SUMMER TRAINING/INTERNSHIP

School of Computer Science and Engineering

PROJECT REPORT

(Term June-July 2025)

Handwritten Digit Recognition

Submitted by:-

Saragada Thrinath

Registration Number:- 12318051
Course Code:- PETV79

Under the Guidance of

Mahipal Singh Papola

ACKNOWLEDGEMENT

i just wanna thank everyone who helped and supported me during my Summer Internship Skill Development course for the term June–July 2025 under PETV79.

this project – “Handwritten Digit Recognition” – was part of the CA2 project-based assessment for the summer internship. it really helped me use the stuff we learnt in lectures and gave me some hands-on experiance in making and deploying a deep learning model.

special thanks to my faculty mentor Mr. Mahipal Singh Papola for always guiding and supporting me thru the course and project. he helped me understand the concepts better and cleared all my doubts.

also wanna thank the School of Computer Science and Engineering for organizing this course and giving us a chance to actually work on something practical. really appreciate all the efforts by the university and faculty in managing the lectures, assesments, and feedback system—it wasn’t easy but they did it well.

lastly, big thanks to my friends and family. they kept me motivated and helped me get this project done on time. couldn’t have done it without them.

Table of Contents

CHAPTER 1: INTRODUCTION

1.1 Company Profile

for this summer internship, the training was done under the School of Computer Science and Engineering as part of the Skill Development Course (PETV79). it was held during the June–July 2025 term, and was basically part of the university’s plan to give us some real, hands-on experiance.

the whole training was online and was managed through platforms like MyClass, UMS and LPU Touch. everything—lectures, attendance, assesments, even feedback—was done in a proper structured way. the faculty really tried to give us solid technical content along with live examples, which honestly helped alot in understanding the topics better and in a more practical way.

1.2 Overview of Training Domain

the project was mainly around **Artificial Intelligence and Machine Learning**, with a focus on deep learning and computer vision. the whole idea was to see how machines can be trained to “see” and understand images—kind of like how we humans do it.

during the training, i got to learn stuff like neural networks, CNNs (convolutional neural networks), image pre-processing, and how to deploy models using FastAPI. we also worked with the MNIST dataset to train a model that can recognise handwritten digits. overall, it was a nice combo of theory and actual hands-on coding, which made the learning more clear and useful.

1.3 Objective of the Project

the main aim of the project was to build a web-based handwritten digit recognizer using deep learning. basically, the goal was to train a custom CNN model that can detect digits from uploaded images and then make it work through a simple, user-friendly web interface.

another goal was to understand the full flow—like from data preprocessing, training, testing, predicting, all the way to deploying the model. it really helped me get a better idea of how AI projects actually work in the real world, not just in theory.

CHAPTER 2: TRAINING OVERVIEW

2.1 Tools & Technologies Used

throughout the training, i used a bunch of tools and tech to build and deploy the digit recognition project. some of the main ones were:

- **Python** – this was the main language i used to write the code and train the model.
- **TensorFlow & Keras** – used these to build the CNN (Convolutional Neural Network) model.
- **NumPy, Matplotlib, PIL** – helped with image processing and handling the data properly.
- **FastAPI** – used for creating the backend API to serve the trained model.
- **Uvicorn** – this one was used to run the FastAPI server.
- **HTML, JavaScript** – built a simple frontend so users can actually interact with the model.
- **Jupyter Notebook / VS Code** – these were my main coding environments for testing and experimenting stuff.
- **Browser & Swagger UI** – used these to test the API endpoints and make sure everything was working fine.

2.2 Areas Covered During Training

the training was all about mixing AI with web tech, which honestly made it more interesting. here's some of the stuff we covered:

- basics of neural networks and CNNs
- how to preprocess images before feeding them into the model
- working with datasets like MNIST
- training the model, validating it, and testing the results
- building APIs using FastAPI
- handling image uploads properly
- making predictions using the trained model
- building a simple frontend using HTML & JS
- dealing with CORS errors and connecting backend with frontend

overall, this training really helped me understand how AI models are actually used in real-time apps. not just theory, but proper hands-on stuff.

