HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION COMMUNICATION TECHNOLOGY

Capstone project report:

Hand-written digits recognition system

Supervised by:

Associate Professor Thân Quang Khoát

Group members:

Đỗ Khánh Nam (**L**)	20225988
Lê Thanh Tùng	20226070
Nguyễn Đức Anh	20235890
Võ Phú Lộc	20215221

Hanoi

2025

Contributions

- **Đỗ Khánh Nam (60%)**
  - Implemented the core codebase, conducted experiments and evaluations. (40%)
  - Wrote the *Experiment and Results*, *Discussion*, and *Conclusion* sections of the report. (20%)
- **Lê Thanh Tùng (15%)**
  - Wrote the *Introduction* section of the report. (10%)
  - Formatted and edited the report using LaTeX. (5%)
- **Nguyễn Đức Anh (15%)**
  - Wrote the *KNN background* section of the report. (10%)
  - Created the presentation slides. (5%)
- **Võ Phú Lộc (10%)**
  - Wrote the *CNN background* section of the report. (10%)

Table of Contents

1 Introduction

Handwritten digit recognition is a fundamental problem in the field of computer vision and machine learning, serving as a benchmark task to evaluate classification algorithms and models. It involves the automatic identification of digits from 0 to 9, written by hand, which can vary greatly in style and form from person to person. The ability to accurately recognize handwritten digits has practical applications in areas such as reading postal codes, processing bank checks, and digitizing handwritten forms. This project aims to tackle the handwritten digit recognition problem using two prominent machine learning approaches. **K-Nearest Neighbors** (KNN) and **Convolutional Neural Networks** (CNN).

KNN is a simple, yet effective, non-parametric classification algorithm that assigns a class to a sample based on the majority label of its k closest neighbors in the training set. Due to its intuitive nature and ease of implementation, KNN serves as a good baseline for evaluating more advanced models. However, it is often sensitive to the dimensionality of data and may struggle with scalability as the dataset grows.

In contrast, CNNs represent a class of deep learning models specifically designed to work with grid-like data such as images. By learning hierarchical representations through convolutional layers, CNNs can automatically extract relevant features from images without the need for manual feature engineering. This makes CNNs particularly effective for image classification tasks, including handwritten digit recognition.

In this project, we use the popular MNIST dataset, which contains 70,000 grayscale images of handwritten digits, to train and evaluate both KNN and CNN models. The goal is to compare their performance in terms of accuracy, training time, and prediction speed. Through this comparative analysis, we aim to highlight the strengths and limitations of traditional machine learning versus deep learning approaches in the context of image recognition. This report details the methodology, implementation, and evaluation of both models, offering insights into their practical applications and effectiveness in solving the handwritten digit recognition problem.

2 Background

2.1 K-Nearest Neighbors (KNN)

2.1.1 Introduction

K-Nearest Neighbors (KNN) is a fundamental supervised learning algorithm known for its simplicity and effectiveness, especially in classification tasks such as handwritten digit recognition. It belongs to the family of instance-based or "lazy"learning algorithms, meaning it makes predictions by directly comparing new inputs with stored training examples rather than building an explicit model during training. Due to its intuitive nature and ability to perform well with sufficient data, KNN is often used as a baseline for evaluating more complex models.

2.1.2 Core Idea

The central concept of KNN is that similar examples tend to be near each other in the feature space. When given a new sample, the algorithm identifies its *k* closest training examples - called its neighbors - based on a specified distance metric. The predicted label is then determined by the classes of these neighbors. The basic KNN algorithm works as follows, with variations for unweighted and weighted approaches:

- **Compute Distances:** For a given input sample, calculate the distance between it and every sample in the training dataset using a selected distance metric.
- **Find Nearest Neighbors:** Identify the *k* samples with the smallest distances to the input.
- **Aggregate Neighbor Labels:**
  - **Unweighted KNN:** Each of the *k* neighbors contributes equally. The class label is determined by majority vote - the most frequent class among the neighbors is assigned to the input.
  - **Weighted KNN:** Neighbors contribute proportionally to their proximity. Closer neighbors are given higher importance using a weighting function (commonly inverse distance weighting). The predicted class is the one with the highest weighted vote.

2.1.3 Distance Metrics

- **Euclidean Distance (L2)**

  Calculates the straight-line distance between two points (vectors **p** and **q**) in an N-dimensional space. Involves finding the difference for each dimension ($p_i - q_i$), squaring it, summing these squares across all N dimensions, and finally taking the square root.

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

Where:

- N: The number of dimensions (or features) in each vector.
- $\mathbf{p}$: The first vector, often representing the features of one data point.
- $\mathbf{q}$: The second vector, often another data point from the dataset.

- **Manhattan Distance**
  Calculates the distance by summing the absolute differences along each dimension (like moving along city blocks).

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

Where:

- N: The number of dimensions (or features) in each vector.
- $|p_i - q_i|$: The absolute difference between the i-th features.

2.1.4 Weighting in KNN

In weighted KNN, closer neighbors contribute more to the final prediction than those farther away. This is achieved by assigning weights to neighbors based on their distance to the query point. Instead of simply counting each neighbor equally (as in unweighted KNN), their votes are scaled by how "close"they are. In this project, we implement two weighting formulas: inverse distance weighting and Gaussian kernel weighting.

- **Inverse Distance Weighting**

$$w_k = \frac{1}{d_k}$$

The weight of a data point is the inverse of its distance to the query point. Neighbors that are very close (small distance) get high weights. Neighbors that are far away (large distances) get low weights. This method works well when distances are meaningful and quickly reducing the influence of distant points is desirable.

- **Gaussian Kernel Weighting**

$$w_k = \exp\left(-\frac{d_k^2}{2\sigma^2}\right)$$

Where:

- d_k is the distance (e.g., Euclidean) between the user's drawing vector and the k-th closest MNIST image vector found by KNN.
- σ controls how fast the weights drop off with distance.
- w_k is the resulting voting strength (between 0 and 1) for that k-th MNIST neighbor.

