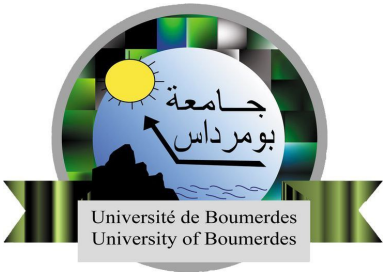
PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH



M'Hamed Bougara University of Boumerdès
Faculty of Science
Department of Computer Science

Dissertation

Submitted for the fulfillment of the degree of :
LICENCE in Computer Science
Specialization: Information Systems

Topic :

Application Web de Reconnaissance de Chiffres Manuscrits (MNIST)

Presented by :
Ladjali Samra
Saibi Hala
habbi yousra
Ben mouhoub Lydia

Before the jury composed of:

Dr. AKROUR Djouher	Supervisor
Dr. Lida Ichallamen	Examiner

Academic Year 2025/2026

Acknowledgements

We would like to express our deepest gratitude to everyone who contributed, directly or indirectly, to the completion of this work.

First and foremost, we would like to extend our sincere appreciation to our supervisor, **Dr. Djouher Akrour**, for her invaluable guidance, insightful recommendations, patience, and continuous support throughout the realization of this project. Her expertise, encouragement, and dedication have been essential in overcoming challenges and successfully accomplishing this work.

We would also like to express our sincere gratitude to the jury member for accepting to evaluate our work and for the time, attention, and constructive feedback provided throughout the assessment process.

Our heartfelt gratitude is addressed to all the professors and staff members of the Department of Computer Science at **M'Hamed Bougara University of Boumerdès** for the knowledge, skills, and academic training they provided throughout our university studies.

Finally, we are deeply grateful to our families and friends for their unwavering support, understanding, encouragement, and confidence in us. Their presence and motivation have been a constant source of strength throughout this academic journey and have greatly contributed to the successful completion of this work.

Thank you all.

Dedication

To our families,
for their love, patience, and unconditional support.

To our teachers,
who shared their knowledge with dedication and passion.

To our friends and colleagues,
for their encouragement, good spirit, and the moments we shared together.

To all those who believe that technology can be used in the service of humanity.

Samra Ladjali · Saibi Hala · Habbi Yousra · Lydia Ben Mouhoub

Abstract

This dissertation presents the design and development of a web application for handwritten digit recognition, named **NeuralDigit Pro**, built on the MNIST dataset. The primary objective is to make machine learning techniques accessible to a general audience through an intuitive interface that requires no technical expertise.

The first part establishes the theoretical foundations of handwritten digit recognition, covering image preprocessing techniques, the transition from classical machine learning to deep learning, and the architecture of Convolutional Neural Networks (CNNs). A state-of-the-art review demonstrated that hybrid CNN-ViT models currently achieve up to 99.97% accuracy on MNIST.

The second part describes the implementation and comparison of four classifiers on the full MNIST dataset (60,000 training images, 10,000 test images): a Convolutional Neural Network (CNN) achieving 99.2% accuracy, a Support Vector Machine (SVM) at 98.1%, a Random Forest at 97.0%, and a k-Nearest Neighbors (k-NN) algorithm at 96.8%. The CNN was selected as the production model due to its superior accuracy and generalization capability.

The third part presents NeuralDigit Pro, a complete web application integrating the CNN model through a Flask REST API. The interface offers two input modes (freehand canvas drawing and image upload), a four-stage preprocessing pipeline visually exposed to the user, Top-3 predictions with Grad-CAM activation maps, an uncertainty indicator, and an administrator dashboard with usage statistics. The application demonstrates the critical importance of preprocessing consistency between training and inference phases, and serves as both a pedagogical tool and a production-grade machine learning deployment reference.

Keywords: MNIST, handwritten digit recognition, convolutional neural network (CNN), Flask, Grad-CAM, deep learning, web interface.

Résumé

Ce mémoire présente la conception et le développement d'une application web de reconnaissance de chiffres manuscrits, baptisée **NeuralDigit Pro**, basée sur le jeu de données MNIST. L'objectif principal est de rendre les techniques d'apprentissage automatique accessibles au grand public à travers une interface intuitive, sans nécessiter de compétences techniques particulières.

Dans un premier temps, nous avons étudié les fondements théoriques de la reconnaissance de chiffres manuscrits, en couvrant les techniques de prétraitement d'images, la transition de l'apprentissage automatique classique vers l'apprentissage profond, et les architectures de réseaux de neurones convolutifs (CNN). Une revue de l'état de l'art a montré que les modèles hybrides CNN-ViT atteignent jusqu'à 99,97% de précision sur MNIST.

Dans un deuxième temps, nous avons implémenté et comparé quatre classificateurs sur l'ensemble du jeu de données MNIST (60 000 images d'entraînement, 10 000 de test) : un Réseau de Neurones Convolutif (CNN) atteignant 99,2% de précision, une Machine à Vecteurs de Support (SVM) à 98,1%, une Forêt Aléatoire à 97,0%, et un algorithme des k Plus Proches Voisins (k-NN) à 96,8%. Le CNN a été sélectionné comme modèle de production.

Enfin, nous avons développé NeuralDigit Pro, une application web complète intégrant le modèle CNN via une API REST Flask. L'interface offre deux modes de saisie (dessin sur canevas et téléversement d'image), un pipeline de prétraitement en quatre étapes visuellement exposé, des prédictions Top-3 avec cartes Grad-CAM, un indicateur d'incertitude, et un tableau de bord administratif. L'application souligne l'importance de la consistance entre les phases d'entraînement et d'inférence, et constitue à la fois un outil pédagogique et une démonstration de déploiement de modèle en production.

Mots-clés : MNIST, reconnaissance de chiffres manuscrits, réseau de neurones convolutif (CNN), Flask, Grad-CAM, apprentissage profond, interface web.

Contents

Acknowledgements i

Dedication ii

Abstract iii

Résumé iv

General Introduction 1

1 Theoretical Framework 2
- 1.1 Introduction . . . 2
- 1.2 Introduction to Handwritten Digit Recognition . . . 2
  - 1.2.1 Problem Description . . . 2
  - 1.2.2 Real-world Applications of Handwritten Recognition . . . 3
- 1.3 Digital Image Fundamentals and Preprocessing . . . 4
  - 1.3.1 Digital Image Representation . . . 4
  - 1.3.2 Basic Preprocessing Techniques . . . 5
- 1.4 From Machine Learning to Deep Learning . . . 5
  - 1.4.1 Classical Machine Learning Approach . . . 5
  - 1.4.2 Artificial Neural Networks (ANN) . . . 6
  - 1.4.3 Convolutional Neural Networks (CNN) . . . 7
- 1.5 Reference Architectures . . . 8
  - 1.5.1 LeNet-5 . . . 8
  - 1.5.2 Moderne Architectures . . . 9
- 1.6 Training Methodology and Evaluation Metrics . . . 9
  - 1.6.1 Data splitting . . . 9
  - 1.6.2 Performance Metrics . . . 10
  - 1.6.3 Overfitting . . . 10
- 1.7 State of the art . . . 10
- 1.8 Conclusion . . . 11

2 Design and Implementation of the Prediction System 12
- 2.1 Introduction . . . 12
- 2.2 System Architecture . . . 12
  - 2.2.1 Global Architecture of the Prediction Pipeline . . . 12
  - 2.2.2 Data Flow . . . 12
  - 2.2.3 Tools and Libraries Used . . . 13
- 2.3 Data Preparation . . . 13

2.3.1 Dataset Description . . . 13
2.3.2 Data Preprocessing: . . . 13
2.3.3 Train/Test Split Strategy . . . 14
2.4 Model Design . . . 14
2.4.1 Convolutional Neural Network (CNN) . . . 14
2.4.2 Support Vector Machine (SVM) . . . 15
2.4.3 Random Forest . . . 15
2.4.4 k-Nearest Neighbors (k-NN) . . . 15
2.5 Model Training . . . 15
2.5.1 CNN Training Process . . . 15
2.5.2 Classical Model Training . . . 16
2.6 Comparative analysis . . . 16
2.7 Conclusion . . . 17

3 Web Application and User Interface 18
3.1 Introduction . . . 18
3.2 Objectives and Application Role . . . 18
3.3 System Overview . . . 19
3.3.1 General Architecture . . . 19
3.3.2 Communication Between Components . . . 19
3.4 Backend Development . . . 20
3.4.1 Flask . . . 20
3.4.2 Model Integration . . . 20
3.4.3 REST API Design . . . 20
3.5 Frontend Development . . . 21
3.5.1 Technologies Used . . . 21
3.5.2 Interface Design . . . 22
3.5.3 User Experience . . . 23
3.6 Image Processing in the Application . . . 23
3.6.1 Preprocessing Pipeline . . . 23
3.6.2 Consistency with Training Data . . . 24
3.7 Integration and Testing . . . 25
3.7.1 Frontend-Backend Connection . . . 25
3.7.2 Functional Testing . . . 25
3.7.3 Performance Considerations . . . 25
3.8 Deployment . . . 26
3.8.1 Local Deployment . . . 26
3.8.2 Limitations and Constraints . . . 26
3.9 Security and Limitations . . . 26
3.9.1 Input Validation . . . 26
3.9.2 Admin Dashboard — Authentication Limitation . . . 27
3.9.3 Model Limitations in Real-World Use . . . 27
3.9.4 Grad-CAM Implementation Note . . . 27
3.9.5 Privacy and Data Retention . . . 28
3.9.6 Recommendations for Robust Deployment . . . 28
3.10 Achievements and future work . . . 28
3.11 Conclusion . . . 29

General Conclusion 30

General Introduction

Handwritten Digit Recognition focuses on converting digits from images into a digital format that computers can process. Unlike processing standardized data, interpreting human handwriting involves significant complexity due to the vast variability in writing styles. This task requires modeling diverse shapes, slants, and stroke thicknesses, necessitating the use of advanced machine learning architectures to accurately decipher individual digits. [1, 2]

Handwritten digit recognition is a common multiclass classification problem, where the goal is to identify a digit from 0 to 9 based on an image of its handwritten form. Each class contains multiple handwritten examples, reflecting the natural variability in human handwriting. This task is challenging because individuals write the same number differently, in terms of size, shape, font, thickness, and style. Furthermore, technological advancements, such as digital writing tools and high-performance computers, have enabled researchers to collect larger datasets and develop more effective identification methods.