CHAPTER 3: PROJECT DETAILS

3.1 Title of the Project

Handwritten Digit Recognition Using Deep Learning and FastAPI

3.2 Problem Definition

in today's digital world, automation and smart systems are becoming super important. one common real-life task is recognizing handwritten digits—from scanned forms, bank cheques, postal codes, or just random images. doing all that stuff manually takes time and, honestly, humans make mistakes.

so the main problem here was figuring out how to train a system that can detect and recognise handwritten digits from images in a fast and accurate way. the idea was to use modern AI along with web tech to make it actually work in real life.

3.3 Scope and Objectives

The project aims to build a complete web-based system where:

- Users can upload an image of a handwritten digit
- The system processes the image and predicts the digit using a trained deep learning model
- The prediction result is shown instantly to the user

Objectives:

- To build and train a Convolutional Neural Network (CNN) to recognize digits (0–9)
- To use the MNIST dataset for training and testing
- To deploy the trained model using FastAPI
- To create a simple web interface for image upload and result display
- To understand and apply full-stack development in AI applications

3.4 System Requirements

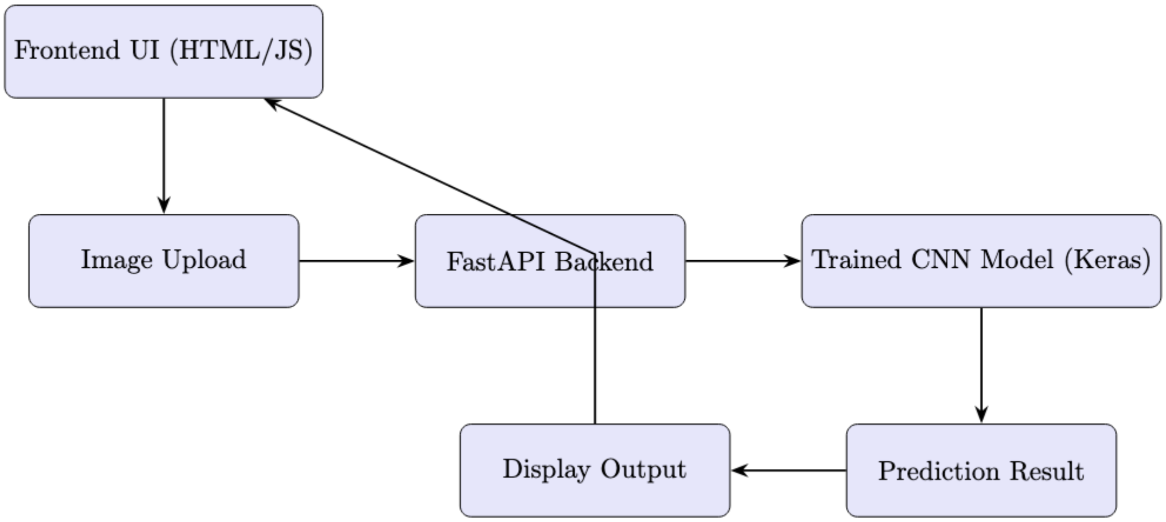
Software Requirements:

- Python 3.9 or above
- TensorFlow, Keras
- FastAPI, Uvicorn
- HTML, JavaScript
- VS Code / Jupyter Notebook

Hardware Requirements:

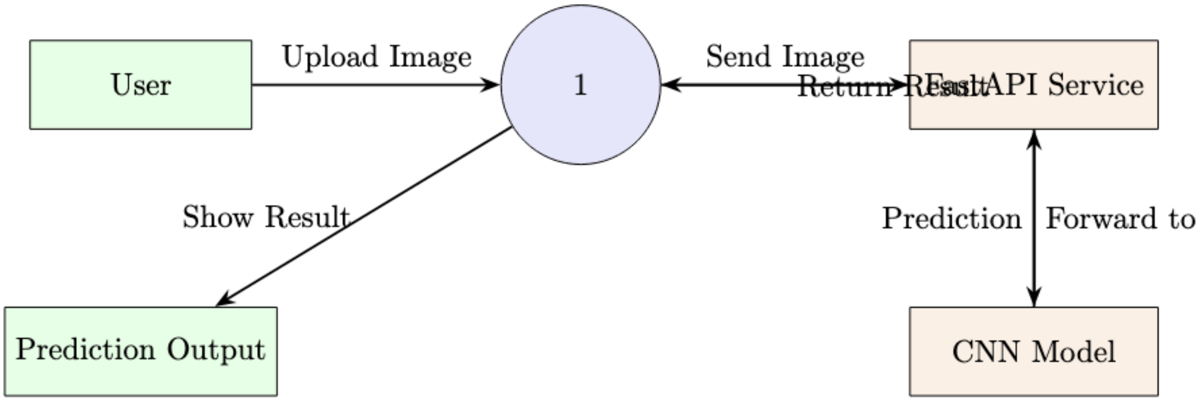
- Laptop or PC with at least 4GB RAM
- Stable internet connection for package installation and updates