This is a smooth weighting function that gives high weights to very close neighbors and drops off exponentially as distance increases. Unlike inverse weighting, it never gives extremely large or undefined weights, even at distance zero. The parameter σ determines how quickly the weights drop off:

- Small σ: only very close neighbors have high weights.
- Large σ: more distant neighbors are still influential.

2.2 Convolutional Neural Network (CNN)

2.2.1 Intuitive Idea

Convolutional Neural Networks (CNNs) represent a groundbreaking advancement in the field of Artificial Intelligence, first introduced in the 1980s by Yann LeCun, a pioneering computer science researcher. This innovation marked a significant milestone in Computer Vision, offering a novel approach to processing and interpreting visual data. The primary inspiration for CNNs stems from the human brain's visual cortex, which efficiently processes complex visual patterns. Unlike traditional Artificial Neural Networks (ANNs) that rely on fully connected layers where every neuron is linked to every neuron in the subsequent layer, CNNs employ specialized convolutional layers to extract hierarchical features from input images. This design makes CNNs particularly effective for tasks involving spatial data, such as the recognition of handwritten digits in our project, where the MNIST dataset serves as the foundation.

2.2.2 Foundation Concepts of Neural Networks

To fully grasp the operational mechanics of CNNs, it is essential to delve into the foundational concepts that underpin neural networks, as these principles are integral to the functioning of CNN architectures:

- **Backpropagation:** This is the cornerstone training algorithm for neural networks, enabling the model to learn from errors. The process involves two key phases: forward propagation, where input data is passed through the network to generate an output, and backward propagation, where the error (difference between predicted and actual output) is propagated backward through the layers to adjust the weights. Backpropagation utilizes the gradient descent method, calculating the gradient of the loss function with respect to each weight. These gradients guide the iterative updates to the weights, aiming to minimize the loss function and enhance the model's predictive accuracy. In the context of CNNs, backpropagation is critical for refining the filters and weights across convolutional and fully connected layers, ensuring the network adapts to the spatial patterns in images like those in the MNIST dataset.

- **Activation Functions:** Activation functions introduce non-linearity into the neural network, allowing it to model complex relationships within the data that linear models cannot capture. In CNNs, the ReLU (Rectified Linear Unit) function is widely adopted for hidden layers due to its simplicity and effectiveness. Defined mathematically as $f(x) = \max(0,x)$, ReLU sets all negative input values to zero while preserving positive values, which helps mitigate the vanishing gradient problem—a common issue in deep networks where gradients become too small to propagate effectively during training. This property accelerates convergence and enhances the network's ability to learn intricate features.

  For the output layer, the Softmax function is employed, transforming raw scores (logits) into probabilities that sum to one, making it ideal for multi-class classification tasks. This function ensures that the output of the CNN can be interpreted as the likelihood of each digit class, facilitating accurate classification. The Softmax formula is given by:

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

  Where:

  - z_i represents the input to the i-th node

- N is the total number of classes

- **Loss Function:** The loss function quantifies the discrepancy between the network's predictions and the true labels, serving as the objective to minimize during training. CNNs typically employ the Cross-Entropy Loss, which is particularly well-suited for classification problems with multiple categories. This loss function is highly effective because it penalizes confident but incorrect predictions more severely, driving the network to refine its understanding of the data distribution. In the context of handwritten digit recognition, Cross-Entropy Loss ensures the CNN learns to distinguish subtle differences between similar-looking digits, such as 7 and 2 or 9 and 3. The Cross-Entropy Loss is mathematically expressed as:

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

  Where:

  - y_{ij} is the true label (1 if the j-th class is correct for the i-th sample, 0 otherwise)
  - p_{ij} is the predicted probability that sample i belongs to j-th class
  - N is the number of samples, and C is the number of classes (10 in this project)

2.2.3 Architecture of CNN

The architecture of a CNN is built around several distinctive components that set it apart from traditional neural networks:

- **Convolution:** The convolution operation is a mathematical process that merges two sets of information, acting as a feature extractor in CNNs. This operation involves sliding a small filter (or *kernel*)—a matrix of learnable parameters—over the input image to produce feature maps.

  Each element in the filter is adjusted during training, allowing the network to detect low-level features like edges and textures in the early layers, and more complex patterns like shapes or digit structures in deeper layers. The size and number of filters are critical design choices, influencing the network's ability to capture relevant spatial information from the 28
  $times$28 pixel images in the MNIST dataset.

- **Pooling:** Pooling, often referred to as subsampling, is a dimensionality reduction technique that decreases the number of parameters and computations in the network while preserving the

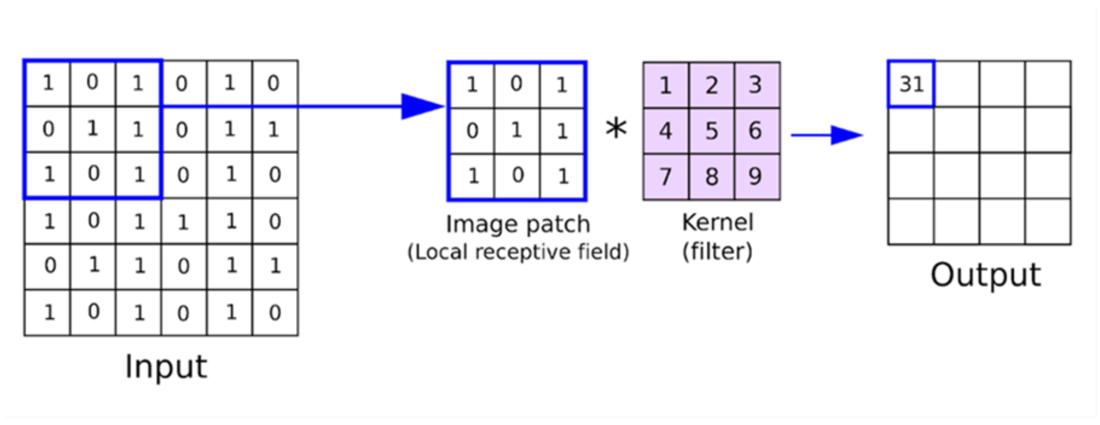

Figure 1: Structure of a convolutional neural network (Source: Anh H. Reynolds)

most salient features. This process helps prevent overfitting by reducing spatial resolution. Two common pooling methods are max pooling, which selects the maximum value within a defined region, and average pooling, which computes the average value. In this project, we adopt max pooling, which emphasizes the most prominent features (e.g., the darkest pixel in a region), enhancing the network's focus on significant patterns in the handwritten digits. This choice aligns with the need to maintain robustness against minor variations in digit appearance.

