x):
    return np.where(x > 0, 1, 0.01)
```

First, we have to create a function to do forward propagating, calculating the value from the given data (input layer) through hidden layers then end at the output layer.

```
self.store["A0"] = X.T #(L - 1, m)
for layer in range(1, self.L + 1):
    self.store["Z" + str(layer)] = np.dot(self.store["W" + str(layer)],
                                          self.store["A" + str(layer - 1)]) +
                                          self.store["b" + str(layer)]
    if layer != self.L:
```

```
        self.store["A" + str(layer)] = leakyRelu(self.store["Z" + str(layer)])
    else:
        self.store["A" + str(layer)] = softmax(self.store["Z" + str(layer)])
```

With forwarding propagation, we need to write the back propagation function to calculate the gradient of the loss function (cross-entropy) with respect to each weight in each layer, with the given gradients, we can then implement gradient descent to minimize the loss function.

```
n = X.shape[0]
self.fowardProp(X)
dZ = self.store["A" + str(self.L)] - y.T
dW = 1. / n * np.dot(dZ, self.store["A" + str(self.L - 1)].T)
db = 1. / n * np.sum(dZ, axis = 1).reshape(-1, 1)
dA_prev = np.dot(self.store["W" + str(self.L)].T, dZ)

self.derivative["dW" + str(self.L)] = dW
self.derivative["db" + str(self.L)] = db

for l in range(self.L - 1, 0, -1):
    dZ = dA_prev * leakyRelu_derivative(self.store["Z" + str(l)])
    dW = 1. / n * np.dot(dZ, self.store["A" + str(l - 1)].T)
    db = 1. / n * np.sum(dZ, axis = 1).reshape(-1, 1)
    if l > 1:
        dA_prev = np.dot(self.store["W" + str(l)].T, dZ)

    self.derivative["dW" + str(l)] = dW
    self.derivative["db" + str(l)] = db
```

Now the last thing is to implement gradient descent on the loss function given the input data to train our neural network. Since computing using the entire dataset is very time-consuming, we will implement mini-batch gradient descent to train our neural network.

```
for epoch in range(num_iteration):
    permutation = np.random.permutation(X.shape[0])
    X_shuffled = X[permutation, :]
    y_shuffled = y[permutation, :]
    for batch in range(batches):
        begin_range = batch * batches
        end_range = min(begin_range + batch_size, X.shape[0] - 1)
```

```
        X_batch = X_shuffled[begin_range: end_range, :]
        y_batch = y_shuffled[begin_range: end_range, :]