Modern approaches, such as Convolutional Neural Networks (CNNs), have demonstrated their ability to automatically learn important features from images, including edges, curves, and the overall shapes of digits. They have been widely used to improve recognition accuracy. Despite this advancement, creating a fully reliable recognition system remains challenging due to the diversity of handwriting styles and potential image quality issues. Research continues to focus on improving the accuracy, robustness, and generalization of these systems.[3]

The objectives of our project are:

- To preprocess the handwritten digit images by applying resizing, normalization, and noise reduction techniques to improve data quality.
- To implement and train traditional machine learning models for handwritten digit classification.
- To design, train, and optimize a Convolutional Neural Network (CNN) model for digit recognition.
- To evaluate and compare the performance of the implemented models using measurable metrics such as accuracy, loss, and confusion matrix.
- To design a user interface in the form of an application or website where users can draw numbers directly and the model detects the drawn number in real time.

Chapter 1

Theoretical Framework

1.1 Introduction

This chapter provides the theoretical basis for handwritten digit recognition. It covers image preprocessing, the transition from traditional Machine Learning to Convolutional Neural Networks (CNNs), and the metrics used to evaluate model performance. This framework justifies the technical choices made in the implementation phase.

1.2 Introduction to Handwritten Digit Recognition

Handwritten Digit Recognition focuses on converting digits from images into a digital format that computers can process. This reduces manual work, saves time, and improves data processing accuracy.[4]. However, this requires the use of advanced machine learning architectures to accurately decipher individual digits. [1, 2]

1.2.1 Problem Description

Unlike printed text, handwriting varies greatly from one person to another and is influenced by many personal and external factors. This variability, combined with image noise and similarities between certain characters, makes automatic recognition a really challenging task. Therefore, building an accurate system requires addressing several key difficulties.[5]

Variability in handwriting

Such as stroke thickness, writing speed, slant and pressure. This produces different shapes for the same character. In fact, this leads to high intra-class variation, meaning the same character may look very different across writers. [6]

Noise and Distortion

Input images are often affected by noise introduced during scanning or photographing. Smudge, blur, low resolution, shadow, or paper degradation can reduce image clarity and make feature extraction more difficult [7]



Figure 1.1: (right) Different Digits Writing (left) Different Letters Writing

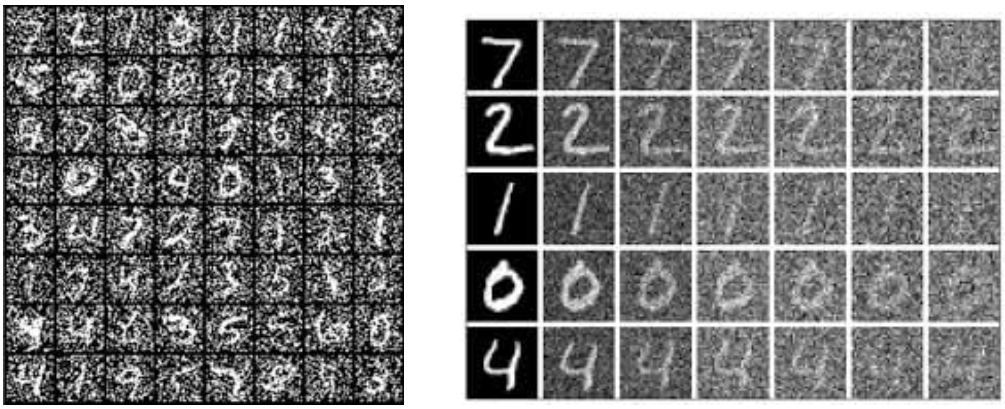


Figure 1.2: Some distorted and noisy handwriting from MNIST dataset

Similar-looking characters

Certain digits and letters have similar shapes, increasing the risk of misclassification. Examples include 3, 8, 4 or 0 and O. While us humans rely on context to distinguish them, machines must learn subtle visual differences, which can be difficult under noisy or distorted conditions.[8]

1.2.2 Real-world Applications of Handwritten Recognition

Banking sector

Banks rely on handwritten recognition systems to read amounts, account numbers, and other information on checks and forms. This helps speed up financial transactions and minimizes human errors during data entry.[9]

Postal systems

In postal systems, handwritten recognition is used to identify zip codes and addresses written on envelopes and packages. By automatically reading these digits, postal services can sort mail more quickly and reduce delivery mistakes.[4, 9]

Document digitization and administrative automation

Many institutions, such as hospitals, universities, libraries, and government offices, deal with handwritten forms and records. Recognition systems allow these documents to be converted into searchable digital data, making storage and organization more efficient.

Education

For example in grading exam sheets or processing student registration forms that contain handwritten entries.

Traffic systems

Extract numbers from vehicle license plates for monitoring and enforcement purposes.[9]

1.3 Digital Image Fundamentals and Preprocessing

1.3.1 Digital Image Representation

Digital images are grids of pixels, where each pixel holds numerical values representing color or intensity, forming the foundation for processing tasks.[10].

A handwritten digit image turns into a grid of numbers called a matrix, where each spot matches a pixel's brightness.[11]

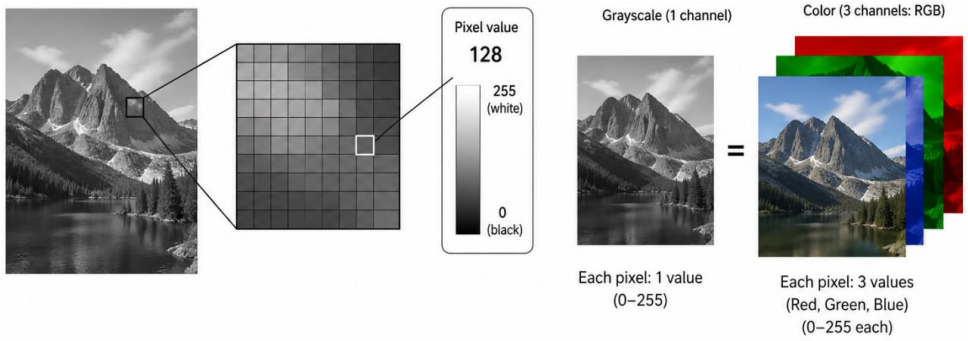


Figure 1.3: Structure of Pixels in Digital Image(image generated with IA)

Pixels and Resolution

Pixels, or picture elements, are the smallest units in a digital image, arranged in a 2D grid defined by width (x-axis) and height (y-axis). Resolution measures the total pixels, like 1920x1080, determining image sharpness; higher values capture finer details but increase file size.[12] [13].

Color Representation

Grayscale images use one value per pixel for brightness (0 black to 255 white in 8-bit). Color images employ RGB model with three channels—red, green, blue—each 0-255, enabling 16.7 million colors per pixel in 24-bit true color.[13].

1.3.2 Basic Preprocessing Techniques

Preprocessing boosts model accuracy by minimizing noise, inconsistencies, and outliers that mislead learning, as studies show up to 50 gains in clustering tasks.[14]

It accelerates convergence in training, like faster gradient descent from normalized data, and curbs overfitting for better generalization to unseen inputs.[15]

Uniform data also cuts computational costs and enhances interpretability by focusing algorithms on relevant patterns.[14]

Normalization

Normalization scales pixel values to a uniform range, like 0-1 via min-max scaling or z-score standardization, ensuring features contribute equally without dominance by larger values.[16]

Resizing

Adjusts images to consistent dimensions, such as 224x224 pixels, using methods like bilinear interpolation to maintain aspect ratios while enabling batch processing in neural networks.[17]

Noise reduction

This removes artifacts from sensors or environments via filters like Gaussian blurring or median filtering, preserving key features for clearer model inputs.[17]

Data augmentation

Data augmentation artificially expands datasets through rotation (e.g., 0-30 degrees) and scaling (e.g., 0.8-1.2x), introducing variations to mimic real-world diversity without collecting new data.[18]

1.4 From Machine Learning to Deep Learning

1.4.1 Classical Machine Learning Approach

Machine Learning (ML) is a computational approach that automatically builds models from training data to capture complex relationships between input and output variables, enabling prediction on unseen data.[19]

Supervised vs Unsupervised learning

Supervised and unsupervised are two basic methods in ML, differing mainly in how they use data for training models.[20]

Supervised Learning is a type of ML where a model is trained using labelled data, meaning each input comes with the correct output. The model learns the relationship between inputs and outputs so that it can classify new data or predict future outcomes.[20]

On the other side, unsupervised learning works with unlabeled data, identifying hidden patterns, clusters, or structures without guidance. Common uses include customer segmentation or anomaly detection.[20]

Classification vs Regression

Classification and regression are fundamental tasks in supervised machine learning, differing mainly in the type of output they predict. Classification handles discrete categories, while regression predicts continuous values.[21]

Feature-Based Learning

Traditional ML sometimes needs features because its algorithms are not designed to "see" raw data directly; they work on structured numeric inputs, so humans must decide what information to extract and how to represent it.[22]