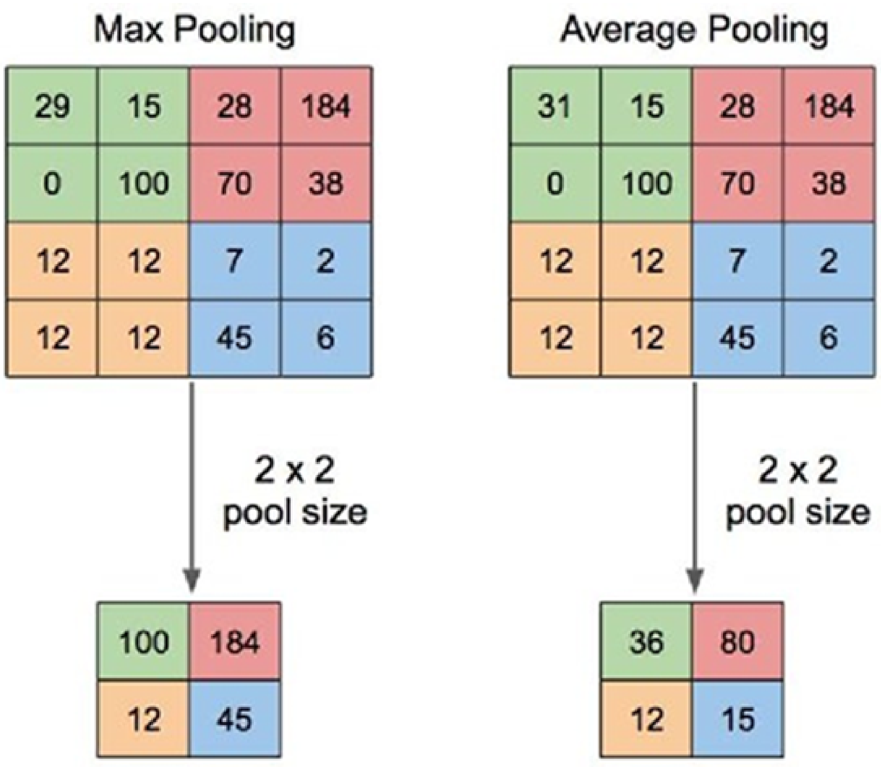

Figure 2: Example of 2×2 pooling layers with stride = 2 (Source: Medium)

The structural design of a CNN typically begins with an input layer that accepts images of 28x28 pixels from the MNIST dataset, which consists of grayscale handwritten digits. This is followed by a sequence of convolutional and pooling layer pairs, where each convolutional layer applies filters to extract features, and each pooling layer reduces the spatial dimensions. After several such pairs, the feature maps are flattened into a one-dimensional vector through a flattening process. This vector is then fed into fully connected layers, which perform high-level reasoning to produce the final classification. The output layer comprises 10 nodes, corresponding to the 10 possible digit classes (0 to 9), and employs the Softmax activation function to output a probability distribution over these classes. This architecture leverages the spatial hierarchy of features—starting from edges and progressing to entire digit shapes—making CNNs exceptionally suited for image-based tasks like ours.

3 Experiment and results

3.1 Dataset and data processing

The dataset utilized in this project is the MNIST dataset, a widely recognized benchmark in the field of image recognition and machine learning. It comprises a total of 70,000 grayscale images of handwritten digits ranging from 0 to 9, divided into 60,000 training images and 10,000 testing images. Each image is 28×28 pixels in size, resulting in 784 features per image when flattened into a vector. The pixel values range from 0 to 255, where 0 represents black (the background) and 255 represents white (the stroke of the digit), with intermediate values indicating varying levels of brightness. The images have been centered within a bounding box and anti-aliased to enhance clarity and consistency. MNIST is particularly suitable for evaluating classification algorithms due to its moderate complexity and well-structured format. In this project, it serves as the primary dataset for developing and testing both traditional methods such as K-Nearest Neighbors (KNN) and deep learning models like Convolutional Neural Networks (CNN).

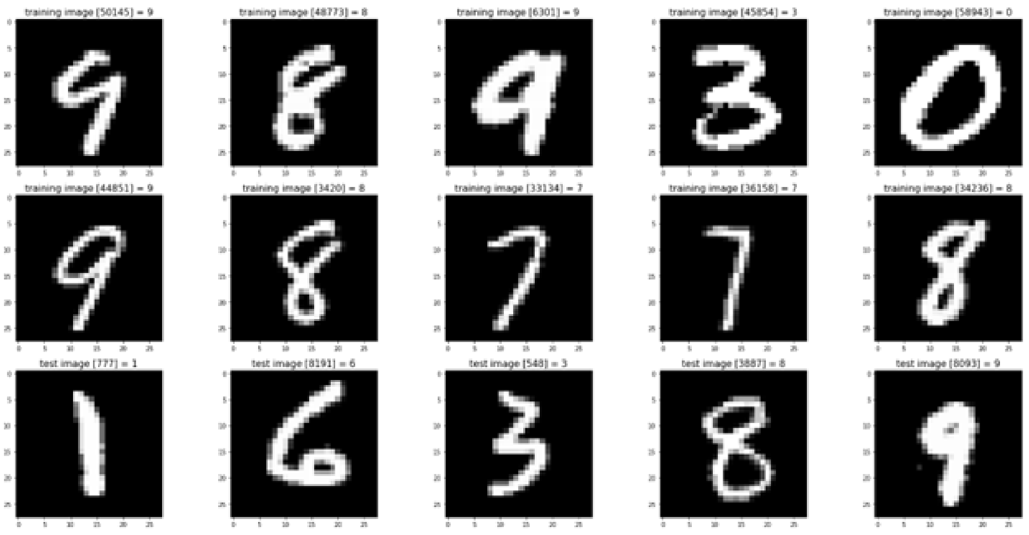
training image [50145] = 9
training image [48773] = 8
training image [6301] = 9
training image [45854] = 3
training image [58943] = 0
training image [44851] = 9
training image [3420] = 8
training image [33134] = 7
training image [36158] = 7
training image [34236] = 8
test image [777] = 1
test image [8191] = 6
test image [548] = 3
test image [3887] = 8
test image [8093] = 9

Figure 3: MNIST dataset examples

In this project, the data processing steps varied slightly depending on the algorithm used. For the K-Nearest Neighbors (KNN) model, the MNIST dataset was simply divided into 60,000 training samples and 10,000 test samples, without further splitting. The images were normalized by scaling pixel values from the original range of 0–255 to a range of 0–1, which helps improve distance-based calculations by reducing the impact of absolute pixel intensity values.