        batch_size_real = end_range - begin_range
        if batch_size_real <= 0:
            continue
        self.fowardProp(X_batch)
        self.backProp(X_batch, y_batch)
        for l in range(1, self.L + 1):
            W_new, b_new = self.updateParameters(l, epoch, optimizer, learning_rate)
            self.store["W" + str(l)] = W_new
            self.store["b" + str(l)] = b_new
```

More detail on the codes that we used to implement MLP will be shown at the end of the report.

3.3 Convolutional neural network (CNN)

3.3.1 Intuitive Idea

In the 1980s, Convolutional Neural Networks (CNNs) were first introduced to the world by Yann LeCun, a computer science researcher. This marked the beginning of a revolutionary approach to the field of Artificial Intelligence, especially Computer Vision [14]. The key idea behind the convolutional neural network is also to mimic how the brain works. However, the difference between ANNs and CNNs is that while ANNs are designed to be fully connected networks where each neuron is connected to every neuron in the next layer, CNNs include convolutional layers that use specific filters to extract image features from the given input. Thus, CNNs are more effective for image-related tasks, and in our case, they have shown noticeable improvement in terms of accuracy in the task of recognizing hand-written digits.

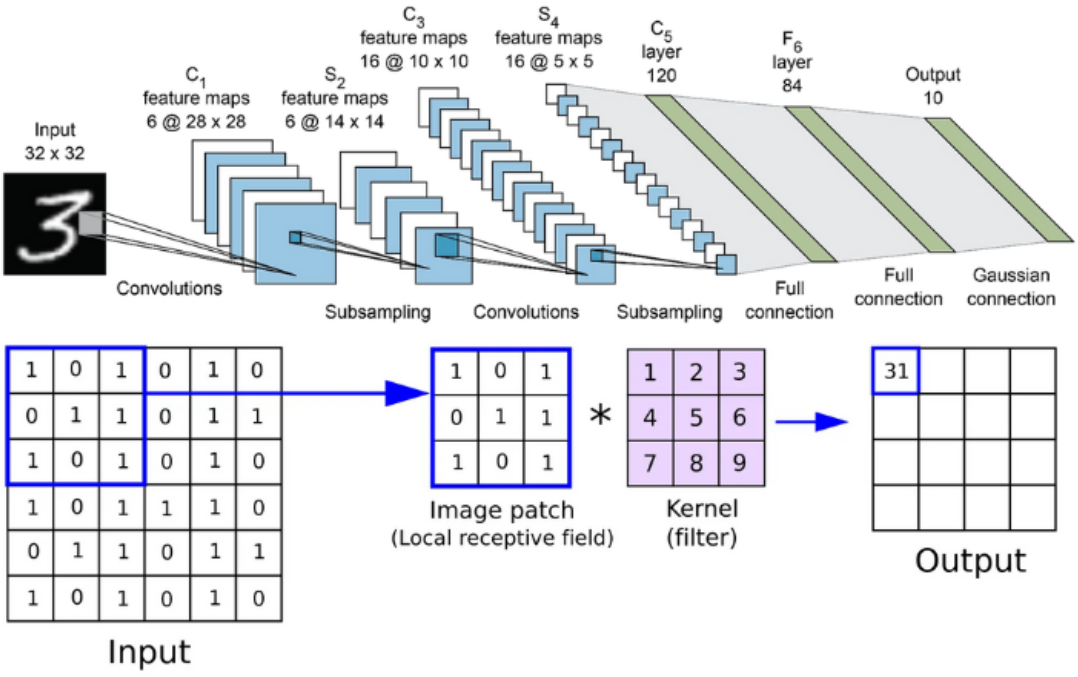


Figure 7: Strucutre of a convolutional neural network (Source: ANH H. REYNOLDS)

The 2 basic components that distinguish between CNNs and MLP are Convolution and Subsampling (or pooling)

- **Convolution**: Convolution (or sometimes called cross-relation) is a mathematical operation that allows the merging of two sets of information. In the case of CNN, convolution is applied to the input data to filter the information and produce a feature map. This filter is also called a kernel, or feature detector. Each element in the filter matrix is also a parameter that needs to be learned so that the network can extract features from input images more effectively.

- **Pooling**: Pooling in the convolutional network is termed downsampling or subsampling, which minimizes the number of neurons in the convolutional layer by retaining the important information. The two most famous one is max pooling and average pooling. In this problem, we will use max pooling in our pooling layers.

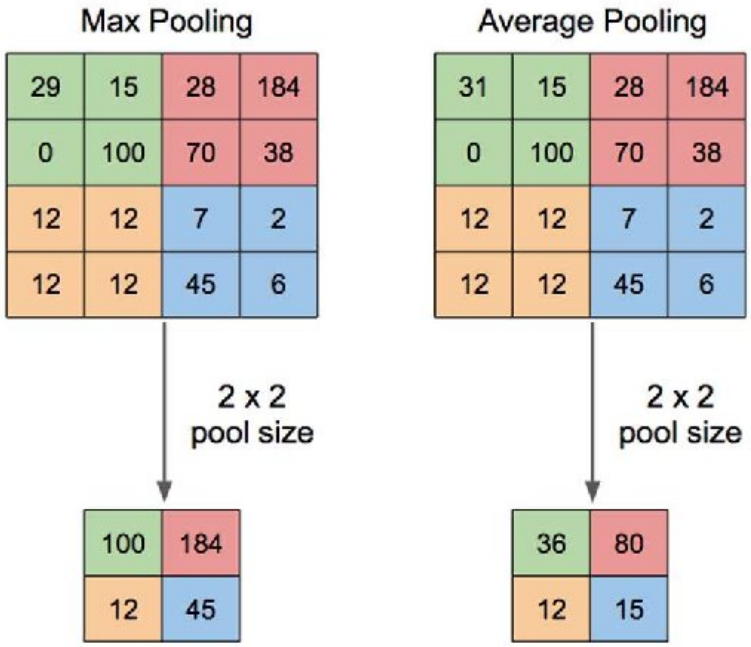


Figure 8: Example of 2x2 pooling layers with stride = 2(source: Muhamad Yani)