Traditional models like linear regression (LR), Support Vector Machines (SVMs), k-Nearest Neighbor (k-NN), or Random Forests (RF) cannot interpret raw pixels, text, or sensor signals as-is; they expect a fixed-length vector of meaningful numbers (features).[22]

Raw data, such as all pixel values of an image, can make the model struggle to find useful patterns, or overfit or become too slow and unstable .[22] In fact, for an image, "raw pixels" are a huge list of color values (e.g., red, green, blue at each position). [4]

Consequently, a simple traditional classifier cannot easily tell which pixel combinations correspond to "car," "face," or "text," so we compress or transform that raw pixel grid into a smaller, more meaningful feature representation (such as color averages, edges, or shapes). [22]

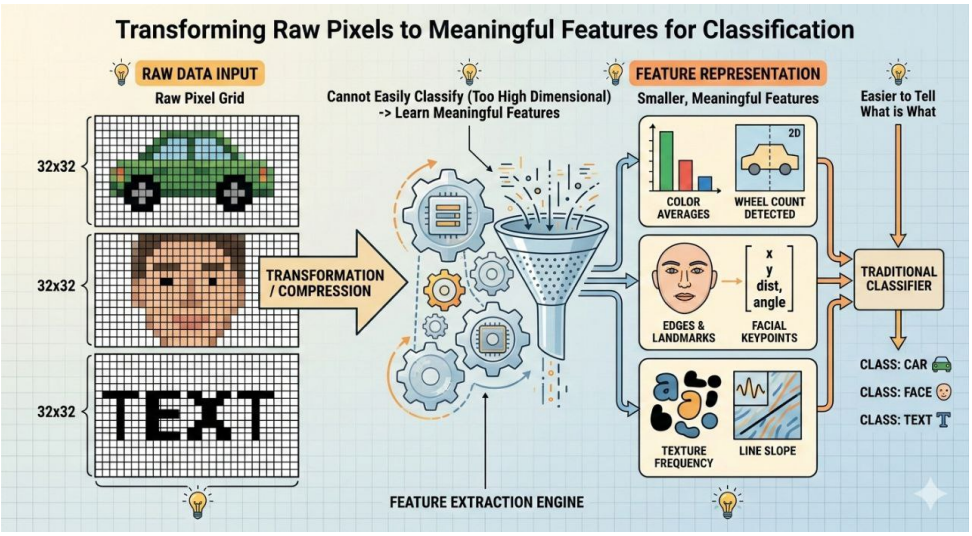


Figure 1.4: Transforming Raw Pixels to Meaningful Features for Classification (image generated with AI)

1.4.2 Artificial Neural Networks (ANN)

ANNs are models inspired by how the brain works. They are made of simple units called neurons organized in layers (input, hidden, and output). Each neuron receives data, applies weights and an activation function, then passes the result to the next layer.

These networks are used to learn from data and solve problems like classification and prediction. During training, they adjust their weights using backpropagation to improve their results.

Neuron

An artificial neuron is the basic unit of an ANN. It receives several input values, where each input is associated with a weight. These inputs are combined together with a bias term, and the result is passed through an activation function to produce an output.

This process is inspired by biological neurons, where information is transmitted through weighted connections, allowing the network to learn and recognize complex patterns from data.[23]

Mathematically, the output of a neuron can be expressed as:

$$y = f(\sum_{i=1}^{n} w_i x_i + b) \tag{1.1}$$

where xi represent the inputs, wi the corresponding weights, b the bias,and f the activation function.[3]

Activation function

An activation function in an artificial neural network is a mathematical function applied to the sum of weighted inputs in a neuron. It introduces non-linearity, which allows the network to learn and represent complex patterns in the data. [24]

Without this non-linearity, even networks with many layers would behave like simple linear models and could not capture complicated relationships.[25]

Some common activation functions are:

- Sigmoid, which maps inputs to a range between 0 and 1.
- ReLU (Rectified Linear Unit), defined as $f(x) = \max(0, x)$, which helps make the network more efficient and speeds up training.[26]

Layers

Layers in an ANN are groups of neurons arranged one after another. Each layer takes input from the previous layer, applies weights, biases, and an activation function, and then sends the result to the next layer.[27]

This setup allows the network to learn step by step. The input layer handles the raw data, hidden layers process it into more abstract features, and the output layer gives the final prediction.[26] When a network has several hidden layers, it is called "deep," which helps it learn complex patterns by processing data through multiple stages.[26]

1.4.3 Convolutional Neural Networks (CNN)

CNN is a type of neural network made for images. It is a Deep learning (DL) technique. This later is a form of ML that enables computers to learn from experience and understand the world through a hierarchy of concepts, where each concept is defined in relation to simpler ones, resulting in a deep graph with multiple layers.

This approach allows systems to automatically build complex representations directly from raw data without relying on hand-crafted features. As a result, it is particularly effective in tasks such as image recognition, where features are progressively abstracted from simple edges to contours, and ultimately to complete objects.[28]

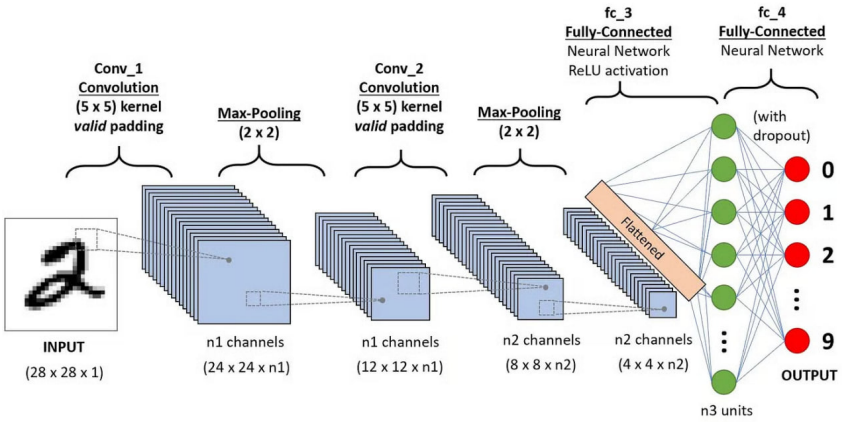


Figure 1.5: Architecture of a CNN model applied to digit recognition[29]

CNNs are efficient because they share parameters and focus on local areas, learning features from simple edges to full objects. They were first used in LeNet-5 (1998) and are now common in computer vision tasks like object detection.

CNN models consist of the following components:

Convolution

Convolution slides a small window, called a kernel, over the input image to produce a feature map. Conceptually, it multiplies the kernel values with overlapping image patches and sums them, highlighting local patterns like edges or textures in the output map.[30]

Filters detect patterns

Filters, or kernels, are learned during training to detect specific features; early layers identify basic patterns like lines or corners, while deeper layers combine them into complex shapes like eyes or wheels. Each filter scans the entire image independently, creating multiple feature maps that stack to form richer representations.[31]

Pooling

Pooling reduces feature map dimensions by summarizing regions, typically via max pooling (taking the maximum value in a window) or average pooling. This downsamples data, cuts computation, adds translation invariance (small shifts don't change output much), and helps prevent overfitting.[32]

1.5 Reference Architectures

1.5.1 LeNet-5

LeNet-5 is one of the earliest CNNs designed for handwritten digit recognition. It is a pioneering model in the field of Deep Learning and is particularly known for its application on the MNIST dataset.

The architecture consists of several layers, including convolutional layers for feature extraction, pooling layers for dimensionality reduction, and fully connected layers for

classification. More specifically, LeNet-5 is composed of seven layers (excluding the input), where convolutional layers use small kernels (5×5) to detect local patterns, followed by average pooling layers that reduce the spatial dimensions. The network typically uses activation functions such as tanh to introduce non-linearity. The final fully connected layers perform the classification task by mapping the extracted features to output classes.

1.5.2 Moderne Architectures

In addition to LeNet-5, more advanced convolutional neural network architectures have been developed for digit recognition and image classification tasks, such as deeper CNN models that improve performance through increased depth and the use of modern activation functions like ReLU. These architectures differ from LeNet-5 in terms of complexity, number of layers, and optimization techniques.

Figure 1.5 illustrates the overall architecture of one of the proposed CNN models. While table bellow represents a summary of existing architectures applied for image classification:

Architecture 1	**Architecture 2**	**Architecture 3**	**Architecture 4**	**Architecture 5**
Only one input layer and two fully connected layers	2 convolutional layers (2×2 filters) + 2 FC layers	3 convolutional layers (2×2 filters) + 2 FC layers	4 convolutional layers (2×2 filters) + 2 FC layers	4 convolutional layers (3×3 filters) + 2 FC layers
INPUT: 28×28×1 FC: 10 output classes FC: 128 hidden neurons	INPUT: 28×28×1 CONV2D: 2×2, 64 filters POOL: 2×2 DROPOUT: 0.25 CONV2D: 2×2, 64 filters DROPOUT: 0.25 FC: 64 neurons DROPOUT: 0.25	INPUT: 28×28×1 CONV2D: 2×2, 64 filters POOL: 2×2 DROPOUT: 0.25 CONV2D: 2×2, 64 filters POOL: 2×2 DROPOUT: 0.25 CONV2D: 2×2, 64 filters DROPOUT: 0.25 FC: 64 neurons DROPOUT: 0.25	INPUT: 28×28×1 CONV2D: 2×2, 64 filters POOL: 2×2 DROPOUT: 0.25 CONV2D: 2×2, 64 filters POOL: 2×2 DROPOUT: 0.25 CONV2D: 2×2, 64 filters POOL: 2×2 DROPOUT: 0.25 FC: 64 neurons DROPOUT: 0.25	INPUT: 28×28×1 CONV2D: 3×3, 32 filters CONV2D: 3×3, 32 filters POOL: 2×2 DROPOUT: 0.25 CONV2D: 3×3, 64 filters CONV2D: 3×3, 64 filters POOL: 2×2 DROPOUT: 0.25 FC: 512 neurons DROPOUT: 0.5

Table 1.1: Different CNN Architectures for Image Classification[33]

1.6 Training Methodology and Evaluation Metrics

1.6.1 Data splitting

Training Set

In ML, the **training set** is the portion of the data used to actually fit or "teach" the model its parameters (weights, splits, clusters, etc.). [34]

Validation Set

The **validation set** is a subset of the data used to evaluate and tune a model during training, without touching the final test set. [35]

It is used to compare different models or hyperparameter choices (learning rate, number of layers, regularization strength, etc.) and to detect overfitting on the training data [36].