For the Convolutional Neural Network (CNN) model, the same 60,000 training samples were further split into 45,000 for training and 15,000 for validation, allowing the model's performance to be monitored and tuned during training without touching the test set. In addition to normalization, the labels for CNN were also converted using one-hot encoding, transforming each digit label into a binary vector of length 10, where only the index corresponding to the true class is set to 1 and all others to 0. This encoding is essential for training with classification loss functions such as categorical cross-entropy. These preprocessing steps ensure that both KNN and CNN models receive well-formatted, standardized inputs, enabling fair and effective training and evaluation.

3.2 KNN

To evaluate the performance of the K-Nearest Neighbors (KNN) algorithm on handwritten digit classification, we conducted experiments using the MNIST dataset. We tested multiple configurations of KNN by varying the number of nearest neighbors (k) from 1 to 20, and explored the impact of different distance metrics (Euclidean and Manhattan) and weighting schemes (Uniform, Inverse Distance, and Gaussian kernel).

3.2.1 Experiment Setup

Each image from the MNIST dataset was flattened into a 784-dimensional vector and normalized to scale pixel values between 0 and 1. For each configuration, the model was trained using the standard 60,000 training samples and evaluated on the 10,000 test samples.

Three weighting strategies were examined:

- **Uniform weighting:** All neighbors contribute equally.
- **Distance weighting:** Closer neighbors have more influence (weight $\propto \frac{1}{d}$).
- **Gaussian kernel weighting:** Weights decrease exponentially with squared distance, with $\sigma = 1$.

3.2.2 Results with Euclidean Distance

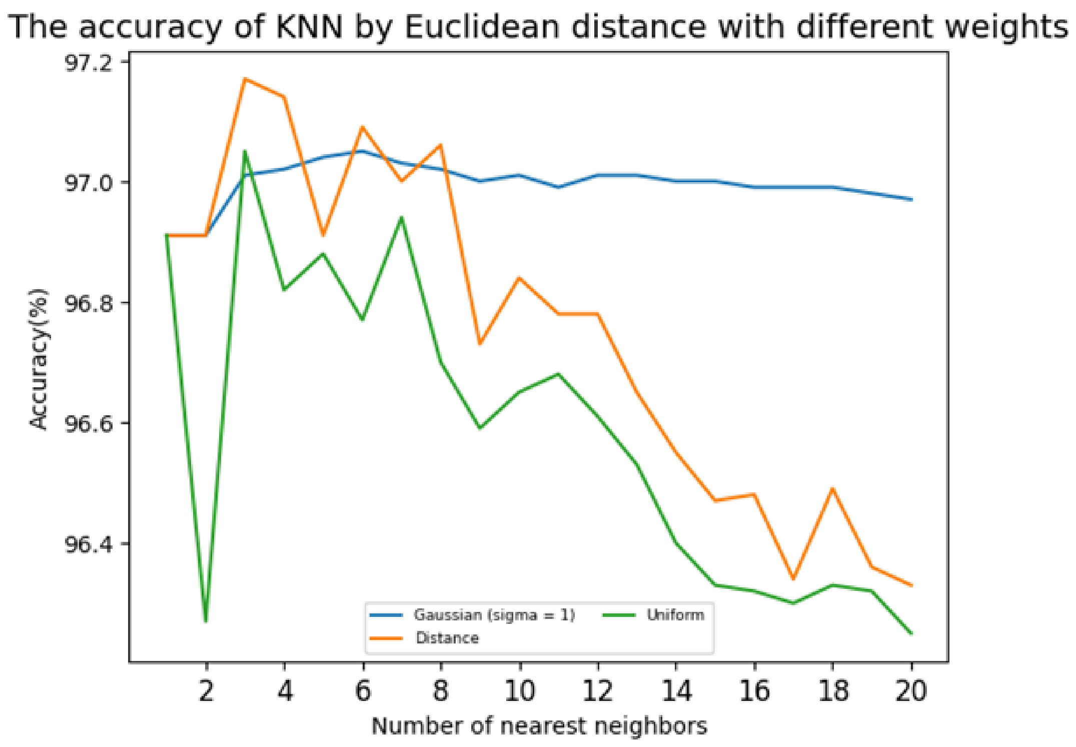

Figure 4: The accuracy of KNN by Euclidean distance with different weights

As shown in the figure, the accuracy for all three weighting strategies remains consistently high, with the distance-based weighting yielding the highest accuracy at lower k values (peaking at 97.17% around $k = 3$). The Gaussian kernel approach offers stable performance across different values of k, though slightly lower than distance-based weighting. The uniform weighting shows a gradual decline in accuracy as *k* increases, suggesting that treating all neighbors equally becomes less effective as more distant, less relevant points are considered.