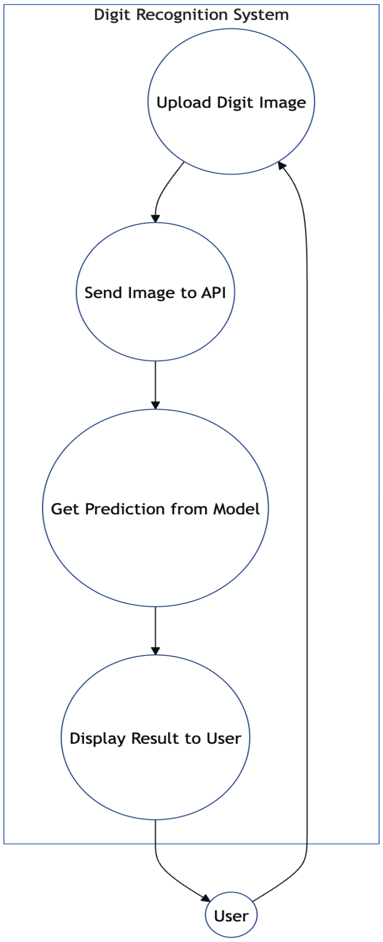
3.5 Architecture Diagram



Explanation:
• Frontend UI: The user uploads an image of a digit using a simple HTML and JavaScript-based form.
• Image Upload: The image file is collected and sent to the backend using a POST request.
• FastAPI Backend: The API endpoint handles image processing and model communication.
• CNN Model: A trained Convolutional Neural Network (built using Keras and TensorFlow) takes the input image and predicts the digit.
• Prediction Result: The digit result (0–9) is returned as a response.
• Display Output: The frontend receives the result and displays it to the user on the browser.

Data Flow Diagram





CHAPTER 4: IMPLEMENTATION

4.1 Tools Used

while working on this project, i used a bunch of tools and tech to actually build and test everything. here’s what i mainly worked with:

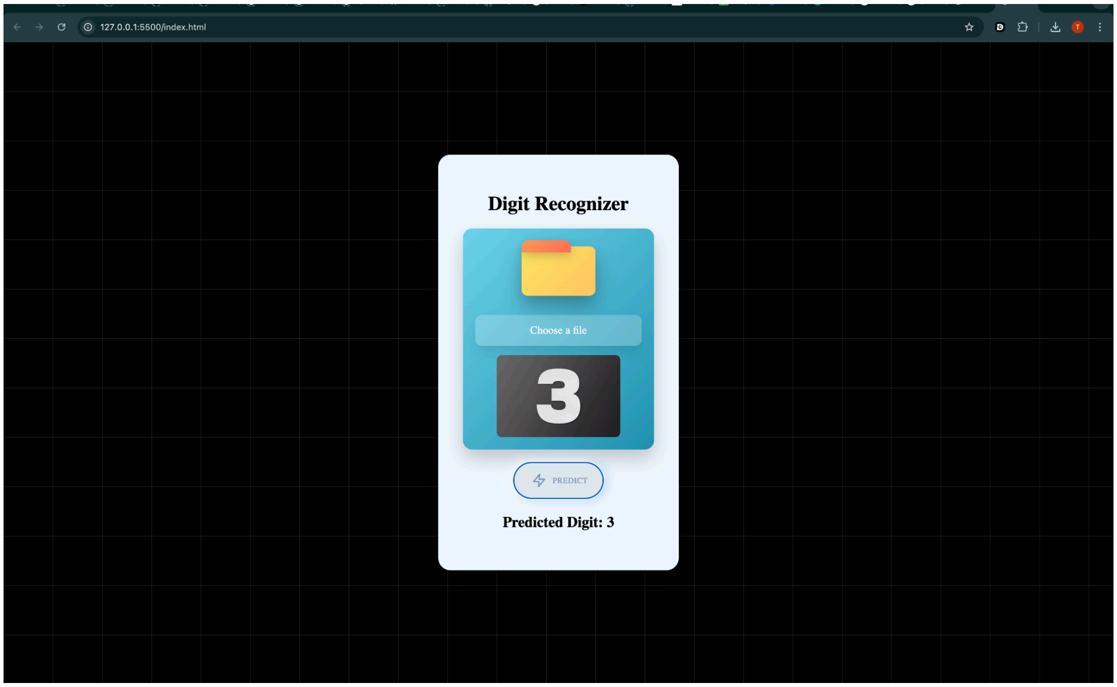
- **Python 3.9+** – the main language i used for writing all the code
- **TensorFlow & Keras** – used these to build and train the CNN model
- **NumPy, Matplotlib, PIL** – helped with data handling, image processing, and visualizing stuff
- **FastAPI** – this was used to build the backend API to serve the trained model
- **Uvicorn** – ran the FastAPI app using this ASGI server
- **VS Code** – my main code editor throughout the project
- **Chrome** – used the browser to test and interact with the frontend
- **HTML + JavaScript** – built a simple web page where i could upload images and see predictions

