Technical Report for Python Final Project

Project Title: Handwritten Digit Recognition

Team Members: Eva Mescher, Sara Sethi, Sean Costello
Course: EECE 2140 – Computing Fundamentals for Engineers
Instructor: Dr. Fatema Nafa

December 5, 2025

Contents

Abstract

This project focuses on developing a complete handwritten digit recognition system using the MNIST dataset and combining machine learning with an interactive graphical interface. The goal was to build a neural network capable of identifying digits drawn by a user, while also designing a user-friendly workflow to preprocess images, save them, and generate predictions. The system consists of two major components: the MNIST model training script and a Gradio-based GUI. The model was implemented in TensorFlow, where the dataset was normalized, visualized, and used to train a three-layer neural network for six epochs. Image-processing steps—including grayscale conversion, binarization, cropping, resizing, and padding—were implemented using NumPy and PIL to ensure that user-drawn digits match MNIST's 28×28 input format.

To create an accessible interface, a Gradio GUI was developed to allow users to draw digits, save them, and run predictions. The interface connects to the trained model and automatically processes each saved image before classification. By looking at Fig. 2 and Fig. 3, we can conclude that the model acheived approximately 99 percent training accuracy and 97.99 percent validation accuracy, with steadily decreasing loss curves indicating stable learning. Overall, the project successfully integrates machine learning, image preprocessing, and user-interface design into a functional real time digit recognition system training accuracy and 97.99 percent validation accuracy, with steadily decreasing loss curves indicating stable learning. Overall, the project successfully integrates machine learning, image preprocessing, and GUI design into a functional system.

1 Introduction

Handwritten Digit Recognition project focuses on creating a system that can predict numbers drawn by users using a trained neural network model. Digit recognition is a key problem in machine learning because it represents the foundations of computer vision, pattern recognition, and automated data processing. Systems like these are used in real world applications such as bank check scanning, postal code reading, and digital form processing. Our project combines a trained MNIST model with a GUI, which allows the user to draw digits on a canvas, to demonstrate how machine learning can be applied interactively.

1.1 Student Learning Objectives

After completing this project, students should be able to:

- Eva: Learn how to use MNIST data set, learn how to create a ML model with greater than 90 percent accuracy, and help create a fun UI for the users.
- Sara: Learn the basics of machine learning, including how models are trained, tested and evaluated using datasets like MNIST.
- Sean: Learn how to use machine learning with models and data sets to create a desired outcome.

1.2 Team Objectives

The goals that the team was able to achieve were:

- Compare different models
- Create comprehensive game and user interface
- Stay on track with the project timeline

1.3 Report Structure

This technical report contains 7 sections. Section 1 is focused more on the introduction and the system diagram, which gives a system overview of the python-based Handwritten Digit Recognition project. Section 2 focuses more on the program architecture, which goes in depth with topics such as the modules/files used, functions used and the flow of execution. It also gives a compact overview of the pseudocode for the main components. Section 3 focuses on the data structures used, such as lists, dictionaries, NumPy arrays and PIL images. Section 4 is about the experimental setup. Section 5 focuses on the model training evaluation. Section 6 focuses on what we learned as a team, which includes model development, image processing, GUI design and debugging and integration. Section 7 concludes the technical report and includes improvements which can be made in the future to improve our user interface even more.