3.2.3 Results with Manhattan Distance

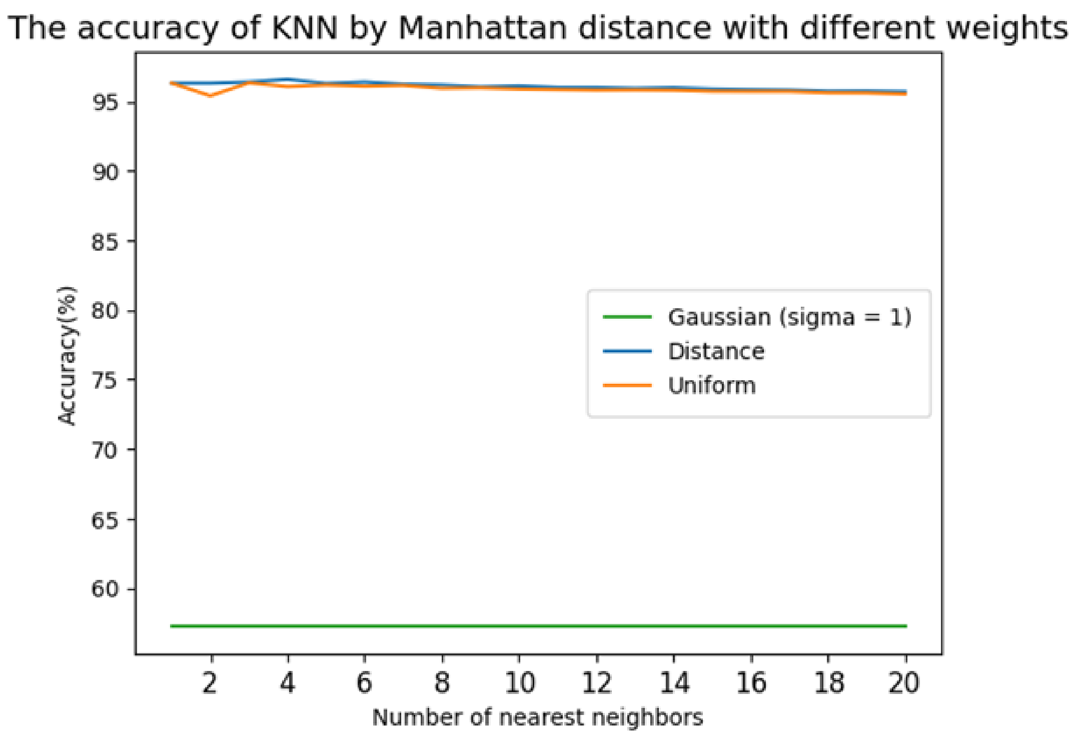

Figure 5: The accuracy of KNN by Manhattan distance with different weights

In contrast, the results with Manhattan distance reveal a significant issue with the Gaussian kernel weighting, which consistently underperforms—achieving accuracy around 56%, far below the other methods. This may be due to the incompatibility of the Gaussian kernel's exponential decay with the Manhattan metric in high-dimensional space. Both uniform and distance-based weighting perform similarly and maintain high accuracy above 95%, indicating robustness across varying *k*.

3.3 CNN

To improve classification performance beyond traditional methods, we implemented a Convolutional Neural Network (CNN) for recognizing handwritten digits from the MNIST dataset. Our goal was to design a model that balances accuracy and computational efficiency while minimizing overfitting. Several model architecture parameters were explored, including:

- **Number of convolution–pooling pairs:** We tested architectures with 1 to 3 blocks of convolution followed by max-pooling. More blocks allow the network to extract more abstract features, but also increase computational cost and risk of overfitting.
- **Number of filters:** The number of filters in each convolutional layer was varied (e.g., 16, 32, 64). More filters enable the model to detect a wider variety of features, but increase the number of parameters.
- **Dense layer configuration:** We examined different numbers of neurons in the fully connected (dense) layers (e.g., 128, 256) to balance model capacity and training time.
- **Dropout rate:** Dropout was applied after the dense layer with varying rates (e.g., 0.2, 0.3, 0.5) to reduce overfitting by randomly deactivating neurons during training.

All models were compiled using the Adam optimizer with categorical cross-entropy loss, and evaluated using the accuracy metric. The models were trained on the training dataset for 32 epochs with a batch size of 32, and validated on a held-out validation dataset.

3.3.1 Convolution and Pooling Pairs

We conducted a series of experiments to evaluate how the number of convolution–pooling (Conv–Pool) pairs impacts model accuracy and generalization. Each Conv–Pool pair consists of a convolutional layer followed by a max pooling layer, which together extract increasingly abstract features while reducing the spatial dimensions of the data.

Experimental Setup: We implemented and trained three different CNN architectures, each containing a different number of Conv–Pool blocks (from 1 too 3). All models shared the following configuration:

- The first convolutional layer used 32 filters with a 5×5 kernel, ReLU activation, and 'same' padding. The second and third convolutional layers (if present) used 48 and 64 filters respectively, also with 5×5 kernels and ReLU activation.
- Each Conv layer was followed by a 2×2 MaxPooling layer to downsample the feature maps.
- After the convolutional layers, all models include a dense layer with 256 neurons (ReLU activation), and a final dense output layer with 10 neurons and softmax activation for digit classification.

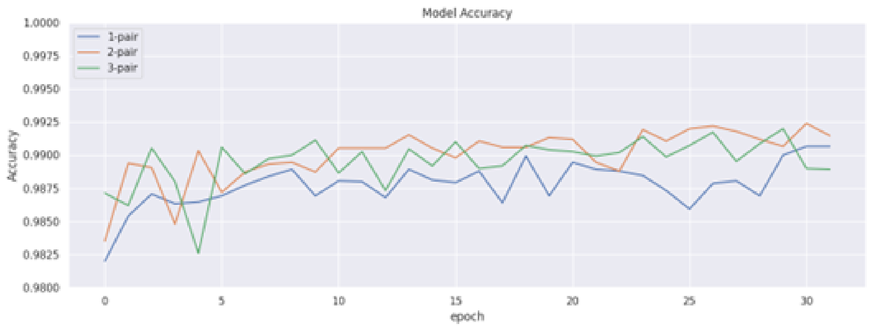

Figure 6: Accuracy of CNN with different Conv–Pool pair configurations

Based on the empirical results, we selected the 2 Conv–Pool pair configuration as the optimal architecture for subsequent experiments. It demonstrated strong generalization, high accuracy, and computational efficiency, making it the most suitable trade-off for our handwritten digit recognition task.