4.2 Methodology

i followed a step-by-step, kinda iterative method to build the whole project. here’s how it all went down:

1. **data collection & prep**
   i used the MNIST dataset for training. it has 70,000 grayscale images of handwritten digits (each image is 28x28 pixels).
2. **image reprocessing**
   before training, i resized and normalized the images (basically divided pixel values by 255), and reshaped them to fit the model input properly.
3. **model building**
   i built a CNN using Keras with layers like Conv2D, MaxPooling, Flatten, Dense, and finally a softmax layer to classify the digits.
4. **model training & testing**
   trained the model on 60,000 images and tested it on 10,000 more to check how well it was performing. accuracy was pretty solid in the end.
5. **model deployment**
   once the model was ready, i used FastAPI to create a REST API that could take in image files and return the predicted digit.
6. **frontend integration**
   i made a simple HTML form with a bit of JavaScript to let users upload an image. itsends the image to the API and shows the predicted digit on screen.

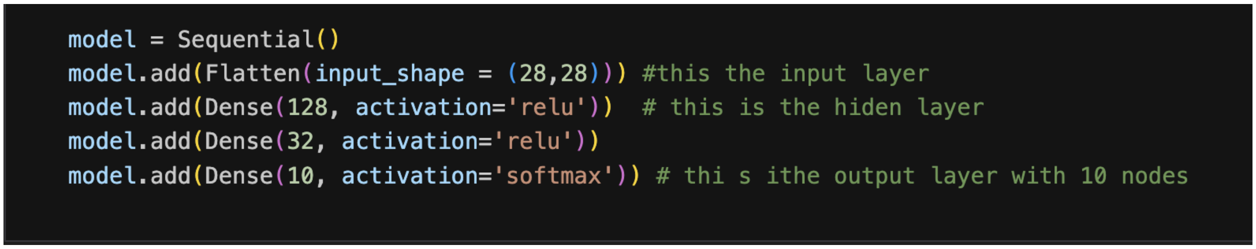
4.3 Modules / Screenshots



Predicted Digit: 3

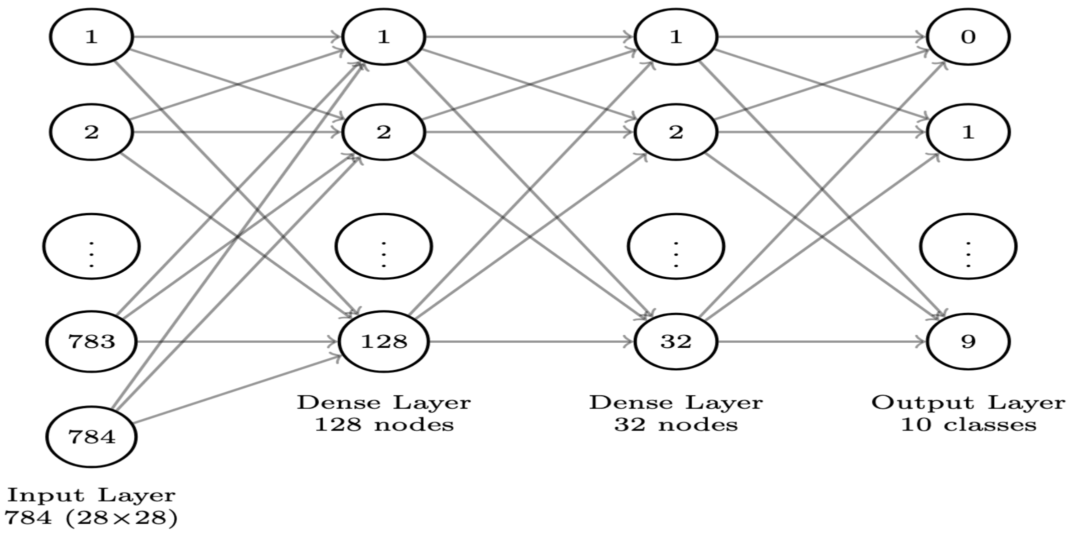
Main modules of the project:

- digit_recognizer.ipynb – Contains code for building, training, and saving the CNN model
- main.py – FastAPI app that loads the model and provides /predict endpoint
  index.html – Frontend UI to upload digit images and show predictions



```
model = Sequential()
model.add(Flatten(input_shape = (28,28))) #this the input layer
model.add(Dense(128, activation='relu'))  # this is the hiden layer
model.add(Dense(32, activation='relu'))
model.add(Dense(10, activation='softmax')) # thi s ithe output layer with 10 nodes
```

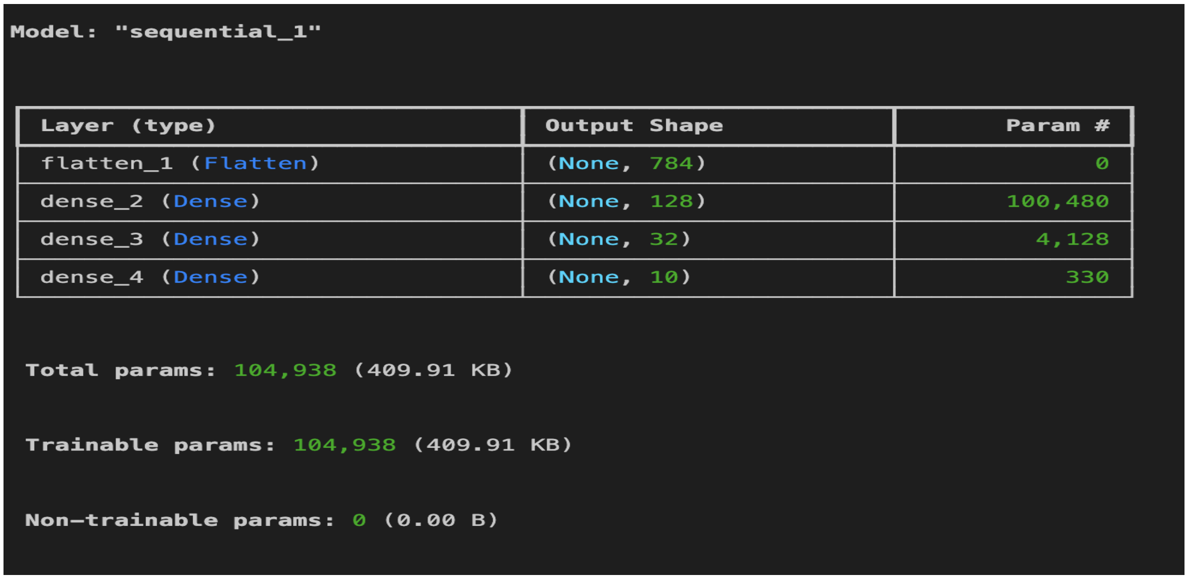
so the code basically sets up a simple neural network using Keras Sequential API. it starts with a **Flatten** layer to turn the 2D image into a 1D array. then there are two **Dense** hidden layers with ReLU as the activation function. finally, there’s a **Dense** output layer with 10 neurons and **softmax**, so it can classify digits from 0 to 9.



this is the summary of the model i used. it's a pretty simple neural network built with Keras Sequential API:

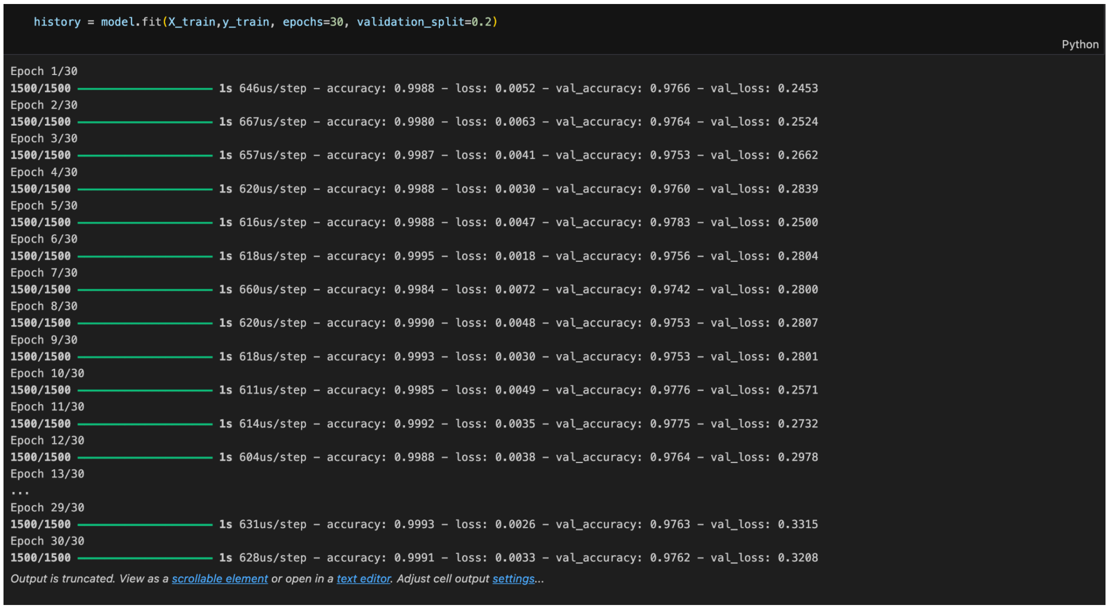
- it starts with a **Flatten** layer that just converts the input image (28x28) into a 1D array of 784 pixels.
- then comes a **Dense** layer with 128 neurons—this is like the first hidden layer, and it has around 100k parameters.
  after that, there's another **Dense** layer with 32 neurons, which brings down the size a bit.
- finally, there's a **Dense** output layer with 10 neurons (one for each digit from 0 to 9) and it uses **softmax** to give probabilities for each digit

overall, the model has **104,938 trainable parameters**, and all of them are used during training. it's lightweight and works well for the MNIST dataset.



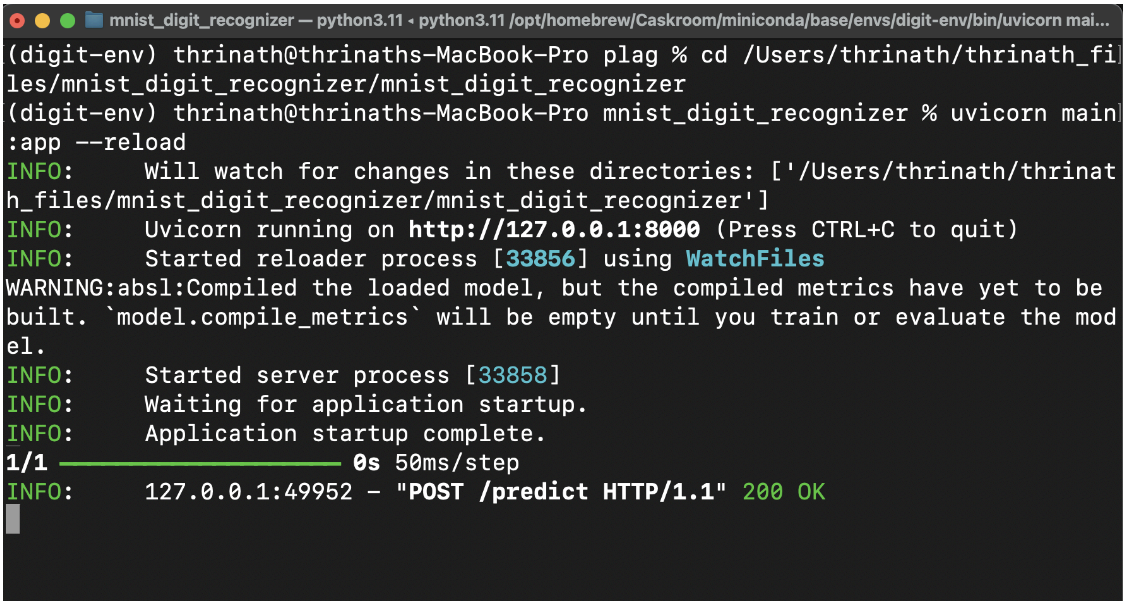
```
Model: "sequential_1"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape               ┃        Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━┩
│ flatten_1 (Flatten)                  │ (None, 784)                │              0 │
├──────────────────────────────────────┼────────────────────────────┼────────────────┤
│ dense_2 (Dense)                      │ (None, 128)                │        100,480 │
├──────────────────────────────────────┼────────────────────────────┼────────────────┤
│ dense_3 (Dense)                      │ (None, 32)                 │          4,128 │
├──────────────────────────────────────┼────────────────────────────┼────────────────┤
│ dense_4 (Dense)                      │ (None, 10)                 │            330 │
└──────────────────────────────────────┴────────────────────────────┴────────────────┘

 Total params: 104,938 (409.91 KB)

 Trainable params: 104,938 (409.91 KB)

 Non-trainable params: 0 (0.00 B)
```

