**Phase-3 Submission Template**

**Student Name:** [Enter Your Name]

**Register Number:** [Enter Your Register Number]

**Institution:** [Insert College Name]

**Department:** [Enter Your Department Name]

**Date of Submission:** [Insert Date]

**Github Repository Link:** [Update the project source code to your Github Repository]

### **1. Problem Statement**

* Build a deep learning model to accurately recognize and classify handwritten digits (0–9) from image data.
* **Motivation**
* Handwritten digit recognition is crucial for automating manual data entry, digitizing forms, postal mail sorting, and enabling assistive tech.
* **Challenges**
* Handwriting varies across individuals in style, slant, thickness, and size.
* Must ensure robustness across diverse inputs and noisy images.
* **Dataset**
* Use the **MNIST dataset**: 70,000 grayscale images of size 28x28 pixels.
* 60,000 for training and 10,000 for testing.
* **Solution Approach**
* Utilize a **Convolutional Neural Network (CNN)** architecture for feature extraction and classification.
* Implement using TensorFlow/Keras or PyTorch.
* **Evaluation Metrics**
* Accuracy, confusion matrix, and per-class performance.
* Optional: model latency and performance on edge devices.
* **Expected Outcome**
* A trained deep learning model that can classify digits in real-time with high accuracy (>98% on test data).
* **Applications**
* Optical Character Recognition (OCR)
* Smart forms and check processing
* Autonomous systems requiring numerical input recognition
* Educational tools and digital handwriting apps

### **2. Abstract**

*This project focuses on recognizing handwritten digits using deep learning techniques to enhance smarter AI applications. The problem addressed is the accurate identification of handwritten numerical characters, which is critical for applications such as automated form processing, postal mail sorting, and educational tools. The objective is to build a robust convolutional neural network (CNN) model that can classify digits from 0 to 9 with high accuracy. The approach involves training the CNN on the well-known MNIST dataset, leveraging multiple convolutional and pooling layers to extract meaningful features, followed by dense layers for classification. The model is optimized using the Adam optimizer and trained over multiple epochs to ensure convergence. The outcome is a highly accurate digit recognition model capable of real-time predictions, which can be easily deployed in practical AI systems to facilitate automated digit reading. This project demonstrates the effectiveness of deep learning in image recognition tasks and its potential for integration into smarter AI applications.*

### **3. System Requirements**

***Hardware:***

* Minimum RAM: 8 GB (recommended 16 GB for faster training)
* Processor: Multi-core CPU (Intel i5 or above recommended)
* GPU: Optional but highly recommended (NVIDIA GPU with CUDA support) for accelerated training and inference
* Storage: At least 2 GB free space to store datasets and model files

**Software:**

* Operating System: Windows 10 / macOS / Linux (Ubuntu recommended)
* Python Version: 3.7 or higher
* Required Libraries:
  + TensorFlow (2.x) or Keras
  + NumPy
  + Matplotlib
  + OpenCV (optional, for image preprocessing)
  + Gradio (for deployment if needed)
* IDE/Environment:
  + Jupyter Notebook or JupyterLab for interactive development
  + Google Colab (cloud-based, with free GPU access) recommended for users without GPU hardware
  + VS Code or PyCharm for script-based development

### **4. Objectives**

**Develop an Accurate Deep Learning Model**

* Design and train a **Convolutional Neural Network (CNN)** to classify handwritten digits (0–9) with high precision.
* **Automate Handwritten Digit Recognition**
* Enable computers to interpret and digitize handwritten input, reducing the need for manual data entry.
* **Learn and Apply Deep Learning Techniques**
* Gain hands-on experience with deep learning concepts such as convolutional layers, activation functions, pooling, and dropout.
* **Work with Real-World Image Data**
* Use and explore the **MNIST dataset** to understand preprocessing, normalization, and visual patterns in image data.
* **Evaluate and Optimize Model Performance**
* Measure the model’s accuracy, precision, recall, and confusion matrix, and fine-tune hyperparameters to improve results.
* **Deploy a Practical AI Solution**
* Lay the groundwork for integrating the model into real-world applications like OCR software, smart forms, or educational tools.
* **Promote Smarter AI Capabilities**
* *Demonstrate how AI can mimic human perception and handwriting understanding, contributing to more intelligent and human-like systems*