Test Set

The **test set** is a reserved subset of data that is used only to evaluate the final performance of a trained model. [37]. The model parameters and hyperparameters are never adjusted based on the test set to avoid overfitting the evaluation protocol itself [38].

1.6.2 Performance Metrics

Accuracy

Accuracy is a basic evaluation metric that measures the proportion of correct predictions out of all predictions made by a classifier on a given data set. [39] Higher accuracy indicates that the model matches the true labels more often. [40].

Confusion Matrix

A confusion matrix shows how a classification model's predictions compare to the true labels (TP, TN, FP, FN). [41]. It allows computing metrics like accuracy, precision, recall, and F1-score. [36]

Precision / Recall

Precision and recall help understand model errors; precision is a fraction of predicted positives that are correct, and recall is a fraction of actual positives that are detected.

1.6.3 Overfitting

Overfitting occurs when a model learns the training data too well, including noise, and performs poorly on new data. [41] To reduce it, we usually apply Dropout that randomly turns off neurons during training [42], or apply Data Augmentation that artificially increases training data diversity [43].

1.7 State of the art

Early approaches relied on classical machine learning algorithms such as Logistic Regression (LR), K-Nearest Neighbors (KNN), Support Vector Machines (SVM), and Random Forests (RFC). Although these methods produced acceptable results, their performance remained limited compared to modern deep learning techniques. [44]

As shown in table bellow, DL models significantly outperform classical machine learning approaches on MNIST.

Today, DL models, especially CNN, dominate handwritten digit recognition research. CNNs automatically extract image features and achieve very high accuracy on benchmark datasets like MNIST. Researchers have also proposed advanced architectures such as ResNet, DenseNet, CapsNet, and hybrid CNN-ViT models to further improve performance. [44, 45]

Hybrid architectures such as CNN-ViT currently achieve the best reported results, reaching approximately 99.97% accuracy. [44][45]

Despite these excellent results, MNIST is now considered a saturated dataset because many models already achieve very high accuracy. Therefore, recent research focuses more on improving robustness, efficiency, and generalization using more difficult datasets such as EMNIST.

Paper	**Model**	**Accuracy**	**Key Strength**	**Limitation**
[44]	LR, MLP, CNN, ViT, CNN-ViT	99.97%	Hybrid models achieved the best performance	EMNIST remains more challenging
[46]	CNN, ResNet, DenseNet, CapsNet	99.75%	CapsNet preserved spatial relationships effectively	Marginal improvement on MNIST
[45]	SVM, KNN, RFC, CNN	98.70%	Clear comparison between classical ML and deep learning	CNN performance below recent SOTA
[47]	Feedforward neural networks	98.67%	Hyperpa-rameter tuning improved shallow networks	Still weaker than CNN-based approaches

Table 1.2: Comparison of Handwritten Digit Recognition Approaches using MNIST dataset

1.8 Conclusion

In this chapter, we have presented a theoretical overview of hand digit recognition systems. In summary, DL, specifically CNN architectures like LeNet-5, offers superior robustness in handling handwriting variability compared to traditional methods. The following chapter will focus on the design and implementation of our CNN model. Furthermore, a comparative analysis will be conducted with traditional ML approaches.

Chapter 2

Design and Implementation of the Prediction System

2.1 Introduction

This chapter presents the complete design and implementation of our handwritten digit recognition system based on the MNIST dataset. The project explores and compares multiple ML and DL approaches, including CNN, SVM, Random Forests, and k-NN, in order to identify the most accurate and efficient model for digit classification.

2.2 System Architecture

2.2.1 Global Architecture of the Prediction Pipeline

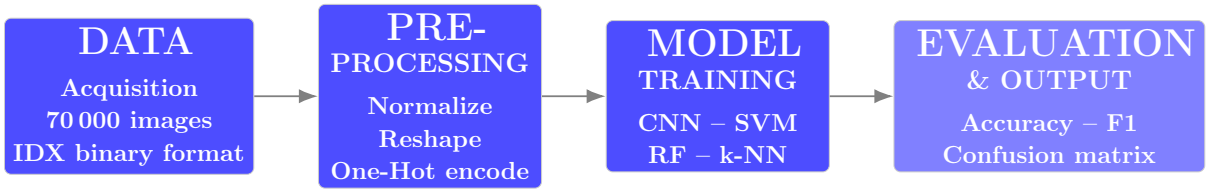


Figure 2.1: Global Architecture of the Prediction Pipeline

2.2.2 Data Flow

The data flows through the pipeline in the following order:

- **Input**: raw MNIST binary files (IDX format) containing 70,000 grayscale digit images.
- **Preprocessing**: normalization, reshaping for CNN compatibility, and one-hot encoding of labels.
- **Model**: parallel training of CNN, SVM, Random Forest, and k-NN classifiers on the processed data.

- **Output**: predicted digit class with associated accuracy metrics and visual evaluation reports.

2.2.3 Tools and Libraries Used

The following tools and libraries form the technical foundation of this project:

- Python 3.x — primary programming language
- TensorFlow / Keras — deep learning framework for building and training the CNN
- Scikit-learn — classical machine learning models (SVM, Random Forest, k-NN) and evaluation metrics
- NumPy — numerical array operations and data manipulation
- Matplotlib / Seaborn — data visualization and result plotting
- Struct / OS — low-level reading of MNIST binary IDX files
- Pandas — structured reporting of classification metrics

2.3 Data Preparation

2.3.1 Dataset Description

We utilized the MNIST (Modified National Institute of Standards and Technology) dataset, which serves as a standard benchmark in image classification research. In this project, we work with a total of 70,000 grayscale images of handwritten digits partitioned and configured as follows:

- Training set: 60,000 images with corresponding labels
- Test set: 10,000 images with corresponding labels
- Image resolution: 28×28 pixels per image
- Number of classes: 10 (digits 0 through 9)
- Pixel values: integers in the range $[0, 255]$ where 0 is black and 255 is white

2.3.2 Data Preprocessing:

Normalization and Scaling Pixel values are divided by 255.0 to rescale them from the integer range $[0, 255]$ to the float range $[0.0, 1.0]$. This normalization ensures numerical stability during gradient descent and improves convergence speed.

Listing 2.1: Normalization

```
x_train = train_images / 255.0
x_test = test_images / 255.0
```

Reshaping for CNN Input: CNN layers in Keras expect 4-dimensional tensors with shape (batch, height, width, channels). Since MNIST images are grayscale (single channel), each image is reshaped from $(28, 28)$ to $(28, 28, 1)$:

Listing 2.2: Reshaping for CNN

```
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)
```

Label Encoding (One-Hot): Integer class labels (0–9) are converted to one-hot encoded vectors of length 10 using Keras's `to_categorical()` function. This format is required by the categorical cross-entropy loss function used during CNN training. For classical models (SVM, Random Forest, k-NN), the original integer labels are used directly.

Listing 2.3: One-hot encoding

```
y_train = to_categorical(train_labels, num_classes=10)
```

2.3.3 Train/Test Split Strategy

We used the standard MNIST dataset split, which consists of 60,000 training examples and 10,000 test examples, to ensure all models were evaluated on identical data.

When training the CNN, We allocated 10% of the training data (6,000 samples) as a validation set to monitor the model's loss and accuracy during training. For the classical classifiers (SVM, Random Forest, and k-NN), We trained them directly on the full training set and reserved the 10,000 test samples strictly for the final evaluation.

2.4 Model Design

2.4.1 Convolutional Neural Network (CNN)

The architecture is directly inspired by Architecture 5 from Table 1.1 :

- `Conv2D(32, 3×3, ReLU)` — first feature extraction layer, detecting low-level patterns such as edges and curves
- `MaxPooling2D(2×2)` — reduces spatial dimensions by half, retaining dominant features
- `Conv2D(64, 3×3, ReLU)` — extracts higher-level features combining primitive patterns
- `MaxPooling2D(2×2)` — further spatial downsampling
- `Conv2D(64, 3×3, ReLU)` — deep feature extraction
- `Flatten()` — converts 2D feature maps into a 1D vector
- `Dense(128, ReLU)` — fully connected layer for pattern learning in feature space

- `Dropout(0.5)` — randomly disables 50% of neurons during training to prevent overfitting
- `Dense(10, Softmax)` — output layer producing class probability distribution over 10 digit classes

2.4.2 Support Vector Machine (SVM)

The SVM model is applied to the flattened pixel vectors ($28 \times 28 = 784$). The model is configured with the following hyperparameters:

Table 2.1: Hyperparameter and Configuration Settings for the SVM Model

Parameter	**Value**
Model Type	Support Vector Classifier (SVC)
Input Dimensions	784 (28×28)
Kernel Type	RBF (Radial Basis Function)
Regularization (C)	10
Kernel Coefficient (γ)	'scale'
Target Label Format	Integer (0–9)
Training Dataset Size	60,000 samples

2.4.3 Random Forest

We implemented a Random Forest classifier using 100 decision trees (`n_estimators=100`). We configured the model to train in parallel across all available CPU cores (`n_jobs=-1`) and set a fixed random seed (`random_state=42`) to ensure reproducibility.