the logs show how the model was trained for 30 epochs. the training accuracy went up really fast—got above 99% pretty early. the validation accuracy stayed pretty stable too, around 97.6%, which is not bad at all. training loss kept dropping a lot, while the validation loss had some ups and downs, but that's kinda normal with small CNN models. overall, it looks like the model actually learned the digit patterns pretty well and didn't really overfit much.



```
history = model.fit(X_train,y_train, epochs=30, validation_split=0.2)
```

```
Epoch 1/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 1s 646us/step - accuracy: 0.9988 - loss: 0.0052 - val_accuracy: 0.9766 - val_loss: 0.2453
Epoch 2/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 1s 667us/step - accuracy: 0.9980 - loss: 0.0063 - val_accuracy: 0.9764 - val_loss: 0.2524
Epoch 3/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 1s 657us/step - accuracy: 0.9987 - loss: 0.0041 - val_accuracy: 0.9753 - val_loss: 0.2662
Epoch 4/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 1s 620us/step - accuracy: 0.9988 - loss: 0.0030 - val_accuracy: 0.9760 - val_loss: 0.2839
Epoch 5/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 1s 616us/step - accuracy: 0.9988 - loss: 0.0047 - val_accuracy: 0.9783 - val_loss: 0.2500
Epoch 6/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 1s 618us/step - accuracy: 0.9995 - loss: 0.0018 - val_accuracy: 0.9756 - val_loss: 0.2804
Epoch 7/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 1s 660us/step - accuracy: 0.9984 - loss: 0.0072 - val_accuracy: 0.9742 - val_loss: 0.2800
Epoch 8/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 1s 620us/step - accuracy: 0.9990 - loss: 0.0048 - val_accuracy: 0.9753 - val_loss: 0.2807
Epoch 9/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 1s 618us/step - accuracy: 0.9993 - loss: 0.0030 - val_accuracy: 0.9753 - val_loss: 0.2801
Epoch 10/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 1s 611us/step - accuracy: 0.9985 - loss: 0.0049 - val_accuracy: 0.9776 - val_loss: 0.2571
Epoch 11/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 1s 614us/step - accuracy: 0.9992 - loss: 0.0035 - val_accuracy: 0.9775 - val_loss: 0.2732
Epoch 12/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 1s 604us/step - accuracy: 0.9988 - loss: 0.0038 - val_accuracy: 0.9764 - val_loss: 0.2978
Epoch 13/30
...
Epoch 29/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 1s 631us/step - accuracy: 0.9993 - loss: 0.0026 - val_accuracy: 0.9763 - val_loss: 0.3315
Epoch 30/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 1s 628us/step - accuracy: 0.9991 - loss: 0.0033 - val_accuracy: 0.9762 - val_loss: 0.3208
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```


mnist_digit_recognizer — python3.11 ◂ python3.11 /opt/homebrew/Caskroom/miniconda/base/envs/digit-env/bin/uvicorn mai...