3.3.2 Number of Filters in Convolutional Layers

Another important hyperparameter in designing a Convolutional Neural Network (CNN) is the number of filters in each convolutional layer. Filters (also called kernels) determine how many distinct features a layer can extract from the input. A higher number of filters allows the network to learn more diverse and complex patterns but also increases the number of parameters, potentially leading to longer training times and overfitting. To identify an effective filter configuration, we conducted an experiment where we trained five different CNN models, each with a different pair of filter sizes for the two convolutional layers. All other architectural components and training parameters were kept constant to ensure a fair comparison. The following filter configurations were tested:

Model	**Conv layer 1 filter**	**Conv layer 2 filter**
0	16	32
1	24	48
2	32	64
3	48	96
4	64	128

All models used a kernel size of 5×5 and ReLU activation for both convolutional layers. Each convolutional layer was followed by a max pooling layer to reduce spatial dimensions. After the convolutional stages, all models included a Flatten layer, a fully connected dense layer with 256 neurons (ReLU), and an output layer with 10 neurons and softmax activation.

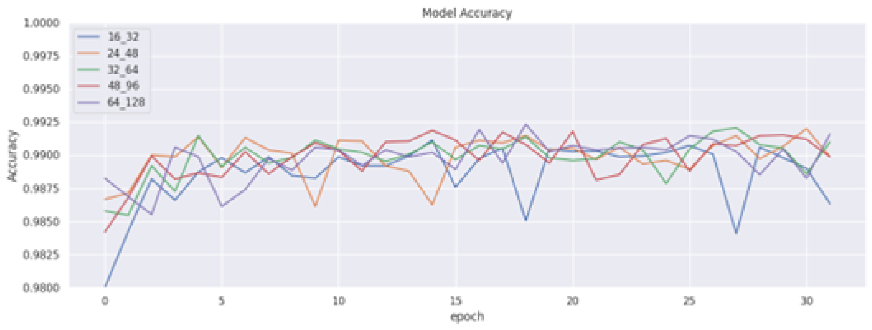

Figure 7: Model accuracy across different filter configurations

Based on the experimental results, all filter configurations achieved high accuracy on the validation set, with minimal differences between them. However, the model using 64 filters in the first convolutional layer and 128 filters in the second demonstrated consistently strong performance across epochs, with stable accuracy and minimal fluctuation. Although this configuration involves a higher number of parameters compared to smaller filter sets, it offers the most robust learning capacity, enabling the network to extract richer and more diverse features from the input images. Given the relatively small size of the MNIST dataset and the availability of computational resources, the 64–128 filter configuration was

3.3.3 Number of Neurons in the Dense Layer

The dense layer (also known as the fully connected layer) plays a critical role in combining the spatial features extracted by the convolutional layers to perform classification. The number of neurons in this layer determines the model's capacity to learn complex patterns and decision boundaries. However, an excessively large dense layer can lead to overfitting and increased computational cost, while a small dense layer may lack the capacity to generalize effectively.

To find the optimal number of neurons in the dense layer, we designed and trained six CNN models that differ only in the size of the dense layer. The dense layer sizes tested were: 32, 64, 128, 256, 512, 1024. All models used the same convolutional base with 2 convolutional layers with 64 and 128 filters in each layer respectively.

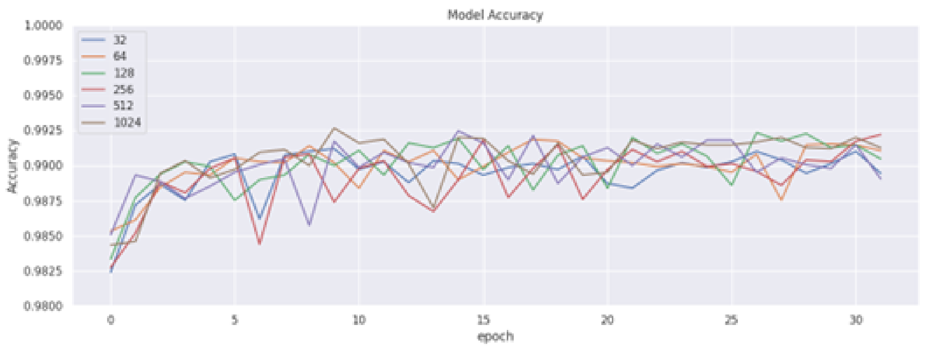

Figure 8: Validation accuracy across dense layer sizes (32–1024)

Smaller sizes (32, 64) showed lower and less stable accuracy, indicating limited capacity to model complex patterns. Larger sizes (512, 1024) performed well but introduced higher computational cost without significant accuracy gain. The 256-neuron dense layer achieved consistently high and stable validation accuracy, making it the most balanced option. It offers sufficient learning capacity while maintaining efficiency and generalization, and was therefore selected for the final CNN architecture.

3.3.4 Dropout Rate

Dropout is a widely used regularization technique in deep learning that helps prevent overfitting by randomly deactivating a fraction of neurons during each training iteration. This forces the model to learn more robust and generalized features rather than relying too heavily on specific neurons. However, an inappropriate dropout rate can either lead to underfitting (if too high) or insufficient regularization (if too low). Therefore, tuning this hyperparameter is essential for achieving strong generalization performance. Dropout is applied to both *conv–pool layer* and *dense layer*. When applied after convolutional and pooling layers, dropout randomly disables individual activations in the 3D output tensor (height $\times$ width $\times$ channels).

Experimental Setup: To determine the optimal dropout rate, we trained six CNN models with dropout rates ranging from 0.0 (no dropout) to 0.5, in increments of 0.1. All models shared the same architecture. The model consists of two Conv2D layers with 64 and 128 filters (5×5, ReLU), followed by MaxPooling, a Dense layer with 256 ReLU neurons, and a 10-neuron softmax output layer.