**5. Flowchart of Project Workflow**

### *+--------------------------+*

*| 1. Import Libraries |*

*| - TensorFlow/Keras |*

*| - Numpy, Matplotlib |*

*+--------------------------+*

*↓*

*+--------------------------+*

*| 2. Load Dataset |*

*| - Use MNIST (from |*

*| Keras.datasets) |*

*+--------------------------+*

*↓*

*+--------------------------+*

*| 3. Preprocess Data |*

*| - Normalize pixel values |*

*| - Reshape arrays |*

*| - One-hot encode labels |*

*+--------------------------+*

*↓*

*+--------------------------+*

*| 4. Visualize Data (EDA) |*

*| - Plot sample digits |*

*| - Check class balance |*

*+--------------------------+*

*↓*

*+--------------------------+*

*| 5. Build Model |*

*| - CNN architecture |*

*| - Conv2D, MaxPooling, |*

*| Flatten, Dense layers |*

*+--------------------------+*

*↓*

*+--------------------------+*

*| 6. Compile Model |*

*| - Set optimizer (e.g., |*

*| Adam), loss (e.g., |*

*| categorical\_crossentropy) |*

*+--------------------------+*

*↓*

*+--------------------------+*

*| 7. Train Model |*

*| - Train with validation |*

*| - Use callbacks if needed|*

*+--------------------------+*

*↓*

*+--------------------------+*

*| 8. Evaluate Model |*

*| - Accuracy, Confusion |*

*| Matrix, Classification |*

*| Report |*

*+--------------------------+*

*↓*

*+--------------------------+*

*| 9. Save/Export Model |*

*| - .h5 or SavedModel format|*

*+--------------------------+*

*↓*

*+--------------------------+*

*| 10. Deployment (optional)|*

*| - Use Streamlit/Flask to |*

*| create interface |*

*+--------------------------+*

### **6. Dataset Description**

* **MNIST Dataset (Modified National Institute of Standards and Technology)**
* **Description**: The most widely used benchmark dataset for handwritten digit recognition. Contains 70,000 labeled grayscale images (60,000 training + 10,000 test).
* **Download Link**:
  + From Keras (automatically downloads):
* python
* CopyEdit
* from tensorflow.keras.datasets import mnist
* (X\_train, y\_train), (X\_test, y\_test) = mnist.load\_data()
  + Or from Yann LeCun’s website
* **Kaggle Datasets**
* **Description**: Kaggle provides MNIST and variations (e.g., MNIST in CSV format, augmented versions).
* **Examples**:
  + MNIST in CSV format
  + Digit Recognizer Competition
* **Extended MNIST (EMNIST)**
* **Description**: An extension of MNIST that includes **letters (A–Z)** and digits (0–9), useful for extending your model to alphanumeric recognition.
* 📎 **Link**: <https://www.nist.gov/itl/products-and-services/emnist-dataset>
* **Synthetic Digit Datasets**
* **Description**: You can generate synthetic digit images using tools like OpenCV, PIL, or fonts to simulate handwritten input for augmentation or experimentation.
* **Google Quick, Draw! Dataset**
* **Description**: A massive dataset of doodles including digits and objects, drawn by real users.
* ***Link****:* [*https://quickdraw.withgoogle.com/data*](https://quickdraw.withgoogle.com/data)