To be more specific, CNNs consist of some significant components. They often have one input layer and a series of convolutional layers and subsampling pairs. In these layers, we can choose the activation function, specify properties such as the number of filters, the size of kernels, strides, and padding, and decide on the type of subsampling layers (MaxPooling, AveragePooling, etc.). After several convolutional layers and subsampling blocks, a flattening process is applied to transform the input into a vector. This vector is then passed through fully connected layers to make the final prediction.

As mentioned, we have an input layer with 784 units and a 10-unit output layer that uses the softmax activation function. The difference lies in implementing the CNN architecture between these two layers. The challenge is how to choose the architecture to achieve high accuracy without overfitting or underfitting.

3.3.2 Experiments/new findings

In our experiment, we also utilized the rescaled and split dataset from the ANN part. Due to the numerous components that required decisions, we opted to conduct separate tests for each element. The architecture experiment for CNNs is based on the Kaggle notebook cited here: [2].

To begin with, we aimed to determine the optimal number of pairs of convolutional layers and subsampling blocks that would yield the highest accuracy while maintaining acceptable computational costs. For the sake of the argument, we standardized the kernel size to 5x5 and employed the ReLU activation function with the same padding and no strides. Throughout the process, we compared three different scenarios: 1 pair, 2 pairs, and 3 pairs of Conv-Subsampling blocks. Assuming the first block had 32 filters, we incrementally increased the number of filters in the second and third blocks to 48 and 64 respectively, and then plotted the validation accuracy of those 3 models.

Figure 9: Accuracy plot of the model with different number of pairs of Conv-Subsampling blocks

As can be seen from the graph above, the accuracy of the validation set for both 2-pair and 3-pair configurations was similar. Due to computational cost, we prefer to implement 2 pairs of Conv-Subsampling blocks into our architecture.

Next, we need to determine the number of filters to use in each convolutional layer. In this step, we have decided to implement 2 pairs of Conv-Subsampling blocks, each with a kernel size of 5x5, and we will apply the ReLU activation function. We aim to compare the results of six different models, each with a different number of filters. Denote that the i-th model has $((i-1)*8+8)$ filters in the first layer and $((i-1)*16+8)$ filters in the second layer. We can then plot the graph of accuracy on the validation set as follows:

Figure 10: Accuracy plot of the model with different number of filters in each Conv-Subsampling blocks