1.4 System Diagram

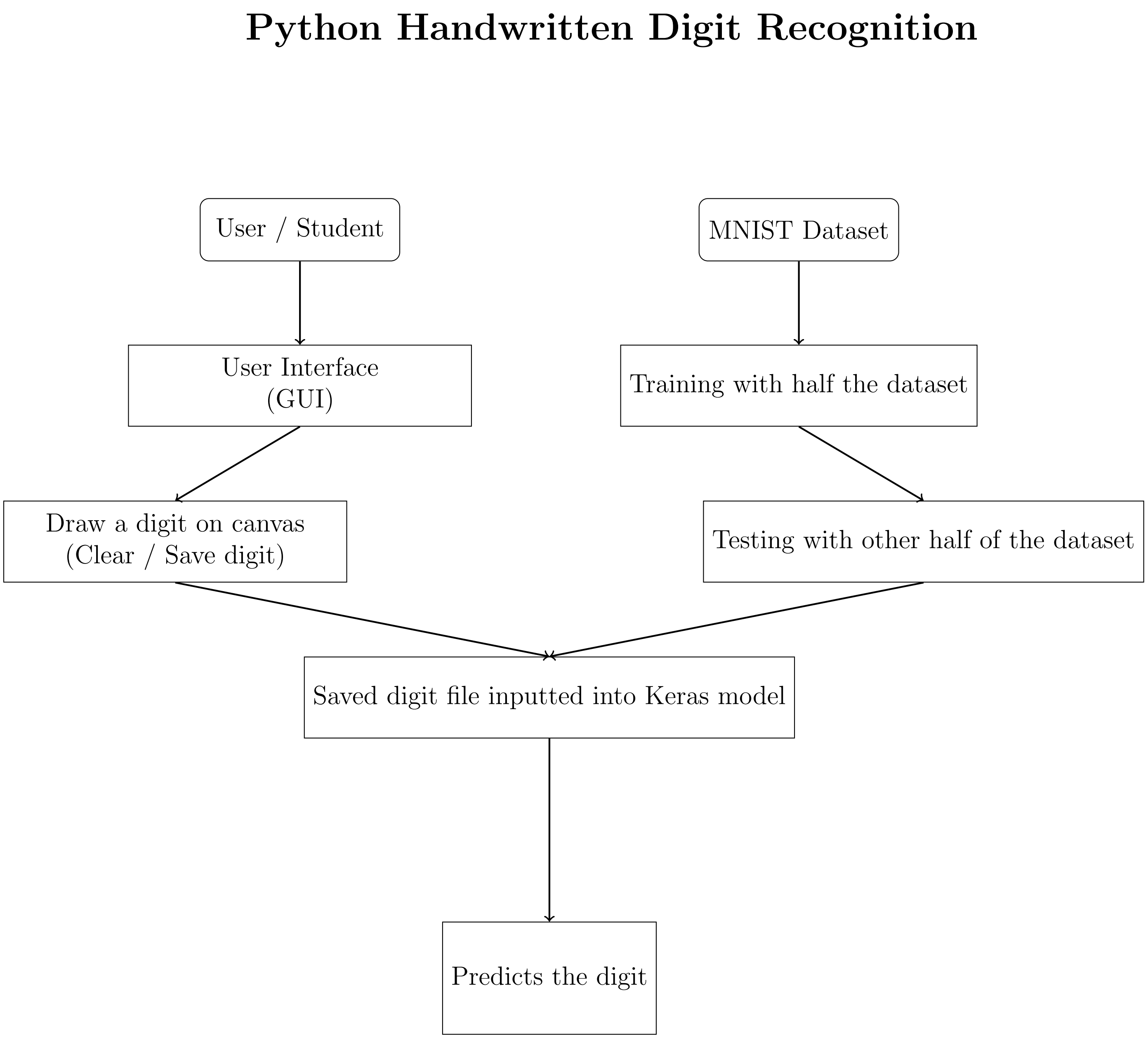

Python Handwritten Digit Recognition
User / Student
MNIST Dataset
User Interface (GUI)
Training with half the dataset
Draw a digit on canvas (Clear / Save digit)
Testing with other half of the dataset
Saved digit file inputted into Keras model
Predicts the digit

Figure 1: System overview of the Python-based Handwritten Digit Recognition.

2 Methodology

2.1 Program Architecture

The project consists of three major parts:
Modules/files:

- **MNIST Dataset:** This file loads the MNIST dataset using TensorFlow and performs all model-related tasks. The modules used are tensorflow, numpy, matplotlib.pyplot, os and cv2. Tensorflow was used to load MNIST, build the neural network, train, evaluate, and save the model. Numpy was used for preprocessing and array manipulation. Matplotlib.pyplot was used to visualize sample digits and plot training/validation graphs. OS2 was used to check file paths and save the trained model. CV2 was used for image inversion and preprocessing when testing or visualizing images. The notebook trains a 3-layer neural network for 6 epochs, achieving 99 percent training accuracy and 97.99 percent validation accuracy. The trained model is then saved as handwritten.keras.