Because Random Forest is naturally robust to overfitting and handles raw pixel intensities well, we were able to train the model directly on the flattened bootstrap samples without applying any feature scaling.

2.4.4 k-Nearest Neighbors (k-NN)

The k-NN classifier stores the entire training dataset and classifies new examples by finding the $k = 3$ closest training samples in Euclidean distance. While conceptually simple, k-NN requires no explicit training phase; however, prediction time is computationally expensive for large test sets as it involves computing distances to all 60,000 training examples.

2.5 Model Training

2.5.1 CNN Training Process

The CNN is trained using the `model.fit()` method with the following configuration:

- **Epochs**: 20 — the model passes over the full training set 20 times
- **Batch size**: 32 — weights are updated after every 32 samples

- **Validation split**: 10% — 6,000 samples are held out for validation monitoring
- **Shuffle**: True — the training data is reshuffled at the beginning of each epoch

Training and validation accuracy/loss curves are recorded in the history object and plotted after training to diagnose underfitting or overfitting.

2.5.2 Classical Model Training

The SVM, Random Forest, and k-NN models use scikit-learn's standard `fit()` interface on the flattened $(N, 784)$ feature matrices. No explicit epoch loop is required; training runs as a single optimization procedure. Hyperparameters were selected based on established best practices for the MNIST task:

- **SVM**: RBF kernel, $C = 10$, `gamma='scale'` — balances margin width and classification accuracy
- **Random Forest**: 100 trees, parallel execution — provides strong accuracy with manageable training time
- **k-NN**: $k = 3$ neighbors — small k captures local patterns without excessive noise sensitivity

2.6 Comparative analysis

We evaluated our four models on the same 10,000-sample test set. The results obtained from our experiments are summarized in Table 2.2 below.

Table 2.2: Comparative performance of the four classifiers on the MNIST test set (10,000 samples)

Model	**Test Accuracy**	**Training Time**	**Key Characteristic**
CNN	99.2%	∼8 min (CPU)	Hierarchical spatial feature learning
SVM (RBF)	98.1%	∼12 min (CPU)	Strong on flattened vectors; slow on large data
Random Forest	97.0%	∼2 min (parallel)	Robust ensemble; no feature scaling needed
k-NN (k=3)	96.8%	Instant (lazy)	No training; high prediction latency

The CNN achieves the highest test accuracy of 99.2% (see confusion matrix bellow) by learning hierarchical spatial features directly from pixel data. It is the most powerful model but requires GPU resources and longer training time for larger experiments. The SVM with RBF kernel reaches 98.1% accuracy and trains without requiring specialized hardware, though it scales poorly to very large datasets. Random Forest delivers 97.0% accuracy with fast training thanks to parallelism, but cannot match the CNN on fine-grained visual tasks. k-NN achieves 96.8% accuracy with no explicit training phase, but suffers from high prediction latency on large test sets.

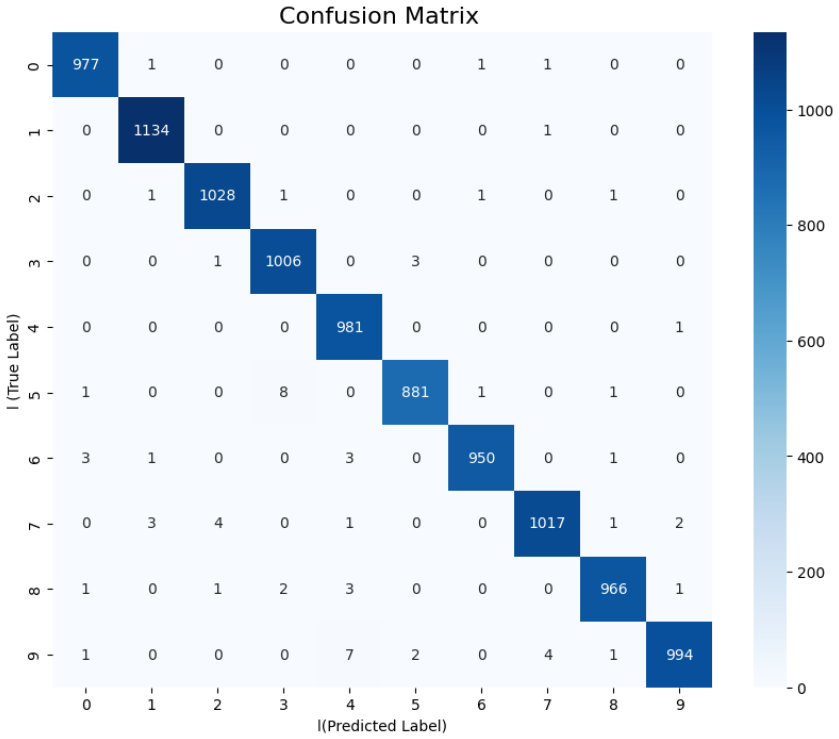


Figure 2.2: CNN Confusion Matrix

2.7 Conclusion

This chapter detailed the complete design and implementation of our digit recognition system. The MNIST dataset was prepared through normalization, reshaping, and label encoding before being fed into four classifiers: CNN, SVM, Random Forest, and k-NN. CNN was selected as the final production model due to its superior classification accuracy (99.2%), its ability to generalize to unseen handwriting styles, and its direct compatibility with image input format. Next chapter will be dedicated to present our web application and interface.

Chapter 3

Web Application and User Interface

3.1 Introduction

In this project, we developed NeuralDigit Pro, a web-based application designed to deploy and demonstrate our CNN trained on the MNIST dataset. While standard evaluations rely purely on static offline metrics, we built this platform to test our model's performance and robustness in a real-world, interactive environment.

3.2 Objectives and Application Role

Our primary objective was to make the trained model accessible to both technical and non-technical users by abstracting the underlying code into an intuitive user interface.

We implemented two distinct input modes: a real-time drawing canvas utilizing touch or mouse input, and a standard image upload pipeline. To ensure the application serves as a comprehensive tool rather than just a simple black-box predictor, we designed the interface to expose the model's inner workings by visualizing the complete preprocessing pipeline, generating Grad-CAM activation maps, and flagging low-confidence predictions.

To make the model's decision-making process transparent for evaluation, we structured the interface to expose the recognition pipeline at multiple stages:

- **Preprocessing Pipeline:** We display the four core transformations (original, resized, binarized, and normalized) to show exactly how raw user input is formatted for the network.
- **Prediction Metrics:** We output the top-three predictions ranked by probability alongside a comprehensive 10-class histogram.
- **Explainability Features:** We integrate a Grad-CAM heatmap overlay to highlight the specific pixel regions that heavily influenced the network's classification.
- **Administrative Analytics:** We built a background dashboard to track real-world usage statistics and log common misclassification pairs for further model refinement.

3.3 System Overview

3.3.1 General Architecture

We implemented a classic client-server architecture for the application, decoupling it into two distinct layers that communicate via a RESTful HTTP API. This architectural split ensures modularity, allowing us to update or modify the frontend graphical interface independently of the backend inference engine.

Component	**Role**
Frontend (HTML/CSS/JS)	User interface: drawing canvas, image upload, display of results and visualizations
Flask Backend (Python)	HTTP server exposing REST endpoints, driving the preprocessing pipeline and model inference
Keras Model (.h5)	CNN network loaded into memory at server startup, called upon for each prediction request
SQLite Database	Persistent logging of every prediction for subsequent analysis and the admin dashboard

Table 3.1: Core components of the NeuralDigit Pro system

In summary, the client (browser) sends user input to the Flask server via HTTP requests. The server preprocesses the image, executes inference using the pre-loaded Keras model, logs the result in SQLite, and returns the prediction with associated visualizations to the client.

3.3.2 Communication Between Components

The frontend and backend communicate exclusively via HTTP/REST requests. The backend, built with Flask, exposes the following endpoints:

Endpoint	**Method**	**Description**
`/predict`	POST	Canvas drawing (base64 JSON)
`/predict_upload`	POST	Image upload (multipart) — supports single or segment mode
`/api/admin/stats`	GET	Usage statistics (per digit, confusion, time-series)
`/api/admin/clear`	POST	Clears prediction history from the database

Table 3.2: API endpoints

For canvas drawings, the image is base64-encoded and sent as JSON. For uploads, the client can request segmentation mode, where the server performs contour detection to extract individual digits. Each digit is processed independently, and the response includes bounding boxes and an annotated overview image. The uncertainty indicator triggers when the top confidence falls below 0.60. This stateless design ensures request independence and simplifies future distributed deployment.

3.4 Backend Development

3.4.1 Flask

We developed the backend using Flask, a lightweight Python micro-framework, which we selected for its minimal memory footprint and its native compatibility with the Python scientific ecosystem (including NumPy, TensorFlow, OpenCV, and Pillow). This allowed us to efficiently bridge our deep learning model with a production API without the unnecessary overhead of larger frameworks like Django.

In our initial configuration, we restricted user image uploads to a maximum size of 16 MB to ensure stability. We then organized the API routes into two main families: user endpoints for classification (`/predict` and `/predict_upload`) and administrative endpoints for monitoring system performance (`/admin`, `/api/admin/stats`, and `/api/admin/clear`).

3.4.2 Model Integration

Loading the Trained Model. The trained model is loaded at server startup via `tf.keras.models.load_model()` from the file `mnist_model_final.h5`. The TensorFlow import is wrapped in a `try/except` block that activates a fallback mode (mock predictions) if TensorFlow is not installed or the model file is absent. This graceful degradation mechanism ensures the interface remains functional in all circumstances.