### 

### **7. Data Preprocessing**

* **Collect and Understand the Dataset**
* Use the **MNIST dataset** as the primary data source.
* Understand image format (28x28 pixels, grayscale) and label structure (digits 0–9).
* **Data Preprocessing**
* Normalize pixel values (0–255 → 0–1).
* Reshape input data for compatibility with deep learning models.
* One-hot encode the digit labels for classification.
* **Design the Deep Learning Model**
* Use a **Convolutional Neural Network (CNN)** architecture for feature extraction and classification.
* Include layers like Conv2D, MaxPooling, Flatten, Dense, and Dropout to enhance performance and prevent overfitting.
* **Train the Model**
* Split data into training and validation sets.
* Train the CNN using an appropriate optimizer (e.g., Adam) and loss function (e.g., categorical crossentropy).
* Monitor training with accuracy and loss graphs.
* **Evaluate the Model**
* Test the model on the test dataset to measure accuracy, precision, recall, and F1-score.
* Visualize results with confusion matrix and sample predictions.
* **Model Optimization**
* Tune hyperparameters such as learning rate, number of filters, and epochs.
* Use regularization techniques like dropout and data augmentation if needed.
* **Model Deployment (Optional)**
* Save the trained model in .h5 or .pt format.
* Deploy using a web framework (e.g., Flask, Streamlit) or integrate into a mobile app for real-time digit recognition.
* **Applications and Future Scope**
* Use in OCR systems, smart forms, educational tools, postal services, etc.

### **8. Exploratory Data Analysis (EDA)**

#### 1. **Check Dataset Shape and Distribution**

* print(f"Training data shape: {X\_train.shape}")
* print(f"Test data shape: {X\_test.shape}")
* print(f"Training labels shape: {y\_train.shape}")

#### 2. **Visualize Label Distribution**

* import seaborn as sns
* import matplotlib.pyplot as plt
* import numpy as np
* # Convert one-hot to labels if needed
* labels = np.argmax(y\_train, axis=1)
* sns.countplot(x=labels)
* plt.title("Distribution of Digits in Training Set")
* plt.xlabel("Digit")
* plt.ylabel("Count")
* plt.show()
* **Insight**: The MNIST dataset is balanced — each class (0–9) has ~6,000 samples.

#### 3. **Visualize Sample Images from Each Class**

* plt.figure(figsize=(10, 4))
* for i in range(10):
* plt.subplot(2, 5, i+1)
* index = np.where(labels == i)[0][0]
* plt.imshow(X\_train[index].reshape(28, 28), cmap='gray')
* plt.title(f"Digit: {i}")
* plt.axis('off')
* plt.suptitle("Sample Images of Each Digit")
* plt.show()

#### 4. **Check Pixel Value Range**

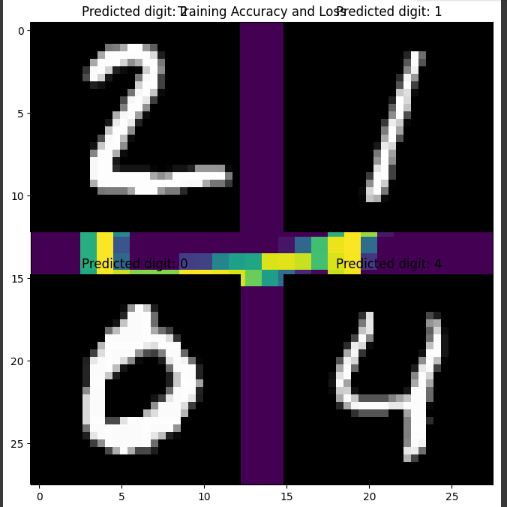
* print("Pixel range:", X\_train.min(), "to", X\_train.max())
* **Insight**: Should be between 0.0 and 1.0 if normalized correctly.