- **GUI:** This Python file contains the Gradio user interface used to draw digits, save them, and run predictions using the trained MNIST model. It imports os, numpy, PIL, tensorflow and gradio. OS is used for file and folder operations. Numpy is used for image array processing. PIL is used for cropping, resizing, padding, and converting drawn digits. Tensorflow is used for loading the trained model. Gradio is used for creating the sketchpad and buttons in the GUI.

Classes or functions:

- **Functions:** There are no custom classes defined. The program uses helper functions. topil(img) was used to convert any input from the gradio sketchpad (dict, NumPy array, path, or PIL image) into a consistent PIL Image object. saveimg(img) was used to preprocess the drawn digit into MNIST format. This included flattening transparency, converting to grayscale, binarizing to black-digit/white background, finding the bounding box of the digit and cropping, resizing to 20px and padding to 28x28. After preprocessing, the digit is saved to digit1.png, digit2.png, etc. predictall() was used to open every saved image, convert it to a 28×28 array, invert pixel values (MNIST style), run the model prediction, and to return each filename with its predicted digit. clearcanvas() was used to return a blank 400x400 white image to reset the drawing canvas. clearsaveddigits() was used to delete all images stored in the digits/folder and to clear the prediction textbox.

Flow of execution:

- **Model:** It first loads the MNIST dataset. The images are then normalized to the [0,1] range. A 3-layer neural network is built using TensorFlow and trained for 6 epochs with validation. After training, the final model is saved as handwritten.keras.

- **GUI:** When the program starts, it loads the saved handwritten.keras model. It then creates a directory named digits/ to store processed drawings. The gradio interface is initialized with a drawing canvas, prediction textbox, status textbox, and buttons for clearing, saving, and predicting digits.

- **Saving a digit:** When the user draws on the canvas and clicks "Save Digit," the image is converted into a PIL image, binarized, cropped around the digit, resized to 20px, padded to a 28x28 MNIST style image, and saved in the digits/ folder with an auto-incrementing filename.
- **Running predictions:** When the user presses "Predict All Saved Digits," the script iterates through each saved image, converts it into a 28x28 numPy array, inverts the pixel values to match MNIST format, and passes it to the loaded model. The predicted digit for each file is displayed in the predictions textbox.
- **Clearing and resetting:** The "Clear Canvas" button returns a blank drawing canvas, and "Clear Saved Digits" removes all previously saved digit images from the digits/ directory and clears the prediction output.

2.2 Pseudocode for Main Components

Algorithm 1 Digit Drawing System – Compact Overview

```
Part 1: Image Normalization
function ToPIL(img)
    Convert array into PIL image
    return processed image or None
end function
Part 2: Saving a Digit
function SaveDigit(raw)
    pil ← ToPIL(raw);
    if None then return error
        Grayscale → binarize → find bounding box
        if no pixels then return "No digit"
            Crop → resize longest side to 20px → pad to $28 \times 28$
            Save as next "digitX.png" in /digits/
            return confirmation

Part 3: Predicting Saved Digits
            function PredictAll
                for each "digitX.png" do
                    Load $28 \times 28$ image, invert pixels
                    Predict using model, store digit and confidence
                end for
                return results or None
            end function
Part 4: Clearing Functions
            function ClearCanvas
                return blank $400 \times 400$ canvas
            end function
            function ClearSavedDigits
                Delete all files in /digits/
            end function
Part 5: UI Workflow
            procedure UI
                Display canvas and output box
                Buttons: ClearCanvas, SaveDigit, PredictAll, ClearSavedDigits
            end procedure
```

3 Data Structures Used

- **Lists:** used for storing results and coordinates. Since we aren't predicting a large number of digits at a time list were an easy way to store some of these predictions and results.
- **Dictionaries:** used for sketchpad return objects. Dictionaries were used along with out Gradio UI because they were a quick way to process user inputs.
- **NumPy Arrays:** used for image processing and model input. NumPy arrays are time efficient for pixel level operations and transformations.
- **PIL images:** used for image manipulation and storing. PIL images were used because they are highly optimized and memory efficient for common image operations.