Prediction Pipeline. The central function `run_prediction()` orchestrates the complete flow:

1. Preprocessing of the PIL image received as input
2. Model call to obtain the softmax probability vector over 10 classes
3. Extraction of the top three predictions (Top-3) with their indices and scores
4. Grad-CAM heatmap computation for the top predicted class
5. Persistence of the result in the SQLite database
6. Return of the enriched result to the frontend

3.4.3 REST API Design

All responses are in JSON. The nominal response body of `/predict` contains:

Field	Description
`top3`	List of three objects {digit, confidence} ranked by decreasing probability
`is_uncertain`	Boolean true if top-1 confidence is below the 60% threshold
`all_probs`	Vector of 10 softmax probabilities for all digit classes (0–9)
`pipeline`	Dictionary of four base64 images: original, resized, binary, normalized
`heatmap`	Base64 image of the Grad-CAM activation map

Table 3.3: Response fields of the `/predict` endpoint

Example JSON response from the `/predict` endpoint:

Listing 3.1: Example JSON response

```
{
    "top3": [
        {"digit": 7, "confidence": 0.9542},
        {"digit": 1, "confidence": 0.0231},
        {"digit": 9, "confidence": 0.0087}
    ],
    "is_uncertain": false,
    "all_probs": [0.0001, 0.0231, 0.0002, 0.0003, 0.0001, 0.
       0002, 0.0004, 0.9542, 0.0005, 0.0129],
    "pipeline": {
        "original": "...",
        "resized": "...",
        "binary": "...",
        "normalized": "..."
    },
    "heatmap": "..."
}
```

3.5 Frontend Development

3.5.1 Technologies Used

The interface is built with HTML5, CSS3, and vanilla JavaScript. No external JavaScript framework is required, which reduces dependencies and simplifies deployment.

The main browser APIs used are:

- **Canvas API**: freehand drawing, mouse and touch coordinate capture, PNG export as base64
- **Fetch API**: asynchronous HTTP requests to the Flask backend
- **File API**: reading and previewing image files uploaded by the user
- **Drag and Drop API**: image drop via drag-and-drop onto the upload zone

3.5.2 Interface Design

The interface adopts a dark, cyberpunk-inspired aesthetic with a violet accent (`#7c6af7`), animated particle network, and custom cursor. It is structured as a single-page application with three scrollable sections: Hero, Demo, and Architecture.

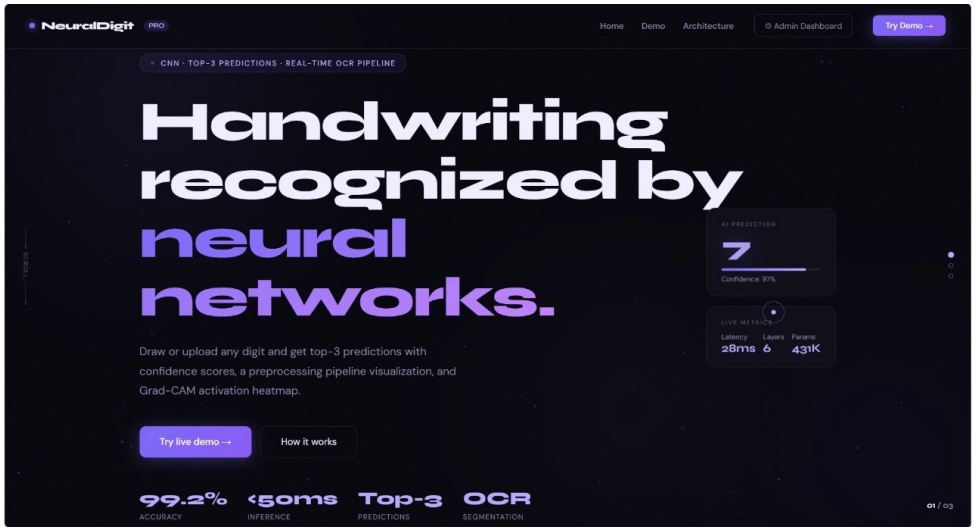


Figure 3.1: Home Page of the Handwriting Recognition Application

Drawing Canvas. An HTML5 canvas (370×220 pixels) captures freehand drawing. White strokes on a black background simulate MNIST style. Pointer events handle mouse and touch uniformly. The image is exported as base64 PNG via `canvas.toDataURL()` and sent to the `/predict` endpoint.

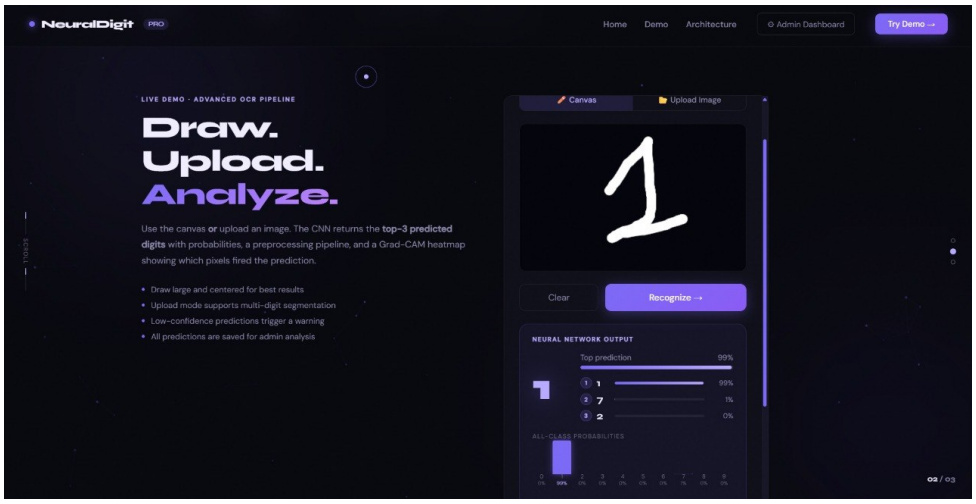


Figure 3.2: Drawing Canvas Interface

Image Upload. The upload zone supports drag-and-drop and file selection. A toggle switch activates multi-digit segmentation mode, which sends the image with `mode='segment'`. The backend returns an annotated image with bounding boxes and per-digit predictions.

Result Display. The results panel displays:

- Main prediction: large digit with animated confidence bar
- Top-3: ranked digits with horizontal probability bars
- All-probabilities histogram: vertical bars for digits 0–9
- Preprocessing pipeline: four thumbnails (original, resized, binarized, normalized)
- Grad-CAM heatmap: color-coded activation map
- Uncertainty banner: yellow alert when confidence is below 60%

3.5.3 User Experience

The interface adopts a dark design with a purple accent (`#7c6af7`), custom animated cursor, particle network, and smooth CSS transitions. Navigation between three sections (Hero, Demo, Architecture) supports mouse wheel, touch swipe, keyboard arrows, and navigation dots. A unique `session_id`, generated client-side, tracks user interactions without requiring authentication.

3.6 Image Processing in the Application

3.6.1 Preprocessing Pipeline

The `preprocess_image()` function implements a four-stage pipeline that transforms any input image into a tensor compatible with the MNIST training format (28×28 pixels, single channel, values in $[0, 1]$). Each stage produces a base64 image returned to the frontend for visualization.

Stage	Operation	Output format	Rationale
1	Grayscale conversion	28×28 grayscale	MNIST is monochromatic
2	Lanczos resizing	28×28 pixels	Match model input size; reduce aliasing
3	Otsu binarization	Binary (0 or 255)	Robust to lighting variations
4	Normalization	Float $[0.0, 1.0]$	Consistency with training data

Table 3.4: Four-stage preprocessing pipeline

Stage 1 — Grayscale Conversion. Pillow's `Image.convert('L')` method converts the color image to grayscale. This operation is essential because the MNIST dataset is monochromatic: working with three RGB channels would introduce unnecessary bias and triple the input dimensionality.

Listing 3.2: Grayscale conversion

```
original_gray = pil_img.convert('L')
```

Stage 2 — Resizing to 28×28. The image is resized to 28×28 pixels via `Image.resize((28, 28), Image.LANCZOS)`. The Lanczos resampling algorithm is preferred over simple bilinear interpolation because it reduces aliasing artifacts and better preserves the fine contours of digits, especially during large reductions.

Listing 3.3: Resizing with Lanczos

```
resized = original_gray.resize((28, 28), Image.LANCZOS)
```

Stage 3 — Otsu Binarization. Otsu's method automatically determines an optimal threshold to separate the digit foreground from the background. The algorithm maximizes the inter-class variance between the two pixel classes (foreground and background), making it robust to lighting variations and drawing pressure. It is implemented manually in pure Python to avoid dependencies on OpenCV for this step.

Otsu's Formula. The algorithm finds the threshold t^* that maximizes the inter-class variance:

$$\sigma_B^2(t) = w_0(t) \cdot w_1(t) \cdot [\mu_0(t) - \mu_1(t)]^2$$

where $w_0(t)$ and $w_1(t)$ are the proportions of pixels below and above threshold t, and $\mu_0(t)$, $\mu_1(t)$ are their respective mean intensities. A pixel is classified as foreground (digit) if its value $> t^*$, and as background otherwise.

Stage 4 — Normalization to [0, 1]. The final stage scales pixel values from the integer range $\{0, 255\}$ to the floating-point interval $[0.0, 1.0]$ by dividing by 255.0. This normalization ensures numerical stability during inference and is mandatory for consistency with training-time preprocessing.

Listing 3.4: Normalization

```
normalized_arr = binary_arr / 255.0
model_input = normalized_arr.reshape(1, 28, 28, 1)  # (batch,
    height, width, channels)
```

3.6.2 Consistency with Training Data

Consistency between training-time and inference-time preprocessing is critical for optimal model performance. Every image submitted by the user follows exactly the same pipeline as the original MNIST images: the final tensor of shape $(1, 28, 28, 1)$ is structurally identical to a training batch of size 1. Any divergence between training and inference distributions (dataset shift) would be a direct source of performance degradation.