From the graph, we can see that 24 filters in the first convolutional layer and 48 filters in the second layer gave stable accuracy. Hence, we will choose 24-48 filters in our architecture.

After 2 blocks of Conv-Subsampling blocks, we will then flatten the previous input into a vector. Our next target is to decide how many nodes we should use in the fully connected layer to make the final prediction. In this experiment, we will choose the best model out of 8 models whose nodes range from 0 (no fully connected layer at all) to 2048 nodes. We can illustrate the accuracy of each model as follows:

Figure 11: Accuracy plot of the model with different numbers of nodes in fully connected hidden layer

As illustrated, we can see that the accuracy in 256 nodes in a fully connected layer is acceptable amongst 8 models. Regarding the number of layers in fully connected layers, from experimenting the accuracy does not change much when the number of layers varies, so we decided to choose only 1 layer in the fully connected layer. Thus, our fully connected layer will consist of a layer whose nodes are 256, connected to the 10-node output layer.

To reduce overfitting, a common practice is to add a regularization technique. In this case, we have chosen dropout regularization. Dropout is a technique where neurons are removed randomly, and these removed neurons' weight will not be passed into the forward pass or the backward pass. We need to determine the probability that randomly selected nodes will be dropped out. We will evaluate 8 models with dropout rates ranging from 0 (no dropout) to 70% chance that a node is dropped. The results are as follows:

Figure 12: Accuracy plot of the model with different dropout rates

In conclusion, the final CNN architecture consists of an input layer that takes in (28,28) shape array of pixel values, followed by two convolutional layers and a subsampling block. The number of filters in these layers is set to 24 and 48, with a kernel size of 5x5, using the same padding and no strides. The architecture also includes a fully connected layer with 256 nodes and implementing a dropout rate of 0.2. The final accuracy on the test set is 99.30

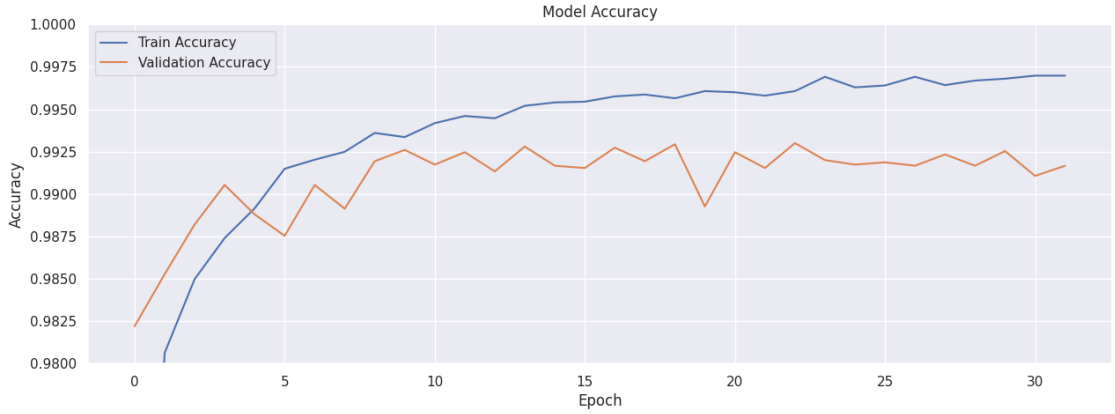


Figure 13: Accuracy plot of train set and validation set on selected CNN structure

3.3.3 Implementation

Implementing convolutional network from scratch takes a lot of time and does not have the support for training from GPU, so it's not effective to implement this approach from scratch. Therefore we will use "Tensorflow" library to help us implement CNN easily.

The full code for experimenting is very very long, consists of many cell inside an ipynb file, so we will only show the creation of the final selected CNN architecture for usage. Which is implemented below:

```
final_model = Sequential()
final_model.add(Conv2D(filters=24, kernel_size=(5, 5), activation='relu',input_shape=(28,28,1)))
final_model.add(MaxPooling2D())
final_model.add(Dropout((0.2)))

final_model.add(Conv2D(filters=48, kernel_size=(5, 5), activation='relu'))
final_model.add(MaxPooling2D())
final_model.add(Dropout(0.2))

final_model.add(Flatten())
final_model.add(Dense(256, activation='relu'))
final_model.add(Dropout(0.2))
final_model.add(Dense(10, activation='softmax'))
final_model.compile(optimizer="adam", loss="categorical_crossentropy", metrics=["accuracy"])
```

4 Arose difficulties for each algorithm

4.1 For K-Nearest-Neighbors

4.1.1 Problems

Determining the suitable number of K for KNN is always a tough problem. When we choose k, if the value of k is too small, it is very noise-sensitive, which means that the given model can be underfitting, which gives inaccurate results on both training and testing data. If K is too large then it may lead to overfitting.

Deciding the suitable distance/similarity measure is always a tough challenge, it requires domain knowledge to figure out the suitable measure to use.

KNN Method at core is also known as the "lazy learning" algorithm because it takes no time to train the model. All it does is store the training data and will be brought out to compute to find the nearest neighbor to the testing data. And with lots of data, just 60000, the problem has proven to be taking very long to calculate and predict the label of the digits. In case of adding more data (in this case more digits) to the training data, the model will take more time to compute, making it very hard to use in real-life practice because of its computing speed.

4.2 For Artificial Neural Network

4.2.1 Problems

Firstly, the dataset does not fully represent all kinds / special variances of numbers. When we run the test, the numbers 1, 2, 4, 7, and 9 are the ones that have the most incorrect predictions. We have done some error analysis, and this is what we've found.

For formal number recognition such as reading license plates number 1, and 2 in the datasets are enough, however, number 1, 2, and 4 in the datasets doesn't contain all of their variance in terms of different ways of writing it. For number 7, the diagonal line seems to be very similar to that feature of number 2, so number 7 often gets mispredicted with number 2. Number 9, the lower half is very similar to number 3, while the images of number 9 in the dataset are drawn very weirdly, and don't have any common ways that most people in the world draw it, so the model cannot learn how to predict the common 9 with the curvy lower half with those given in the datasets.

Although it's very powerful and capable of outperforming various machine learning algorithms, ANN is very sensitive to "variant" images, meaning that slight changes in the rotation, width, and zoom of the image can result in wrong predictions. Not just the complex structure of the dataset, but images have some special features, which are spatial information, which means that the pixels that are adjacent to each other have some digital connection between them. When we flattened out our 28 x 28 images into a 784-sized vector, we destroyed that relationship / spatial information.

4.2.2 Proposed solutions

To resolve the problem of misprediction numbers 1, 2, 4, 7, and 9, we will augment more data from our own, and then retrain the neural network with an augmented dataset.