4 Experimental Setup

Describe:

- We started building and training our model using the MINIST dataset on Google Colab. To test it's accuracy we imported different png files.
- Our first drawing canvas was built on Spyder. We resized these images and imported them into the model to test.
- Our next step aimed to combine these features into one GUI. To do this we utilized Google Colab and Gradio to create a user interface that could save the handwritten digit and input it into the model automatically to predict its value.

5 Results and Analysis

The accuracy was evaluated on the test dataset using the model.fit() command, and validation data. The training metrics of the model were recorded and the graphs were plotted, as shown in Fig. 2 and Fig. 3. Fig. 2 shows the accuracy of the model over six epochs and the blue line represents the training accuracy, which is increasing rapidly and reaches almost 99 percent by the final epoch. The orange line shows validation accuracy, which is also steadily rising and it stays close to the training line. This shows that the model is not overfitting and performs consistently on unseen data. Overall, the accuracy curve shows that the model learned effectively and generalizes well.
Fig. 3 shows the loss of the model over the same epochs. The blue line represents the training loss, which drops sharply, which means that the model is quickly reducing its errors during training. Validation loss, which is the orange line, is also decreasing, though it has a small bump around epoch 3. This kind of small fluctuation is normal and usually just reflects variations in the validation set. By the final epoch, both losses are low, which matches the high accuracy which can be seen in Fig. 2. This proves that the model is performing reliably with very few errors.