Common pitfalls that were explicitly avoided:

- Incorrect normalization: using $[0, 255]$ instead of $[0, 1]$ would cause gradient saturation at inference
- Different resampling: using nearest-neighbour instead of Lanczos would introduce jagged artifacts

- Missing binarization: keeping grayscale values would mismatch the binary training distribution
- Inverted colours: MNIST digits are white on black; the pipeline enforces this convention

3.7 Integration and Testing

3.7.1 Frontend-Backend Connection

Communication uses the JavaScript Fetch API. Canvas drawings are exported via `canvas.toDataURL('image/png')` and sent as JSON. Uploads use FormData for binary files. A state management mechanism prevents double submissions: the button is disabled and shows 'Analyzing...' until the response is received.

3.7.2 Functional Testing

Prediction Accuracy. Tests on manually drawn digits confirm consistency with MNIST evaluation metrics. Ambiguous shapes (1/7, 4/9, 3/8) correctly trigger the uncertainty banner when confidence falls below 60%.

Error Handling. The backend validates all inputs:

- Invalid/corrupted image: HTTP 400 with explicit message
- Missing file: HTTP 400 'No image file provided'
- Empty canvas: frontend validation prevents submission before the request is sent
- Model unavailable: graceful fallback to mock predictions — the server does not crash
- No digits detected in segmentation mode: HTTP 400 'No digits found'
- File too large: automatic rejection above the 16 MB Flask limit

Performance. End-to-end latency is typically 50–300 ms per request, broken down as follows: canvas export < 5 ms, network round-trip 10–20 ms, preprocessing 15–25 ms, CNN inference 20–35 ms, database write < 10 ms. This provides near-real-time feedback for the user.

3.7.3 Performance Considerations

Multi-digit segmentation chains inferences linearly with the number of detected digits but remains interactive for fewer than ten characters. Grad-CAM generation adds approximately 15–20 ms per request. For production workloads requiring high concurrency, the Flask development server should be replaced with Gunicorn.

3.8 Deployment

3.8.1 Local Deployment

Local startup requires three commands after installing dependencies:

Listing 3.5: Local deployment commands

```
# Copy the trained model into the project directory
cp mnist_model_final.h5 neuraldigit_pro/

# Install dependencies
pip install flask tensorflow numpy pillow opencv-python

# Start the server
python app.py

# Open http://localhost:5000
```

System requirements:

- Python 3.8+ with a minimum of 2 GB RAM
- Dependencies: Flask, TensorFlow, NumPy, Pillow, OpenCV
- Optional: NVIDIA GPU for faster inference

For production, the Flask development server should be replaced with Gunicorn (a production-grade WSGI server), Nginx should be configured as a reverse proxy, and containerization with Docker is strongly recommended. SQLite should also be migrated to PostgreSQL for concurrent workloads.

3.8.2 Limitations and Constraints

- **Concurrency**: the single-threaded Flask server requires Gunicorn with multiple workers for concurrent requests
- **Memory**: 300–400 MB consumption fits within 512 MB free-tier limits
- **Segmentation**: processing time scales linearly with digit count
- **Database**: SQLite handles concurrent writes poorly; migration to PostgreSQL is recommended for production
- **Cold starts**: serverless platforms may take 2–5 seconds for the first request after inactivity

3.9 Security and Limitations

3.9.1 Input Validation

Several validation mechanisms protect the server against malformed or malicious inputs:

- **Size limit**: `MAX_CONTENT_LENGTH = 16 MB` — any request exceeding this limit is automatically rejected by Flask before processing
- **Image format validation**: Pillow raises an exception if the transmitted file is not a valid image; the exception is caught and returned as HTTP 400
- **Secure base64 decoding**: the JSON payload is handled inside a `try/except` block that prevents a server crash in the event of an invalid base64 string
- **SQL injection prevention**: all database queries use parameterized placeholders (?), eliminating injection risks

3.9.2 Admin Dashboard — Authentication Limitation

An important security limitation to acknowledge is that the administrator dashboard (`/admin`, `/api/admin/stats`, `/api/admin/clear`) is currently accessible without authentication. Any user who knows the URL can view prediction statistics or clear the database. This is acceptable for a local demonstration context, but represents a significant vulnerability in a public deployment. Adding HTTP Basic Authentication via Flask-HTTPAuth, or a token-based mechanism, is listed as a priority for future work.

3.9.3 Model Limitations in Real-World Use

The CNN model, which achieves 99.2% accuracy on the MNIST test set, has inherent limitations when faced with real-world inputs:

Limitation	**Impact**	**Mitigation**
Cultural writing styles (barred 1, open 4)	High	User education; uncertainty banner triggers
Non-uniform backgrounds (shadows, lines)	Medium	User advised to use clean background
Off-centre or very thin/thick digits	Medium	Canvas guidance; pipeline partially compensates
Connected or cursive digits	High	Fallback to single-digit mode; uncertainty warning
Inverted colours (light digit on dark background)	Severe	Manual inversion required; automatic detection not implemented
Non-digit characters (letters, symbols)	High	No mitigation; model not designed for this

Table 3.5: Model limitations and mitigation strategies

3.9.4 Grad-CAM Implementation Note

The Grad-CAM heatmap generation uses `model.get_layer(index=-4)` to access the last convolutional layer. This index-based approach is architecture-dependent: if the model structure changes, the index must be updated accordingly. Furthermore, if the Grad-CAM

computation fails for any reason (e.g., incompatible layer shapes), the system falls back to a randomly generated heatmap for demonstration purposes. This fallback is clearly logged server-side but is not surfaced to the user. A more robust implementation would reference the layer by name rather than by index.

3.9.5 Privacy and Data Retention

- All predictions are stored locally in the SQLite database. No data is transmitted to external servers
- Each prediction includes a thumbnail of the input image (base64, 56×56 pixels) for admin review
- The database can be cleared via the `/api/admin/clear` endpoint or by deleting the `predictions.db` file
- No user authentication is required, which simplifies use but means prediction history cannot be attributed to specific individuals without the session identifier

3.9.6 Recommendations for Robust Deployment

1. **User guidance**: display clear instructions (draw large, centred, on clean background)
2. **Image preprocessing improvements**: add automatic contrast enhancement and inversion detection
3. **Admin authentication**: implement HTTP Basic Auth or token-based access for the admin dashboard
4. **Model retraining**: extend the training dataset with European handwriting variants, augmented backgrounds, and synthetic distortions
5. **Ensemble approach**: combine CNN predictions with uncertainty estimation (e.g., Monte Carlo dropout)
6. **Production hardening**: implement rate limiting, request logging, and HTTPS

3.10 Achievements and future work

The development of NeuralDigit Pro has successfully demonstrated:

1. The feasibility of deploying a high-accuracy CNN model (99.2% on MNIST) in a production-ready web application
2. The critical importance of preprocessing consistency between training and inference phases
3. The value of explainability tools (Grad-CAM) and uncertainty indicators for user trust and model transparency

4. The effectiveness of a no-framework frontend approach, minimizing dependencies while delivering a modern, responsive user experience

5. The practicality of a lightweight, single-file deployment (Flask + SQLite) suitable for demonstration, educational, and small-scale production contexts

Several directions for future improvement have been identified:

- **Model enhancement**: retrain on extended datasets (EMNIST, Kuzushiji-MNIST) to improve robustness to diverse writing styles and add letter recognition
- **Admin security**: add HTTP Basic Authentication or token-based access to the admin dashboard
- **Real-time feedback**: implement active learning where uncertain predictions are flagged for human review and added to a retraining dataset
- **Performance optimization**: convert the Keras model to TensorFlow Lite or ONNX for faster inference and reduced memory footprint
- **Production hardening**: containerization with Docker, orchestration with Kubernetes, and CI/CD pipeline for automated testing
- **User accounts**: add authentication and personalized prediction history with persistent storage across sessions
- **Mobile application**: wrap the web interface in a native mobile app (React Native or Flutter) for touch-optimized digit recognition

3.11 Conclusion

NeuralDigit Pro provides a complete, transparent, and robust deployment of a handwritten digit recognition system. By combining a high-performance CNN (99.2% accuracy) with an intuitive user interface, explainable AI techniques (Grad-CAM), and production-grade features (database logging, admin monitoring, graceful degradation), the application serves both as a practical tool and as an educational reference for deploying machine learning models in real-world conditions.

This work contributes not only to the field of handwritten digit recognition, but also to the broader practice of deploying machine learning models as accessible, interactive, and explainable web applications

General Conclusion

In this project, we designed, implemented, and successfully deployed a comprehensive handwritten digit recognition system that bridges classical ML and modern DL techniques. We began by establishing a solid theoretical foundation in digital image preprocessing and neural network architectures. We then conducted a comparative study across four distinct classifiers using the complete MNIST dataset, which ultimately culminated in the development and deployment of our best-performing model inside our custom web application, NeuralDigit Pro. This work traces a clear engineering workflow, moving from initial algorithmic experimentation to building an interactive, user-facing interface.

Our experimental results established a clear performance hierarchy among the models we evaluated. Our CNN achieved the highest performance with a 99.2% test accuracy. This significantly outperformed our baseline models, including our SVM with an RBF kernel (98.1%), our Random Forest classifier (97.0%), and our k-NN implementation (96.8%). These benchmarks confirmed that DL models consistently excel at tasks requiring automated, hierarchical feature extraction. We successfully saved this optimal CNN architecture to export it directly into our production pipeline.