#### 5. **Plot Pixel Intensity Distribution**

* plt.hist(X\_train.ravel(), bins=50)
* plt.title("Distribution of Pixel Intensities")
* plt.xlabel("Pixel Intensity")
* plt.ylabel("Frequency")
* plt.show()

#### 6. **Average Image for Each Digit (Optional Advanced)**

* fig, axes = plt.subplots(2, 5, figsize=(10, 5))
* for i in range(10):
* avg\_img = X\_train[np.argmax(y\_train, axis=1) == i].mean(axis=0).squeeze()
* ax = axes[i // 5, i % 5]
* ax.imshow(avg\_img, cmap='gray')
* ax.set\_title(f"Avg Digit: {i}")
* ax.axis('off')
* plt.suptitle("Average Image of Each Digit")
* plt.show()
* **Insight**: This shows the most common pixel patterns per digit.



### **9. Feature Engineering**

* Since this project uses **image data**, traditional feature engineering (like extracting manual numeric or categorical features) is limited. Instead, we focus on **preparing the data to help the deep learning model learn features automatically** — particularly using CNNs.

#### 1. **Pixel Intensity Normalization**

* Normalize pixel values from the range **[0, 255]** to **[0, 1]** to speed up training and improve convergence.
* X\_train = X\_train / 255.0
* X\_test = X\_test / 255.0

#### 2. **Reshape for CNN Compatibility**

* Reshape the images to match the expected input shape of the CNN (i.e., add a channel dimension).
* python
* CopyEdit
* X\_train = X\_train.reshape(-1, 28, 28, 1)
* X\_test = X\_test.reshape(-1, 28, 28, 1)

#### 3. **One-Hot Encoding of Labels**

* Convert digit labels (e.g., 7) into one-hot vectors like [0,0,0,0,0,0,0,1,0,0] for multi-class classification.
* python
* CopyEdit
* from tensorflow.keras.utils import to\_categorical
* y\_train = to\_categorical(y\_train, num\_classes=10)
* y\_test = to\_categorical(y\_test, num\_classes=10)

#### 4. **Data Augmentation (Optional but Recommended)**

* Introduce small variations in training images (rotation, zoom, shift, etc.) to help the model generalize better.
* from tensorflow.keras.preprocessing.image import ImageDataGenerator
* datagen = ImageDataGenerator(
* rotation\_range=10,
* zoom\_range=0.1,
* width\_shift\_range=0.1,
* height\_shift\_range=0.1
* )
* datagen.fit(X\_train)

#### 5. **Learned Features via CNN (Core of Feature Engineering in Deep Learning)**

* *In deep learning,* ***convolutional layers automatically learn features*** *such as edges, textures, and shapes through filters.*

### **10. Model Building**

### 1. **Convolutional Neural Networks (CNNs)**

* **Description**: CNNs are specifically designed for image data. They use convolutional layers to automatically learn spatial hierarchies in images (edges, textures, shapes) and pooling layers to reduce dimensionality.
* **Why Suitable**:
  + **Image-Specific Architecture**: CNNs are excellent for image recognition because they focus on learning from local patterns, which is crucial for understanding handwritten digits.
  + **Parameter Sharing**: CNNs share weights across the image, allowing them to efficiently learn features at multiple scales.
  + **Use Case**: CNNs are the go-to model for digit recognition tasks like MNIST, and they have proven to perform exceptionally well on these types of datasets.
* **Example Architecture**: A typical CNN for this task might consist of:
  + **Convolutional layers** to detect features such as edges or curves.
  + **MaxPooling layers** to reduce spatial dimensions and avoid overfitting.
  + **Fully connected layers** (dense layers) to classify the digit.

### 2. **Fully Connected Neural Networks (FCNs)**

* **Description**: Fully connected networks consist of multiple layers where each neuron is connected to every neuron in the subsequent layer.
* **Why Suitable**:
  + **Simple and Effective**: Though not as efficient as CNNs for image data, FCNs can still provide reasonable results on the MNIST dataset by learning pixel-level relationships.
  + **Use Case**: Useful as a baseline model for comparing performance with more complex architectures like CNNs.
* **Example Architecture**: A simple FCN could consist of:
  + **Flattening** the image to a vector (from 28x28 to 784 features).
  + **Dense layers** with ReLU activations.
  + **Output layer** with softmax for multi-class classification.