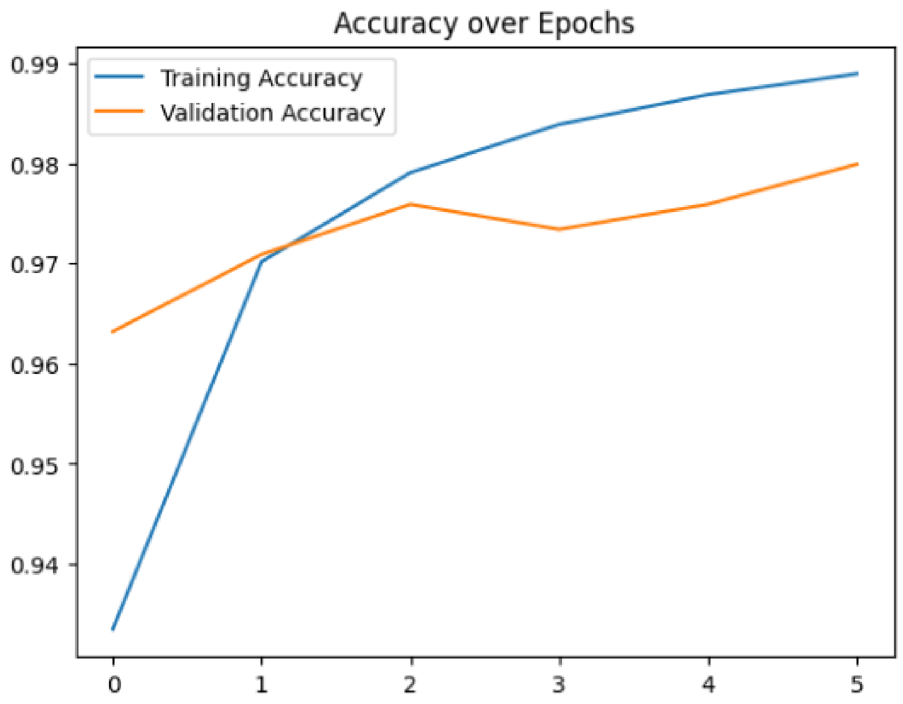


Figure 2: Accuracy of the model over 6 epochs

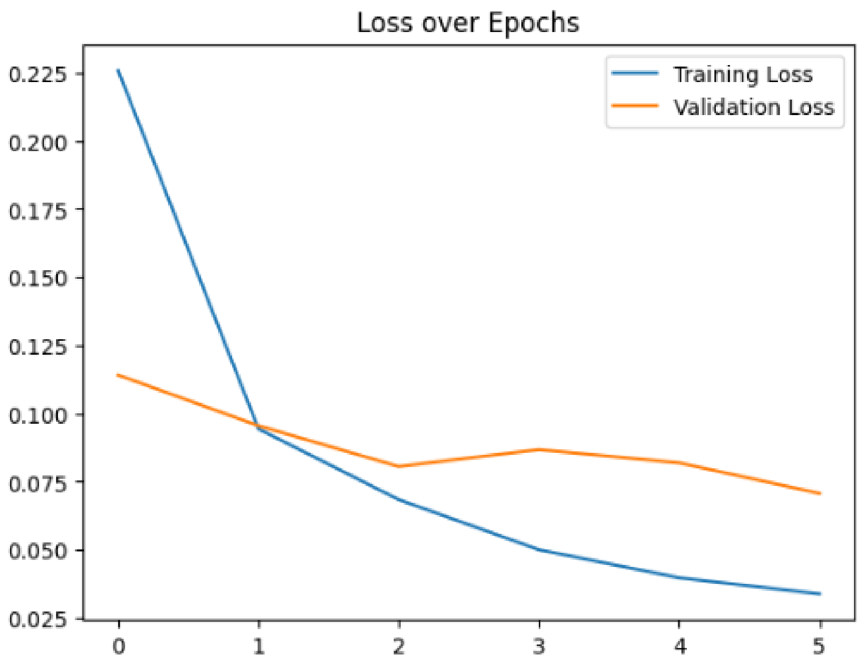


Figure 3: Loss of the model over 6 epochs

6 What We Learned

During this project, we learned how to:

- **Model development:** We learned how to load and preprocess the MNIST dataset, normalize data, build a multi-layer neural network, and evaluate model performance using accuracy and loss curves. We learned how to interpret these curves to determine whether a model is overfitting or generalizing properly. We each saved a copy of the digitrecognition1 file and worked individually on integrating the GUI with the model. Later, we combined our work into a single final program.
- **Image processing:** We gained experience and hands on practice using NumPy, PIL, and openCV to manipulate images. This included converting drawn images to grayscale, binarizing them, detecting bounding boxes, resizing the digit to the correct aspect ratio, and padding it to match MNIST's expected 28×28 format.
- **GUI design:** Using Gradio, we learned how to build an interactive graphical environment where users can draw digits, save them, and generate predictions. This required integrating model loading, file operations, and visual components into a smooth workflow.
- **Debugging and integration:** One of the biggest learning outcomes was understanding how to debug multi-module projects. We learned to trace errors step-by-step, verify image arrays at each stage, test functions independently, and coordinate between data preprocessing, GUI functions, and the model. Overall, the project taught us how to combine machine learning, image processing, and GUI development into a single, functioning application.

7 Conclusion and Future Work

Our project demonstrates a complete machine-learning inference pipeline that integrates image processing with neural network prediction. We designed our model to be able to handle multiple images automatically to allow the user to test multiple different digits at a time. One of the main improvements we could make in the future is expanding our dataset and allowing the users to input a wider variety of numbers. Our current model was trained on digits 0 through 9, so expanding this dataset could allow the user to input two digit numbers or even negative digits. Another improvement we could make to our model in the future would be to improve our user interface. We could develop the graphics further and allow the user to use a variety of color, rather than simply black and white. We could gamify our project in the future so the user has a task to complete or time pressure that allows them to compete with themselves or even the AI model.

References

[1] W3Schools, Matplotlib Pyplot Tutorial. https://www.w3schools.com/python/matplotlib_pyplot.asp