To transition our project from an offline notebook to a functional tool, we built NeuralDigit Pro using a lightweight Flask backend and a vanilla JavaScript frontend. We engineered the platform to enforce absolute preprocessing consistency between training and inference phases by subjecting all user inputs to an identical four-stage pipeline consisting of grayscale conversion, Lanczos resizing, Otsu binarization, and normalization. Furthermore, we integrated transparent explainability features, utilizing Grad-CAM activation heatmaps and per-class probability histograms to expose the model's decision-making process. By keeping our model compact ($\sim$1.6 MB) and saving logs to a local SQLite database, we ensured the entire application operates efficiently under a minimal 512 MB memory budget suitable for free-tier cloud deployment.

Despite these achievements, our system exhibits clear limitations. Because the CNN was trained exclusively on the North American MNIST dataset, it suffers from a cultural writing-style bias, occasionally misclassifying European variants like barred '1's or open '4's. Furthermore, our contour-based segmentation struggles with overlapping digits, the Otsu binarization remains sensitive to uneven lighting and lined paper backgrounds, and the administrative dashboard lacks an authentication layer.

To address these boundaries, future work will focus on retraining the model using the broader EMNIST dataset and implementing contrast-enhancement techniques to improve preprocessing robustness. On the engineering side, we plan to secure administrative endpoints with token-based authentication and containerize the application using Docker for a scalable CI/CD pipeline. Finally, we aim to convert the network to a TensorFlow Lite format to explore edge deployment on mobile platforms.

Bibliography

[1] S. Ahlawat, A. Choudhary, A. Nayyar, S. Singh, and B. Yoon, "Improved handwritten digit recognition using convolutional neural networks (cnn)," *Sensors*, vol. 20, no. 12, p. 3344, 2020.

[2] F. Stauder, "Ocr vs. htr or 'what is ai, actually?'," Transkribus.org, May 09, 2021, accessed Feb. 28, 2026. [Online]. Available: https://blog.transkribus.org/en/insights/ocr-vs-htr

[3] "Handwritten digit recognition - an overview | sciencedirect topics," https://www.sciencedirect.com/topics/computer-science/handwritten-digit-recognition.

[4] "Technology and organization of accounting: Recent changes," *International Journal of Recent Technology and Engineering*, vol. 8, no. 4, pp. 12 737–12 740, 2019.

[5] W. AlKendi, F. Gechter, L. Heyberger, and C. Guyeux, "Advancements and challenges in handwritten text recognition: A comprehensive survey," *Journal of Imaging*, vol. 10, no. 1, p. 18, 2024.

[6] A. A. M. A. A. M, I. Jossy, M. R. Bijoy, N. L. Koshy, and S. K. R. S. K. R, "Handwritten text recognition: A survey of ocr techniques," *International Journal of Advances in Engineering and Management*, vol. 6, no. 11, pp. 205–220, 2024.

[7] ——, "Handwritten text recognition: A survey of ocr techniques," *International Journal of Advances in Engineering and Management*, vol. 6, no. 11, pp. 205–220, 2024.

[8] "Robust handwriting recognition with limited and noisy data," ar5iv, 2019, accessed Feb. 28, 2026. [Online]. Available: https://ar5iv.labs.arxiv.org/html/2008.08148

[9] R. Singh, "Enhanced handwritten digit recognition using artificial neural networks," *Communications on Applied Nonlinear Analysis*, vol. 32, no. 2, pp. 772–782, 2025.

[10] D. House. Pixmaps and rgb color. Clemson University. Accessed: Feb. 28, 2026. [Online]. Available: https://people.computing.clemson.edu/~dhouse/courses/405/notes/pixmaps-rgb.pdf

[11] "Exploring the mnist digits dataset," https://towardsdatascience.com/exploring-the-mnist-digits-dataset-7ff62631766a/, accessed: 2025-01-01.

[12] A. S. University. (n.d.) Fundamentals of digital images and colour. Accessed: 28 February 2026. [Online]. Available: https://sparks.learning.asu.edu/videos/fundamentals-of-digital-images-and-colour

[13] E. Scientific. (n.d.) Digital image basics. Accessed: 28 February 2026. [Online]. Available: https://evidentscientific.com/en/microscope-resource/knowledge-hub/digital-imaging/digitalimagebasics

[14] Couchbase, "Data preprocessing in machine learning," https://www.couchbase.com/blog/data-preprocessing-in-machine-learning/, accessed: 2025-01-01.

[15] T. Cloud, "Techpedia article on data preprocessing," https://www.tencentcloud.com/techpedia/108579, accessed: 2025-01-01.

[16] Ultralytics, "What is normalization in machine learning?" 2026, accessed: May 26, 2026. [Online]. Available: https://www.ultralytics.com/glossary/normalization

[17] Keylabs.ai, "Best practices for image preprocessing in image classification," https://keylabs.ai/blog/best-practices-for-image-preprocessing-in-image-classification/, accessed: 2025-01-01.

[18] TechieFreak.org, "Image preprocessing techniques for computer vision," https://techiefreak.org/artificial-intelligence/image-preprocessing-techniques-for-computer-vision, accessed: 2025-01-01.

[19] M. Yousef and J. Allmer, *miRNomics: MicroRNA Biology and Computational Analysis*. Humana, 2016.

[20] A. Buckalew, "Supervised vs unsupervised learning explained - take control of ml and ai complexity," Take Control of ML and AI Complexity, 2025, mar. 09, 2025.

[21] F. Pereira, T. Mitchell, and M. Botvinick, "Machine learning classifiers and fmri: A tutorial overview," *NeuroImage*, vol. 45, no. 1, pp. S199–S209, 2009.

[22] M. Team, "Frontiers in artificial intelligence," Frontiers, 2022. [Online]. Available: https://www.frontiersin.org/journals/artificial-intelligence/articles

[23] ISO, "Iso - international organization for standardization," 2025, iso.org. [Online]. Available: https://www.iso.org/home.html

[24] Wikipedia, "Wikipedia, the free encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/Main_Page

[25] Ultralytics, "Revolutionizing the world of vision ai," 2001. [Online]. Available: https://www.ultralytics.com/

[26] GeeksforGeeks, "A computer science portal for geeks," 2024. [Online]. Available: https://www.geeksforgeeks.org/

[27] Meegle, "Free visual workflow project management tool," 2015. [Online]. Available: https://www.meegle.com/

[28] J. Heaton, "Ian goodfellow, yoshua bengio, and aaron courville: Deep learning," *Genetic Programming and Evolvable Machines*, vol. 19, no. 1–2, pp. 305–307, 2017.

[29] Z. Keita, "An introduction to convolutional neural networks (cnns)," Nov. 2023, dataCamp. [Online]. Available: https://www.datacamp.com/tutorial/introduction-to-convolutional-neural-networks-cnns

[30] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.

[31] F. Huot, "Convolutional neural networks explained," 2026, available at: https://www.semanticscholar.org/paper/Convolutional-neural-networks-explained-Huot/8e966341e5a63abb4afc3be990e2cccd0284f7a5 (accessed Mar. 30, 2026).

[32] P. Purwono, A. Ma'arif, W. Rahmaniar, H. I. K. Fathurrahman, A. Z. K. Frisky, and Q. M. ul Haq, "Understanding of convolutional neural network (cnn): A review," *International Journal of Robotics and Control Systems*, vol. 2, no. 4, pp. 739–748, 2023.

[33] R. S. District, "Mail (google mail inbox reference)," 2020, accessed Apr. 09, 2026. [Online]. Available: https://mail.google.com/

[34] Y. Xu and R. Goodacre, "On splitting training and validation set," 2018.

[35] "Nature article," 2021.

[36] "Training and validation split." [Online]. Available: https://pmc.ncbi.nlm.nih.gov/articles/PMC6373628/

[37] "Train test split." [Online]. Available: https://www.semanticscholar.org/paper/A-critical-look-at-the-current-train-test-split-in-Tan-Yang/bbcae47119a1b68f7899f4f59f470cb16d15d9fa

[38] T. et al., "A critical look at train/test split," 2021.

[39] "The analytics and applications of big data and machine learning," *Highlights in Science, Engineering and Technology (HSET)*, accessed: 2025-01-01. [Online]. Available: https://drpress.org/ojs/index.php/HSET/article/view/18946

[40] "Application of machine learning algorithms to predict thyroid disease risk: An experimental comparative study," *PeerJ Computer Science*, 2022, accessed: 2025-01-01. [Online]. Available: https://peerj.com/articles/cs-898/

[41] "Arxiv paper." [Online]. Available: https://arxiv.org/pdf/2203.08491

[42] R. O. Ogundokun, P. A. Owolawi, C. Tu, and E. van Wyk, "Autoencoder-assisted stacked ensemble learning for lymphoma subtype classification: A hybrid deep learning and machine learning approach," *Tomography*, vol. 11, no. 8, p. 91, 2025. [Online]. Available: https://www.mdpi.com/2379-139X/11/8/91

[43] "Elsevier article." [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1569843223001887

[44] B. N. Dhouha, "Handwritten digit recognition: Comparative analysis of ml, cnn, vision transformer, and hybrid models on the mnist dataset," *Directory of Open Access Journals*, 2024. [Online]. Available: https://doaj.org/article/7e5641d196504e92a5e4a06dc2457452

[45] "Handwritten digit recognition using machine learning and deep learning techniques: A comparative study of svm, knn, rfc, and cnn models," *International Journal of Research - GRANTHAALAYAH*, 2024. [Online]. Available: https://www.granthaalayahpublication.org/journals/granthaalayah/article/view/6111

[46] F. Chen, N. Chen, H. Mao, and H. Hu, "Assessing four neural networks on handwritten digit recognition dataset (mnist)," *arXiv preprint arXiv:1811.08278*, 2018.

[47] V. Teh, E. D. H. Wai, C. J. Cheng, J. C. Y. Loong, L. J. Yang, Z. A. bin Abdul Salam *et al.*, "Enhancing neural network models for mnist digit recognition," *Journal of Applied Technology and Innovation*, vol. 8, no. 1, pp. 15–20, 2024.