### 3. **LeNet-5 (Legacy CNN Architecture)**

* **Description**: LeNet-5 is one of the first CNN architectures designed for digit recognition, which was used to recognize MNIST digits in the 1990s.
* **Why Suitable**:
  + **Benchmark Model**: Since LeNet-5 was initially designed for digit recognition, it serves as a strong baseline model.
  + **Legacy Approach**: While more modern CNN architectures have surpassed it, LeNet-5 offers a historical foundation and is computationally less expensive, making it great for educational purposes or benchmarking.
* **Architecture**:
  + **Convolutional layers** (for feature extraction).
  + **Average pooling** (downsampling).
  + **Fully connected layers** (for classification).

### 4. **Residual Networks (ResNet)**

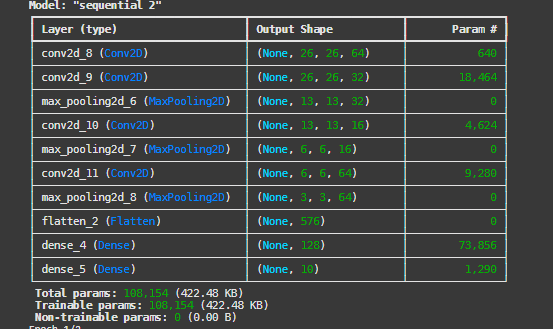
* **Description**: ResNet uses residual connections (skip connections) that allow the model to learn residual functions instead of direct mappings.
* **Why Suitable**:
  + **Deep Learning Efficiency**: ResNet allows for the construction of deeper networks without the issue of vanishing gradients, which can make it suitable for more complex datasets beyond MNIST in the future.
  + **Improved Performance**: ResNet typically achieves higher accuracy by using deeper architectures while avoiding overfitting.
* **Example Use Case**: While ResNet is typically applied to larger datasets (e.g., ImageNet), using a small version like ResNet-18 or ResNet-34 can provide valuable insights and higher accuracy.

### 5. **Data Augmentation and Transfer Learning (Optional)**

* **Description**: Transfer learning involves using a pre-trained model on a large dataset (e.g., ImageNet) and fine-tuning it for the task at hand.
* **Why Suitable**:
  + **Pre-trained Models**: Transfer learning is beneficial when computational resources are limited or if there’s a need to generalize beyond MNIST by using models pre-trained on larger datasets.
  + **Data Augmentation**: Augmenting the MNIST dataset (rotations, shifts, zooms) helps the model generalize better by learning invariant features.

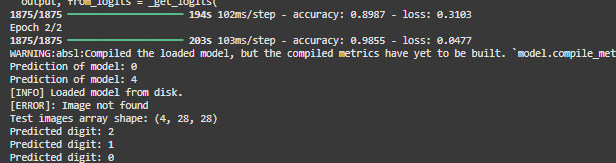
### 6. **K-Nearest Neighbors (KNN) (Simple Baseline Model)**

* **Description**: KNN is a non-parametric classification algorithm that assigns the class of a sample based on the majority class of its nearest neighbors in the feature space.
* **Why Suitable**:
  + **Simple and Intuitive**: KNN can serve as a baseline model to compare performance against more complex models like CNNs.
  + **Non-Linear Classifier**: While not ideal for high-dimensional image data, it can still perform decently with a small dataset or when using a kernel trick.
* **Use Case**: This is a good baseline model to understand the classification boundary of the dataset.