[2] NumPy Developers, NumPy Documentation. https://numpy.org/

[3] TensorFlow, Keras API Documentation. https://www.tensorflow.org/guide/keras

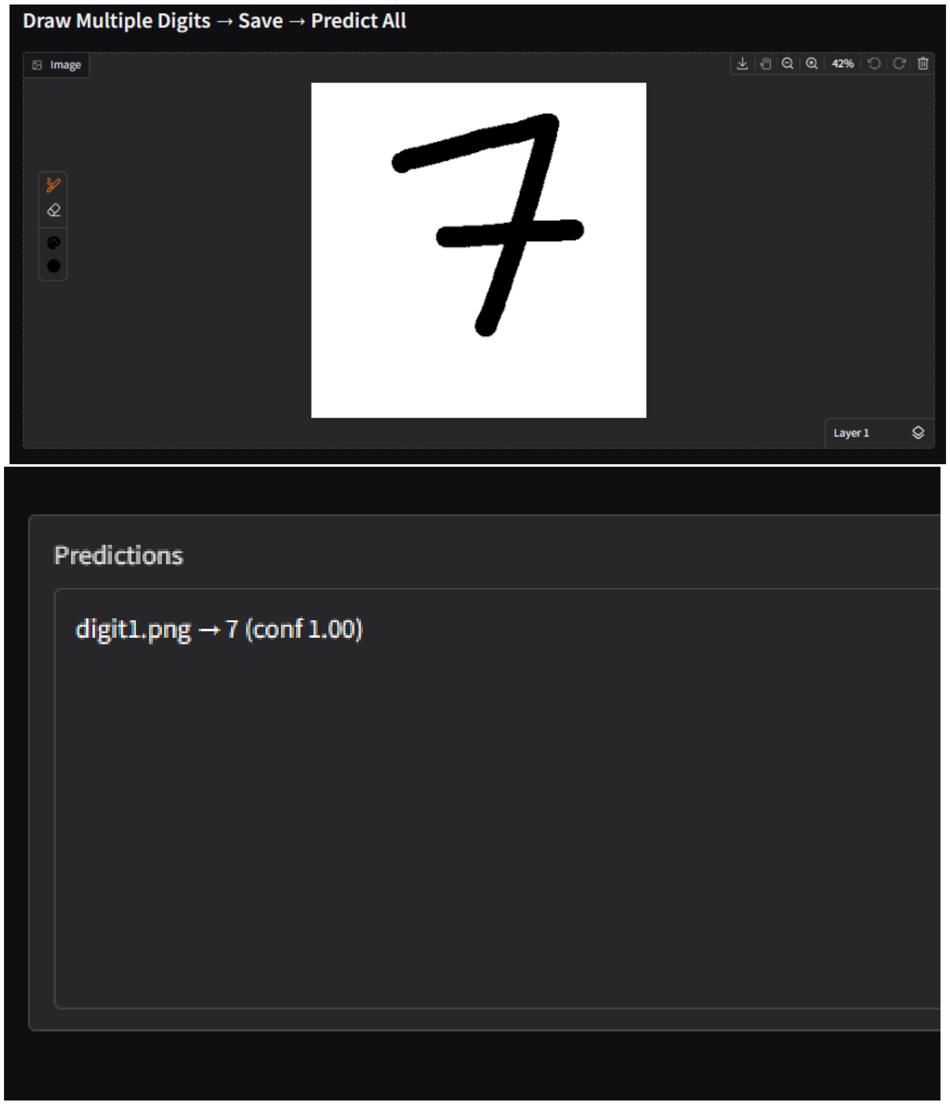
[4] TensorFlow Datasets, MNIST Dataset Documentation. https://www.tensorflow.org/datasets/catalog/mnist

A Appendix A: Source Code

```
import os
import numpy as np
import cv2
import matplotlib
import matplotlib.pyplot as plt
import tensorflow as tf

#The code below trains a neural network on the MNIST handwritten
   digits dataset

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.
   load_data() # Loads the MNIST dataset from TensorFlow    60,000
   training and 10,000 test images.
x_train = x_train.astype("float32") / 255.0
x_test  = x_test.astype("float32") / 255.0


model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)), # Flattens each
       28 28  image into a 1D vector of 784 pixels.
    tf.keras.layers.Dense(128, activation='relu'), # First hidden
       layer with 128 neurons using ReLU activation
    tf.keras.layers.Dense(128, activation='relu'), # Second hidden
       layer with 128 neurons using ReLU activation.
     tf.keras.layers.Dense(10, activation='softmax') # Output layer:
        10 neurons (for digits 0  9 ), softmax gives probabilities.
 ])

model.compile(optimizer='adam', loss='
   sparse_categorical_crossentropy', metrics=['accuracy']) #
   Compiles the model: Adam optimizer + cross-entropy loss (for
   integer labels).
```