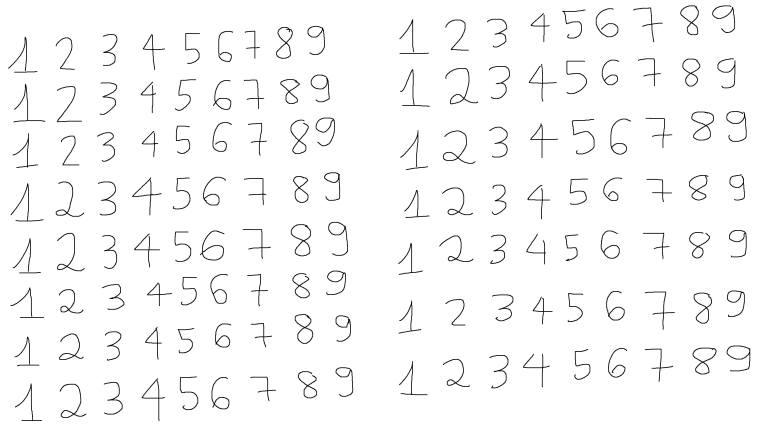


Figure 14: One of our three augmented images of hand-written digits

Each augmented image are processed by inverting images into a black background, and white hand-written digits, and dilating it so that it can increase the thickness of the digits. The images then were processed by using our image processing algorithm, being saved as a tabular data frame with 785 columns, the first 784 columns represent the pixels of each digit, the last column is the label value of that digit.

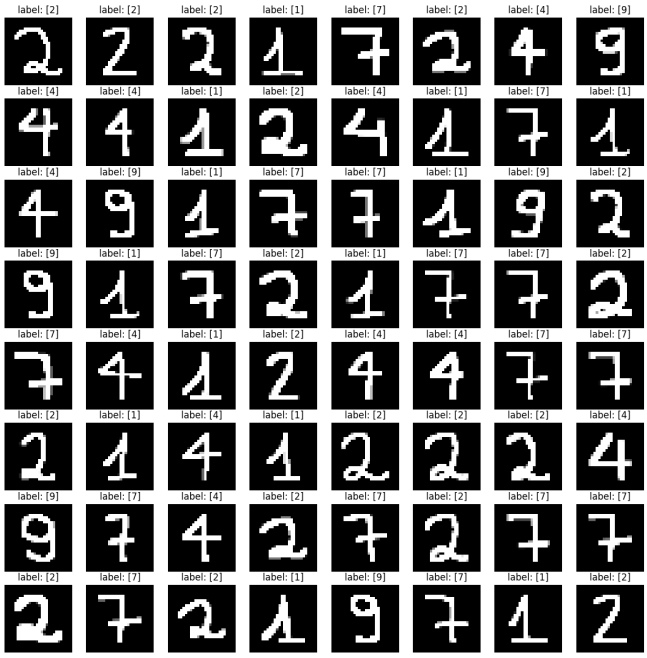
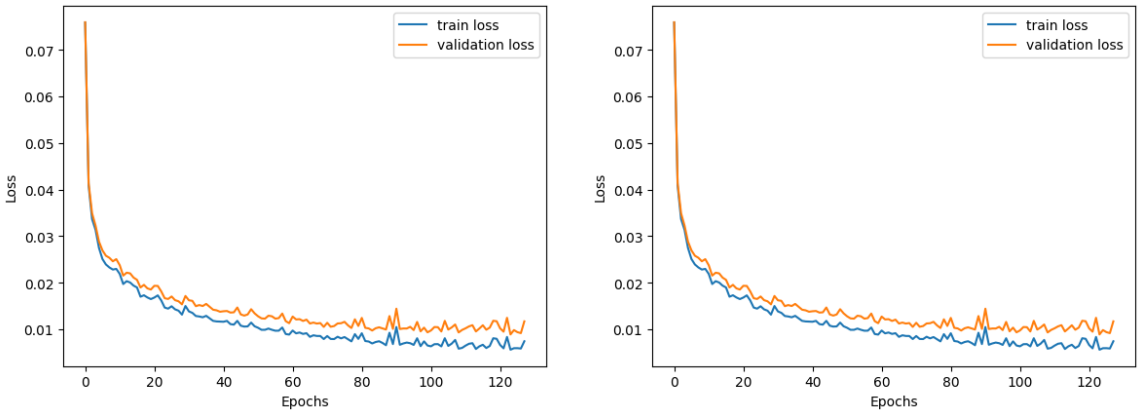


Figure 15: Augmented data after being processed

We will then duplicate this augmented data 2 or 3 times to increase its influence on the training process. After that, we will concatenate this data with the training data. Then we will train the neural network as normal.

Here are the results of the newly trained neural network with data augmentation using the same architecture concluded above:



(a) Accuracy (b) Loss

Figure 16: plots of accuracy and loss of new neural network

4.3 For Convolutional Neural Network

4.3.1 Problems

While CNNs implementation showed significant improvements in terms of accuracy on the test set, it still faces various challenges. Firstly, although the recognition system has improved, CNN implementation struggles with variations in scale and image rotation. If the image is rotated, zoomed, or its properties (height/width) are changed, misprediction could still occur. However, it can recognize some slightly rotated images, which ANNs cannot.

As the number of training data of MNIST data is not well-sufficient and the data still have some particular problems as mentioned, CNN implementation may be prone to overfitting. Without large datasets, CNN architecture may encounter some difficulties in generalization and the risk of overfitting is noticeable.

Despite its powerful performance, CNN implementation requires expensive computational costs. Without the help of GPUs, the training process can take very long, making the experiment on different architectures difficult. The computational cost increases as the architecture

becomes more complex, hence the selection of parameters/ hyperparameters also becomes a challenge that we need to resolve: choosing an architecture that requires less cost but its performance should be acceptable

4.3.2 Proposed solutions

To address the problem of overfitting, we need more data augmentation. The MNIST dataset is clean, so to make it more translation variance, we need to rotate, zoom, and shift the current images in the dataset a little bit so that this algorithm can work more effectively. However, the process of training is computationally expensive, requiring a long training time, so it will not be done in this capstone project.

5 Final Product

After implementing and testing 3 different approaches to solve the problem, now we can gather it up and deploy it on a simple GUI.

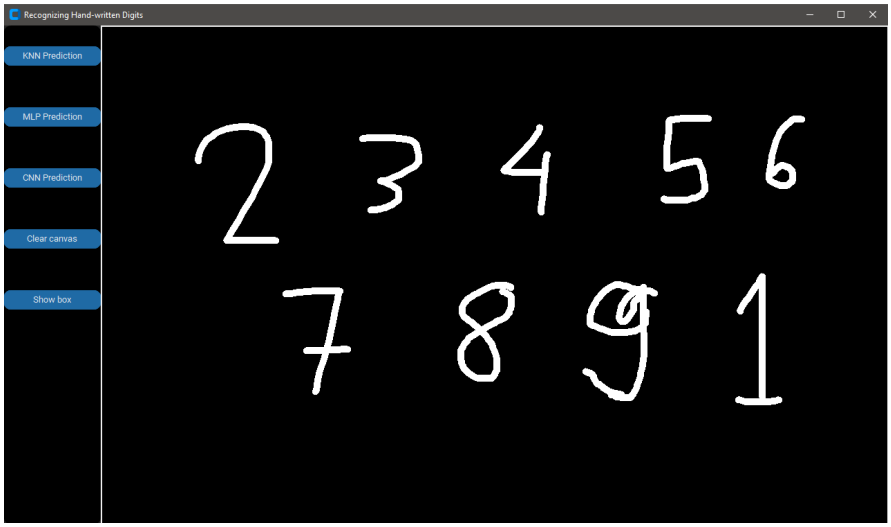


Figure 17: Simple GUI

5.1 General Description

The deployment of 3 approaches will be on a simple GUI. which has 5 buttons. 3 buttons for each of the different approaches to prediction. 1 button to display the bounding box, to locate digits drawn on the screen, and 1 button to clear the canvas.

5.1.1 Implementation

To deploy it, we will save the model structure, then use tkinter and customtkinter to make the simple GUI, the implementation is below:

```

PEN_COLOR = "white"
ct.set_appearance_mode("black")

#Create a new window with fixed resolution
root = ct.CTk()
root.geometry("1280x720")
root.title("Recognizing Hand-written Digits")

#Create a canvas and a sidebar frame for drawing
canvas = ct.CTkCanvas(master=root, width=1000, height=720, bg="black")
canvas.pack(side="right", fill="both", expand=True)

sidebar_frame = ct.CTkFrame(master=root, width=280, corner_radius=10, fg_color="black")
sidebar_frame.pack(side="left", fill="y")
```

Then we will create 5 buttons as mentioned in the general description

```
#First button for KNN prediction
button_1 = ct.CTkButton(master=sidebar_frame, text = "KNN Prediction",
command=recognize_digit_knn, border_width=0, corner_radius=10)
button_1.pack(pady=30)

#Second button for MLP prediction
button_2 = ct.CTkButton(master=sidebar_frame, text = "MLP Prediction",
command=recognize_digit_mlp, border_width=0, corner_radius=10)
button_2.pack(pady=30)

#Third button for CNN Prediction
button_3 = ct.CTkButton(master=sidebar_frame, text = "CNN Prediction",
command=recognize_digit_cnn, border_width=0, corner_radius=10)
button_3.pack(pady=30)

```

```
#Fourth button for Clearing out canvas
button_4 = ct.CTkButton(master=sidebar_frame, text = "Clear canvas",
command=clear_canvas, border_width=0, corner_radius=10)
button_4.pack(pady=30)

#Fifth button for showing bounding box
button_5 = ct.CTkButton(master=sidebar_frame, text = "Show box",
command=show_bounding_box, border_width=0, corner_radius=10)
button_5.pack(pady=30)

canvas.bind("<Button-1>", activate_event)
root.mainloop()
```

5 buttons will be activated by a function, in which they will process the image, feeding it into the chosen model and then a result will be printed out. This is the implementation of one function *regconize_digit_mlp*, other functions such as *recognize_digit_cnn* and *recognize_digit_knn* are similar.

```
def recognize_digit_mlp():
    #Taking necessaries from processing canvas
    processed_img, contours, output_img, unflattened_img = processCanvas(canvas)
    pred = NN.predict(processed_img / 255)

    #Predict each number appeared in the canvas
    for i, rectangle in enumerate(contours):
        left_upper_point = rectangle[1], rectangle[0]
        right_lower_point = rectangle[3], rectangle[2]
        output_str = str(pred[0][i]) + " " + str(round(pred[1][i] * 100, 2)) + "%"
        #Draw rectangle (bounding box) of the predicted digit
        cv2.rectangle(output_img, left_upper_point, right_lower_point, (0, 255, 0), thickness=5)

        #Draw the prediction and the probability
        font = cv2.FONT_HERSHEY_SIMPLEX
        fontScale = 0.7
        color = (0, 255, 0)
        thickness = 2
        cv2.putText(output_img, output_str, (rectangle[1], rectangle[0] - 5),
        font, fontScale, color, thickness)

```