****

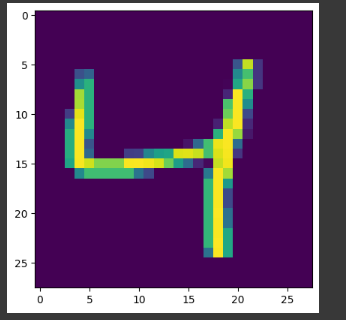
### **11. Model Evaluation**

* **Model Metrics**: For all models, evaluate performance using accuracy, precision, recall, F1-score, and confusion matrix.
* **Computational Cost**: CNNs and ResNets will require more computational resources compared to fully connected networks or simpler models like KNN.
* **Overfitting Risk**: Use techniques like dropout, data augmentation, and regularization to mitigate overfitting in deep learning models.



### **12. Deployment**

* *Deploy using a free platform:*
  + *Gradio + Hugging Face Spaces*
* *Include:*
  + *Public link*



**13. Source code**

*!pip install gradio*

import matplotlib.pyplot as plt

import numpy as np

import os

import PIL

import tensorflow as tf

import cv2

import sys

from tensorflow import keras

from tensorflow.keras import layers, datasets, models

from tensorflow.keras.models import Sequential

"""## Prepare Dataset"""

(train\_images, train\_labels), (test\_images, test\_labels) = datasets.mnist.load\_data()

train\_images = train\_images.reshape((60000, 28, 28, 1))

test\_images = test\_images.reshape((10000, 28, 28, 1))

# Normalize pixel values to be between 0 and 1

train\_images, test\_images = train\_images / 255.0, test\_images / 255.0

print("TRAIN IMAGES: ", train\_images.shape)

print("TEST IMAGES: ", test\_images.shape)

"""## Create Model"""

num\_classes = 10

img\_height = 28

img\_width = 28

model = Sequential([

    layers.Conv2D(64, (3, 3), activation='relu', input\_shape=(28, 28, 1)),

    layers.Conv2D(32, 3, padding='same', activation='relu'),

    layers.MaxPooling2D(),

    layers.Conv2D(16, 3, padding='same', activation='relu'),

    layers.MaxPooling2D(),

    layers.Conv2D(64, 3, padding='same', activation='relu'),

    layers.MaxPooling2D(),

    layers.Flatten(),

    layers.Dense(128, activation='relu'),

    layers.Dense(10, activation='sigmoid')

])

"""## Compile Model"""

model.compile(optimizer='adam',

              loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True),

              metrics=['accuracy'])

model.summary()

"""## Train Model"""

epochs = 2

history = model.fit(

  train\_images,

  train\_labels,

  epochs = epochs

)

"""## Visualize Training Results"""

acc = history.history['accuracy']

loss=history.history['loss']

epochs\_range = range(epochs)

plt.figure(figsize=(8, 8))

plt.plot(epochs\_range, acc, label='Training Accuracy')

plt.plot(epochs\_range, loss, label='Loss')

plt.legend(loc='lower right')

plt.title('Training Accuracy and Loss')

"""## Test Image"""

image = (train\_images[1]).reshape(1,28,28,1)

# Use predict and argmax to get predicted class

model\_pred = np.argmax(model.predict(image, verbose=0), axis=-1)

plt.imshow(image.reshape(28,28))

print('Prediction of model: {}'.format(model\_pred[0]))

image = (train\_images[2]).reshape(1,28,28,1)

# Use predict and argmax to get predicted class

model\_pred = np.argmax(model.predict(image, verbose=0), axis=-1)

plt.imshow(image.reshape(28,28))

print('Prediction of model: {}'.format(model\_pred[0]))

# In the Load Model section:

MODEL\_PATH = "tf-cnn-model.h5"

def predict\_digit(image\_path):

    # load model

    model = models.load\_model(MODEL\_PATH)

    print("[INFO] Loaded model from disk.")

    image = cv2.imread(image\_path, 0)

    image1 = cv2.resize(image, (28,28))    # For cv2.imshow: dimensions should be 28x28

    image2 = image1.reshape(1,28,28,1)

    cv2.imshow('digit', image1 )

    pred = np.argmax(model.predict(image2), axis=-1)

    return pred[0]

def main(image\_path):

    predicted\_digit = predict\_digit(image\_path)

    print('Predicted Digit: {}'.format(predicted\_digit))

if \_\_name\_\_ == "\_\_main\_\_":

    try:

        main(image\_path = sys.argv[1])

    except:

        print('[ERROR]: Image not found')