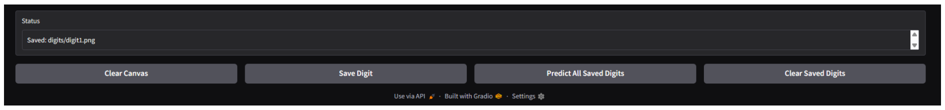
```
model.fit(x_train, y_train, epochs=6, validation_data=(x_test,
   y_test)) # Trains the model for 6 epochs and checks performance
   on test data each epoch.


model.save('handwritten.keras')   # saves the model

m2 = tf.keras.models.load_model('handwritten.keras') # loads
   previously saved model


image_number = 1 # stats with digit number 1
while os.path.isfile(f"digits/digit{image_number}.png"):  # Whiel
   there are more digits in the folder
  try:

    img = cv2.imread(f"digits/digit{image_number}.png")[:,:,0] #
       Reads the image
    img = np.invert(np.array([img])) # Processes the imgae by
       inverting colors and wraps it in another array
    prediction = m2.predict(img) # The model outputs a vector of 10
       probability values (for digits 0  9 ).
    print(f"This␣digit␣is␣probably␣a␣{np.argmax(prediction)}") # np.
       argmax() finds the index (0  9 ) with the highest probability
            the predicted digit.
    plt.imshow(img[0], cmap=plt.cm.binary) #Shows the image
    plt.show()
  except:
    print("Error!") #If there's an error print error
  finally:
    image_number += 1 # Increase the image number so the loop moves
       to the next file

#final
import os
import numpy as np
import gradio as gr
from PIL import Image, ImageOps
import tensorflow as tf

# -----------------------------
# Load your trained model
# -----------------------------
m2 = tf.keras.models.load_model("handwritten.keras")

# -----------------------------
# Folder for saved digits
# ------------------------------
SAVE_DIR = "digits"   # relative path: ./digits
os.makedirs(SAVE_DIR, exist_ok=True)

# ------------------------------
# Helper: normalize image to PIL
# ------------------------------
def _to_pil(img):
    if img is None:
        return None

    # Sketchpad sometimes returns a dict
    if isinstance(img, dict):
        for key in ("composite", "image", "background"):
            if key in img and img[key] is not None:
                return _to_pil(img[key])
        return None

    # Already a PIL image
    if isinstance(img, Image.Image):
        return img

    # Path string
    if isinstance(img, str):
        return Image.open(img)

    # NumPy array
    if isinstance(img, np.ndarray):
        arr = img
        if arr.dtype in (np.float32, np.float64):
            arr = (np.clip(arr, 0, 1) * 255).astype(np.uint8)
        elif arr.dtype != np.uint8:
            arr = arr.astype(np.uint8)

        if arr.ndim == 2:
            return Image.fromarray(arr, mode="L")
        if arr.ndim == 3:
            if arr.shape[2] == 4:
                return Image.fromarray(arr, mode="RGBA")
            if arr.shape[2] == 3:
                return Image.fromarray(arr, mode="RGB")

        return Image.fromarray(arr)

    return None

# ------------------------------
# Save one drawn digit to file
# ------------------------------
def save_img(img):
    pil = _to_pil(img)
    if pil is None:
        return "Draw␣something␣first."

    # Flatten transparency
    if pil.mode in ("RGBA", "LA"):
        bg = Image.new("RGB", pil.size, "white")
        alpha = pil.split()[-1]
        bg.paste(pil, mask=alpha)
        pil = bg
    else:
        pil = pil.convert("RGB")

    # Grayscale
    img = pil.convert("L")
    arr = np.array(img)

    # Binarize (dark = digit)
    arr = np.where(arr < 200, 0, 255).astype(np.uint8)

    # Find bounding box of digit
    coords = np.column_stack(np.where(arr < 255))
    if coords.size == 0:
        return "No␣digit␣drawn!"

    y_min, x_min = coords.min(axis=0)
    y_max, x_max = coords.max(axis=0)

    # Crop tightly around digit
    img = Image.fromarray(arr).crop((x_min, y_min, x_max + 1, y_max
       + 1))

    # Resize longest side to 20 px (MNIST-ish)