```
22      #Display the prediction
23      cv2.imshow('prediction by MLP', output_img)
24      cv2.waitKey(0)
25      print("Digit predicted using MLP")
```

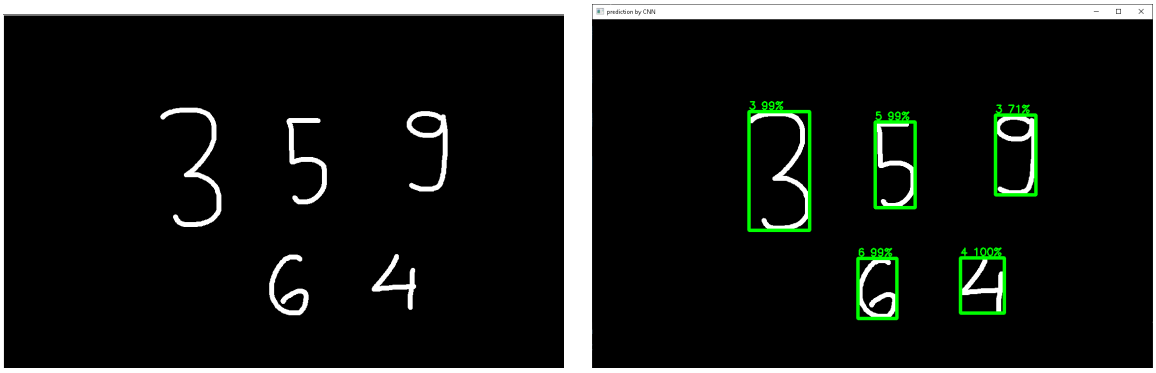


Figure 18: Images before and after being predicted

5.2 How to use

After downloading the demo.rar file. You need to extract it and run the file named "demo".

5.3 Source Code

- For KNN, we conducted on Google Colab notebook, which can be accessed from here: `https://colab.research.google.com/drive/1jNcvfHSw5xKVk2JjFjq9tt6Y-VzXYl8F`
- For MLP (ANN), we conducted and coded our neural network class on Google Colab notebook, which can be accessed here: `https://colab.research.google.com/drive/1PTuHXEhvC-rILLk3ad6r8xqJpfYgVBXc?usp=sharing`
- For CNN, we also conducted our experiments on Google Colab notebook, which can be accessed here: `https://colab.research.google.com/drive/1j-Nzg451h9teuTxJOrLj9n_TCxE4Llxq#scrollTo=DnUiutRD_WQY`

6 Conclusion

In conclusion, we have conducted our experiments and tried to solve the problem by 3 different approaches: K-nearest neighbors, Multi-layer perceptron, and Convolution Neural Network. Those approaches seem to be the most common and simple methods to solve spatial recognition problems. With some advanced image processing technique implementation, our

system hopefully can detect and recognize digits from real-world images.

Thank you, Associate Professor Than Quang Khoat for reading until the end of our group's Capstone project report. If there are any mistakes, or misconceptions in the report, we would like to hear the feedback, and comments from you on our group's report.

References

[1] GitHub - TomSchimansky/CustomTkinter: A modern and customizable python UI-library based on Tkinter — github.com. https://github.com/TomSchimansky/CustomTkinter.

[2] How to choose CNN Architecture MNIST — kaggle.com. https://www.kaggle.com/code/cdeotte/how-to-choose-cnn-architecture-mnist.

[3] PYPL PopularitY of Programming Language index — pypl.github.io. https://pypl.github.io/PYPL.html. [Accessed 15-12-2023].

[4] Understand and Implement the Backpropagation Algorithm From Scratch In Python - A Developer Diary — adeveloperdiary.com. https://www.adeveloperdiary.com/data-science/machine-learning/understand-and-implement-the-backpropagation-algorithm-from-scratch-in-python/?fbclid=IwAR3Y0jECpa3TFxpfaxJbjH9YfsnWAk0DXk6WxkaeSx0HK9sXe77rRcJoPMs. [Accessed 15-12-2023].

[5] Understanding and implementing Neural Network with SoftMax in Python from scratch - A Developer Diary — adeveloperdiary.com. https://www.adeveloperdiary.com/data-science/deep-learning/neural-network-with-softmax-in-python/?fbclid=IwAR2gUk5waf9uZA5_v4Y3UrQtWpUcrKzEFEOCHqzsNIrAHbZwlMtJDDxNX9Q. [Accessed 15-12-2023].

[6] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Watten-

berg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[7] T. E. Bettilyon. How to classify MNIST digits with different neural network architectures — medium.com. https://medium.com/tebs-lab/how-to-classify-mnist-digits-with-different-neural-network-architectures-39c75a0f03e3.

[8] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[9] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[10] A. Clark. Pillow (pil fork) documentation, 2015.

[11] L. Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[12] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020.

[13] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[14] B. Kumar. Convolutional Neural Networks: A Brief History of their Evolution — medium.com. https://medium.com/appyhigh-technology-blog/convolutional-neural-networks-a-brief-history-of-their-evolution-ee3405568597.

[15] F. Lundh. An introduction to tkinter. *URL: www. pythonware. com/library/tkinter/introduction/index. htm*, 1999.

[16] W. McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

[17] A. Mucherino, P. J. Papajorgji, and P. M. Pardalos. *k-Nearest Neighbor Classification*, pages 83–106. Springer New York, New York, NY, 2009.

[18] T. pandas development team. pandas-dev/pandas: Pandas, Feb. 2020.

[19] J. Schmidhuber. Annotated history of modern ai and deep learning, 2022.

[20] P. Umesh. Image processing in python. *CSI Communications*, 23, 2012.

[21] M. L. Waskom. seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021.