"""## Test Multiple Image"""

images = test\_images[1:5]

images = images.reshape(images.shape[0], 28, 28)

print ("Test images array shape: {}".format(images.shape))

# Moved prediction inside the loop where test\_image is defined

for i, test\_image in enumerate(images, start=1):

    org\_image = test\_image

    test\_image = test\_image.reshape(1,28,28,1)

    # Now prediction is calculated within the loop

    prediction = np.argmax(model.predict(test\_image, verbose=0), axis=-1)

    print ("Predicted digit: {}".format(prediction[0]))

    plt.subplot(220+i)

    plt.axis('off')

    plt.title("Predicted digit: {}".format(prediction[0]))

    plt.imshow(org\_image, cmap=plt.get\_cmap('gray'))

plt.show()

"""## Save Model"""

model.save("tf-cnn-model.h5")

"""## Load Model"""

loaded\_model = models.load\_model("tf-cnn-model.h5")

image = (train\_images[2]).reshape(1,28,28,1)

#model\_pred = np.argmax(loaded\_model.predict(image, verbose=0), axis=-1)

#model\_pred = loaded\_model.predict\_classes(image, verbose=0)

plt.imshow(image.reshape(28,28))

print('Prediction of model: {}'.format(model\_pred[0]))

#dly

import gradio as gr

import numpy as np

import tensorflow as tf

from tensorflow.keras.models import load\_model

import cv2

# Load your saved model

model = load\_model("tf-cnn-model.h5")

def preprocess(image):

    # Convert to grayscale if RGB

    if image.shape[-1] == 3:

        image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

    # Resize to 28x28

    image = cv2.resize(image, (28, 28))

    # Normalize and reshape for model

    image = image.astype('float32') / 255.0

    image = image.reshape(1, 28, 28, 1)

    return image

def predict\_digit(image):

    processed\_image = preprocess(image)

    prediction = model.predict(processed\_image)

    predicted\_class = np.argmax(prediction, axis=1)[0]

    return str(predicted\_class)

# Gradio interface - input as image (sketchpad or upload)

iface = gr.Interface(

    fn=predict\_digit,

    inputs=gr.Image(height=28, width=28, image\_mode='L', source='canvas'), # Removed invert\_colors

    outputs="text",

    title="MNIST Digit Classifier",

    description="Draw a digit (0-9) and the model will predict it."

)

iface.launch()

**14. Future scope**

***Expansion to Alphanumeric and Multilingual Recognition****:  
Currently, the model is limited to recognizing digits (0–9). A logical enhancement would be extending the model to recognize alphabets (A–Z), special symbols, or characters from other languages (e.g., Devanagari, Chinese). This would enable broader applications in document digitization and multilingual OCR systems.*

* **Integration with Real-Time Applications**:  
  Future developments could include integrating the model with mobile or embedded systems for real-time digit recognition—such as reading handwritten inputs on smart devices, automating form entries, or assisting visually impaired users using camera input.
* **Model Optimization for Edge Devices**:  
  As the current model may require moderate hardware for training and inference, applying techniques like quantization, pruning, or knowledge distillation can help optimize it for deployment on low-power devices such as Raspberry Pi, Arduino with AI support, or mobile phones.
* **Robustness to Noisy and Distorted Input**:  
  Enhancing the model to handle noisy, blurred, or skewed inputs would make it more reliable in practical scenarios. Future work could include advanced data augmentation strategies or training with real-world noisy datasets to improve generalization.

**13. Team Members and Roles**

*[List the team members who were involved, and clearly define the responsibilities each member undertook. For every task carried out during the project, specify the team member who was responsible for its execution.]*