    w, h = img.size
    if w > h:
        new_h = int((20 / w) * h)
        img = img.resize((20, new_h), Image.LANCZOS)
    else:
        new_w = int((20 / h) * w)
        img = img.resize((new_w, 20), Image.LANCZOS)

    # Pad to 28x28, centered
    w, h = img.size
    pad_left = (28 - w) // 2
    pad_top = (28 - h) // 2
    pad_right = 28 - w - pad_left
    pad_bottom = 28 - h - pad_top

    img = ImageOps.expand(img, border=(pad_left, pad_top, pad_right,
        pad_bottom), fill=255)

    img_arr = np.array(img).astype("uint8")

    # Auto-increment filename
    i = 1
    while os.path.exists(os.path.join(SAVE_DIR, f"digit{i}.png")):
        i += 1
    filename = os.path.join(SAVE_DIR, f"digit{i}.png")

    Image.fromarray(img_arr).save(filename)

    return f"Saved:␣{filename}"

# ----------------------------
# Predict all saved digits
# ----------------------------
def predict_all():
    results = []
    image_number = 1
    found_any = False

    while True:
        path = os.path.join(SAVE_DIR, f"digit{image_number}.png")
        if not os.path.isfile(path):
            break

        found_any = True
        try:
            # Load as grayscale 28x28
            pil = Image.open(path).convert("L")
            arr = np.array(pil).astype(np.uint8)  # shape (28,28)

            # Match your original logic:
            # img = cv2.imread(... )[:,:,0]
            # img = np.invert(np.array([img]))
            x = np.invert(np.array([arr]))  # shape (1, 28, 28)

            prediction = m2.predict(x, verbose=0)
            pred_digit = int(np.argmax(prediction[0]))
            conf = float(np.max(prediction[0]))
            results.append(f"{os.path.basename(path)}␣   ␣{
               pred_digit}␣(conf␣{conf:.2f})")
        except Exception as e:
            results.append(f"{os.path.basename(path)}␣   ␣Error:␣{e}
               ")

        image_number += 1

    if not found_any:
        return "No␣digits␣saved␣yet."

    return "\n".join(results)

# -----------------------------
# Clear canvas helper
# -----------------------------
def clear_canvas(_):
    # Return a blank white 400x400 RGB image
    return 255 * np.ones((400, 400, 3), dtype=np.uint8)

# -----------------------------
# Clear all saved images
# -----------------------------
def clear_saved_digits():
    count = 0
    for fname in os.listdir(SAVE_DIR):
        fpath = os.path.join(SAVE_DIR, fname)
        if os.path.isfile(fpath):
            os.remove(fpath)
            count += 1
    return f"Deleted␣{count}␣saved␣digit␣file(s).", ""  # status,
       empty predictions

# -----------------------------
# Gradio UI
# -----------------------------
with gr.Blocks() as demo:
    gr.Markdown("##␣Draw␣Multiple␣Digits␣   ␣Save␣   ␣Predict␣All")

    with gr.Row():
        pad = gr.Sketchpad(
            image_mode="RGB",
            width=400,
            height=400
        )
        predictions_box = gr.Textbox(
            label="Predictions",
            lines=10,
            interactive=False
        )

    status = gr.Textbox(label="Status", interactive=False)

    with gr.Row():
        clear_btn = gr.Button("Clear␣Canvas")
        save_btn = gr.Button("Save␣Digit")
        predict_btn = gr.Button("Predict␣All␣Saved␣Digits")
        clear_files_btn = gr.Button("Clear␣Saved␣Digits")

    clear_btn.click(fn=clear_canvas, inputs=pad, outputs=pad)
    save_btn.click(fn=save_img, inputs=pad, outputs=status)
    predict_btn.click(fn=predict_all, outputs=predictions_box)
    clear_files_btn.click(fn=clear_saved_digits, outputs=[status,
       predictions_box])

demo.launch()
```

B Appendix B: Additional Figures and Tables



digit1.png → 7 (conf 1.00)

Figure 4: digit 7 predicted



Saved: digits/digit1.png

Figure 5: buttons