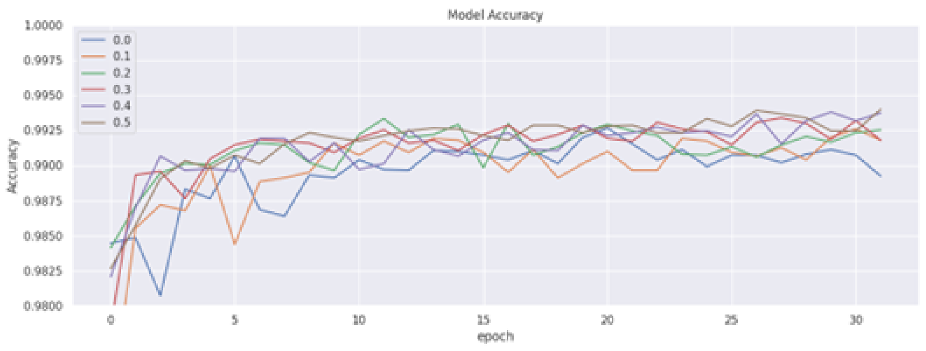

Figure 9: Model accuracy across different dropout rates

Models with low or no dropout (e.g., 0.0-0.1) showed lower validation accuracy and greater fluctuation, indicating a tendency to overfit. Moderate rates (0.2–0.4) performed more consistently, balancing regularization and learning capacity. The model with a 0.5 dropout rate achieved consistently high accuracy with minimal overfitting, especially in later epochs. Although it slightly slows learning early on, it offers strong regularization and generalization. As a result, 0.5 was selected for the final CNN architecture to ensure robust performance on unseen data.

3.3.5 Final Model and Performance

After a series of systematic experiments on architectural parameters—including the number of convolutional blocks, filter sizes, dense layer size, and dropout rates—we designed and trained a final Convolutional Neural Network (CNN) optimized for handwritten digit recognition using the MNIST dataset.

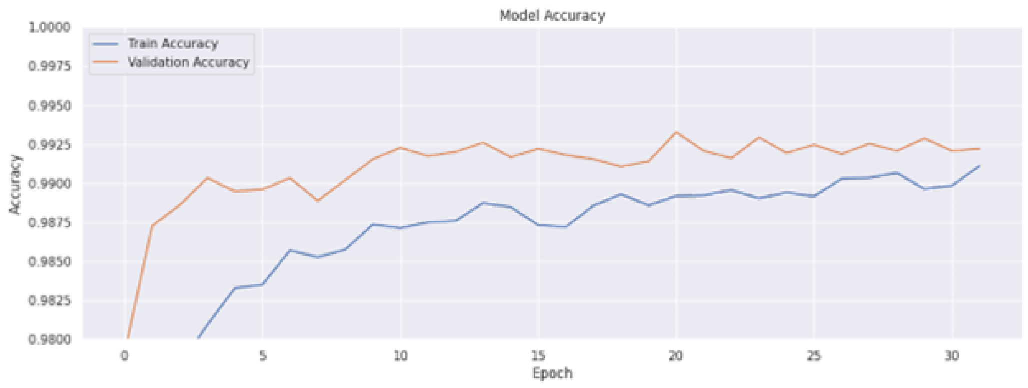

Figure 10: Training and validation accuracy of final model

Table 1: Model Summary

Layer (type)	**Output Shape**	**Param #**
Conv2D (conv2d)	(None, 28, 28, 64)	1,664
MaxPooling2D (max_pooling2d)	(None, 14, 14, 64)	0
Dropout (dropout)	(None, 14, 14, 64)	0
Conv2D (conv2d_1)	(None, 14, 14, 128)	204,928
MaxPooling2D (max_pooling2d_1)	(None, 7, 7, 128)	0
Dropout (dropout_1)	(None, 7, 7, 128)	0
Flatten (flatten)	(None, 6272)	0
Dense (dense)	(None, 256)	1,605,888
Dropout (dropout_2)	(None, 256)	0
Dense (dense_1)	(None, 10)	2,570
Total Parameters		**1,815,050**
Trainable Parameters		**1,815,050**
Non-trainable Parameters		**0**

The model was compiled with the Adam optimizer, using categorical cross-entropy as the loss function and accuracy as the evaluation metric. Training was performed over 32 epochs with a batch size of 32, using the MNIST training set, which was split into training and validation subsets.

The model achieved a validation accuracy consistently above 99%, with a peak accuracy of approximately 99.3%. Throughout the training process, the training accuracy showed steady improvement without any signs of overfitting, indicating the effectiveness of the applied dropout regularization in promoting generalization. The gap between training and validation curves remains stable and small, which indicates that the model generalizes well to unseen data. The dropout layers (with a rate of 0.5) played a significant role in maintaining this generalization by preventing the network from memorizing the training data. To evaluate its performance on unseen data, the model was tested on the MNIST test set, consisting of 10,000 images. It achieved a test accuracy of 0.9939, confirming its excellent generalization and robustness.

3.4 Compare KNN and CNN on Human-Generated Data

To assess the generalization capability of the models beyond the MNIST dataset, both the K-Nearest Neighbors (KNN) and Convolutional Neural Network (CNN) models were evaluated on a set of human-generated handwritten digit samples collected using a custom drawing interface and written by each member of our group. The digits are centered and resized to 28×28 to match the MNIST dataset dimensions. This test set introduces real-world variability such as inconsistent digit size, stroke width, and diverse handwriting styles-factors not present in the clean and normalized MNIST dataset.

KNN Model (k = 3, inverse distance weighting)

The KNN classifier was trained on the full MNIST training set using 3 neighbors and inverse distance weighting, which prioritizes closer samples during classification. When evaluated on the human-generated data, the model achieved an accuracy of26%, significantly lower than its performance on MNIST. This suggests that KNN, being a non-parametric method that heavily depends on the structure and distribution of the training data, struggles with out-of-distribution data and lacks the capacity to generalize to novel handwriting styles.

CNN Model (784–Conv64–Conv128–Dense256–10, Dropout = 0.5)

This convolutional neural network (CNN) model achieved an accuracy of 82% on the same human-generated handwriting dataset, significantly outperforming the k-nearest neighbors (KNN) baseline. This notable improvement highlights the CNN's superior capacity to learn hierarchical spatial features from image data, enabling it to capture complex patterns and variations inherent in handwritten digits. Furthermore, the incorporation of dropout regularization during training effectively mitigated overfitting, enhancing the model's generalization capability. These results suggest that even when trained exclusively on the MNIST dataset, the CNN is able to adapt and perform well on previously unseen handwriting styles, underscoring its robustness and transferability across different but related data distributions..