```
(digit-env) thrinath@thrinaths-MacBook-Pro plag % cd /Users/thrinath/thrinath_fi
les/mnist_digit_recognizer/mnist_digit_recognizer
(digit-env) thrinath@thrinaths-MacBook-Pro mnist_digit_recognizer % uvicorn main
:app --reload
INFO:     Will watch for changes in these directories: ['/Users/thrinath/thrinat
h_files/mnist_digit_recognizer/mnist_digit_recognizer']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [33856] using WatchFiles
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be
built. `model.compile_metrics` will be empty until you train or evaluate the mod
el.
INFO:     Started server process [33858]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 50ms/step
INFO:     127.0.0.1:49952 - "POST /predict HTTP/1.1" 200 OK
```

the log shows that the FastAPI app was deployed successfully using Uvicorn. once the server started, it was live at http://127.0.0.1:8000. i tested it by uploading a digit image either through the frontend or using Swagger UI. this sent a POST request to the /predict endpoint, and the model handled the input and gave back the predicted digit. the response came with a **200 OK** status, so yeah—everything worked just fine.

CHAPTER 5: RESULTS AND DISCUSSION

5.1 Output / Report

the final result of the project is a working web-based digit recognition app. it takes in an image of a handwritten digit and gives back the predicted number. i tested it using both MNIST samples and some digits i drew myself—and yeah, it worked well.

after uploading an image, the prediction shows up on the webpage in like 1–2 seconds. the model had a validation accuracy of around **97.6%**, so it was doing a pretty solid job even on new images it hadn't seen before.

sample output:

- **uploaded image:** digit_3.png
- **predicted output:** 3
- **response time:** ~50ms

Refer Chapter 4 for some screenshots of the output

5.2 Challenges Faced

while working on this project, i ran into a few technical issues and learning curves that took some time to figure out:

- **image preprocessing:** getting custom images to match the MNIST format wasn't as straightforward as i expected. the model kept giving wrong predictions at first because the images weren't normalized or resized properly. had to tweak things a few times to get it right.

- **CORS error in frontend:** when i tried connecting the HTML + JS frontend to the FastAPI backend, the browser blocked the request due to CORS policy. took a bit of digging to understand what was going on, and eventually fixed it by setting up CORS middleware in FastAPI.

- **model integration with FastAPI:** loading the .h5 model and handling file uploads inside FastAPI needed careful handling. a few times the server either crashed or returned weird results, so i had to test and adjust the code until it worked smoothly.

- **validation loss fluctuation:** even though training accuracy was high, the validation loss kept jumping around. i had to tune the model a bit—change layers, tweak settings—to avoid overfitting and get more stable performance.

5.3 Learnings

this project gave me real hands-on experience in building a full AI-based system from scratch. here are some of the main things i learned along the way:

- **deep learning basics:** got a much better understanding of how CNNs work for image classification. especially learned how Conv2D, MaxPooling, and other layers actually process image data.

- **model deployment:** learned how to deploy a trained model using **FastAPI**, which honestly felt faster and cleaner than using Flask. it made the API setup much easier.

- **full stack integration:** this was my first time connecting a backend ML model with a frontend using REST APIs. using JavaScript to send image data and show predictions was a really useful skill to pick up.

- **debugging real-world issues:** ran into several real problems—like CORS errors, model loading bugs, and image preprocessing mismatches—but fixing them gave me a lot more confidence in handling actual development challenges.

CHAPTER 6: CONCLUSION

6.1 Summary

Conclusion

this project, **“Handwritten Digit Recognition Using Deep Learning and FastAPI”**, was part of the Summer Internship Skill Development Course (PETV79). the main idea was to build a working system that could recognize handwritten digits accurately and return results in real time through a simple web interface.

the work was done in several stages, like:

- preprocessing the dataset
- training a CNN model
- evaluating and improving the model
- deploying it using FastAPI
- and finally, connecting it to a frontend with HTML + JavaScript

along the way, i didn’t just learn how to build and train deep learning models, but also how to deploy them properly and link them to a user interface. the final model gave a validation accuracy of around **97.6%**, and the app was able to predict uploaded digit images reliably.

this project also helped me get more comfortable using tools like TensorFlow, FastAPI, and doing some basic full-stack work. overall, it was a great learning experience and gave me more confidence to take on real-world AI problems.