4 Discussion

The results of this project reveal a clear distinction in both the performance and practicality of the K-Nearest Neighbors (KNN) and Convolutional Neural Network (CNN) approaches for handwritten digit recognition, particularly when evaluated on real-world, human-generated data.

While both models achieved high accuracy on the standardized MNIST dataset (above 97%), their performance diverged significantly when tested on user-drawn digits collected through a custom interface. The KNN model with k = 3 and inverse distance weighting achieved only 26% accuracy, whereas the CNN model reached 82% on the same dataset.

This discrepancy highlights the limitations of KNN in handling data that differs even slightly from the training distribution. KNN relies heavily on the closeness of examples in the training set, making it highly sensitive to variations in handwriting style, alignment, and stroke quality. In contrast, the CNN model benefits from its ability to learn spatially-invariant and hierarchical features, allowing it to generalize better to digits that deviate from the training data in terms of position, size, and shape.

Another key difference lies in the computational cost of training each model. Although KNN is a lazy learner and does not require traditional training, it demands substantial resources during the prediction phase. In this project, training and preparing the KNN model for MNIST took approximately 3 hours 34 minutes, primarily due to data preprocessing and distance computations over a large dataset. On the other hand, the CNN model required approximately 16 hours of GPU time on Kaggle to find optimal architecture and 1 hour 46 minutes to train the final model, completing 32 epochs. Although more time-consuming, the CNN benefits from rapid inference once trained, making it more scalable and efficient for real-time or batch predictions.

To further enhance performance on real-world data and make the system more applicable in practice, several improvements are recommended:

- **Data Augmentation**: Introduce variations such as rotations, shifts, scaling, and noise during training to better simulate human-drawn digits and improve model robustness.

- **Domain-Specific Fine-Tuning**: Combine the original MNIST dataset with a portion of the human-generated data to fine-tune the CNN. This can help bridge the domain gap and significantly improve generalization.

- **Alternative Architectures**: Consider lightweight, transfer-learned models (e.g., MobileNet, EfficientNet) pre-trained on image tasks and fine-tuned for digits, which may offer higher accuracy with lower training cost.

5 Conclusion

In this project, we optimized and compared two models K-Nearest Neighbors (KNN) and Convolutional Neural Networks (CNN) for handwritten digit recognition. The optimization process involved systematic experimentation with various hyperparameters for each model.

For KNN, we tested multiple values of k, different distance metrics (Euclidean and Manhattan), and weighting strategies (uniform, inverse distance, and Gaussian). The best performance was achieved with $k = 3$, inverse distance weighting, and Euclidean distance. For CNN, we experimented with the number of convolution–pooling blocks, filter sizes, dense layer sizes, and dropout rates. The optimal CNN architecture consisted of two Conv–Pool blocks (64 and 128 filters), a 256-neuron dense layer, and a dropout rate of 0.5.

When evaluated on human-generated digit data, the CNN significantly outperformed KNN. The CNN model achieved 82% accuracy, while the KNN model reached only 26% accuracy. This demonstrates CNN's superior ability to generalize to new, real-world handwriting styles. Although the CNN required longer training time (16 hours vs. 3.5 hours for KNN), its inference is much faster and more practical for real use.

To further improve real-world performance, future work should include data augmentation, fine-tuning the model on human-drawn digits. Overall, the CNN model proved to be the more accurate and scalable solution for handwritten digit recognition.

References

[1] Illuri, Lakshmi Teja. "Understanding Weighted k-Nearest Neighbors (k-NN) Algorithm." *Medium*, 26 Sept. 2023. `https://medium.com/@lakshmiteja.jp/understanding-weighted-k-nearest-neighbors-k-nn-algorithm-3485001611ce`

[2] LeCun, Yann, Corinna Cortes, and Chris Burges. "MNIST handwritten digit database."14 Jan. 2010.

[3] LeCun, Yann, and Yoshua Bengio. "Convolutional networks for images, speech, and time series."*The Handbook of Brain Theory and Neural Networks*, 3361.10 (1995): 1995.

[4] LeCun, Yann, et al. "Comparison of learning algorithms for handwritten digit recognition."*International Conference on Artificial Neural Networks*. Vol. 60. No. 1. 1995.

[5] LeCun, Yann, et al. "Handwritten digit recognition with a back-propagation network."*Advances in Neural Information Processing Systems*, vol. 2, 1989.

[6] Cdeotte. “How to Choose CNN Architecture MNIST.” *Kaggle*, 24 Sept. 2018. `https://www.kaggle.com/code/cdeotte/how-to-choose-cnn-architecture-mnist`

[7] Bettilyon, Tyler Elliot. “How to Classify MNIST Digits With Different Neural Network Architectures.” *Medium*, 8 Mar. 2022. https://medium.com/tebs-lab/how-to-classify-mnist-digits-with-different-neural-network-architectures-39c75a0f03e3

[8] “Graphical User Interfaces With Tk.” *Python Documentation*. `https://docs.python.org/3.11/library/tk.html`

[9] “Pillow.” *Pillow (PIL Fork)*. `https://pillow.readthedocs.io/en/stable`

[10] NumPy Documentation. `https://numpy.org/doc`

[11] “Keras: The High-level API for TensorFlow.” *TensorFlow*. `https://www.tensorflow.org/guide/keras`

[12] Pandas Documentation - Pandas 2.2.3 Documentation. `https://pandas.pydata.org/docs`

[13] Using Matplotlib - Matplotlib 3.10.3 Documentation. `https://matplotlib.org/stable/users/index.html`

[14] “User Guide.” *Scikit-learn*. `https://scikit-learn.org/stable/user_guide.html`

[15] User Guide and Tutorial – Seaborn 0.13.2 Documentation. `https://seaborn.pydata.org/tutorial.html`

[16] Tan, Mingxing, Quoc V. Le. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks."*arXiv preprint arXiv:1905.11946*, 2019.

[17] Howard, Andrew G., et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications."*arXiv preprint arXiv:1704.04861*, 2017.