ASSIGNMENT ---5

Aim :Digit Recognition using CNN-Deep learning model.

1.Introduction:

Handwritten digit recognition is a well-established problem in machine learning and computer vision. It involves training a model to correctly identify numerical digits (0-9) from images. This has numerous real-world applications, such as postal code recognition, bank check processing, and optical character recognition (OCR).

Convolutional Neural Networks (CNNs) are particularly well-suited for image-based tasks due to their ability to automatically learn hierarchical features from raw pixel data. Unlike traditional machine learning algorithms that often require manual feature engineering, CNNs can extract relevant patterns (edges, shapes, etc.) through their convolutional and pooling layers, leading to highly accurate results.

This assignment will walk through building, training, and evaluating a CNN model for handwritten digit recognition using the widely available MNIST dataset.

2\. Dataset: MNIST

The MNIST (Modified National Institute of Standards and Technology) dataset is a cornerstone for benchmarking image classification algorithms. It consists of:

* **60,000 training images** of handwritten digits (0-9).
* **10,000 testing images** of handwritten digits (0-9).

Each image is a grayscale 28x28 pixel image. The pixel values range from 0 to 255, where 0 represents black and 255 represents white.

3. Steps to Implement a CNN for Digit Recognition

Here's a detailed breakdown of the steps you'll typically follow:

....Step 1: Import Libraries

You'll need several Python libraries for data manipulation, building the CNN, and visualization.

```
` ` ` python
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
import numpy as np
```

......Step 2: Load and Explore the Dataset

Load the MNIST dataset, which is conveniently available within Keras.

```python
(x_train, y_train), (x_test, y_test) = mnist.load_data()

print(f"Training data shape: {x_train.shape}")
print(f"Training labels shape: {y_train.shape}")
print(f"Testing data shape: {x_test.shape}")
print(f"Testing labels shape: {y_test.shape}")

# Visualize some examples
plt.figure(figsize=(10, 5))
for i in range(10):
    plt.subplot(2, 5, i + 1)
    plt.imshow(x_train[i], cmap='gray')
    plt.title(f"Label: {y_train[i]}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

......Step 3: Preprocess the Data

Data preprocessing is crucial for optimal model performance.

* **Reshape Images:** CNNs expect input images to have a channel dimension. For grayscale images, this is typically 1. So, a 28x28 image becomes 28x28x1.

    ```python
    # Reshape images to add a channel dimension (for grayscale, it's 1)
    x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
    x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
    ```

* **Normalize Pixel Values:** Pixel values range from 0 to 255. Normalizing them to a range of 0 to 1 helps the model converge faster and perform better.

```python
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
```

* **One-Hot Encode Labels:** The labels (0-9) are categorical. For multi-class classification, it's standard practice to convert them to one-hot encoded vectors. For example, the digit '3' would become `[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]`.

```python
num_classes = 10
y_train = to_categorical(y_train, num_classes)
y_test = to_categorical(y_test, num_classes)
```

........Step 4: Build the CNN Model

Define the architecture of your Convolutional Neural Network. A typical CNN for digit recognition includes:

* **Convolutional Layers (`Conv2D`):** These layers apply filters to the input image, extracting features like edges, corners, and textures. You specify the number of filters, kernel size (e.g., 3x3), and activation function (ReLU is common).
* **Pooling Layers (`MaxPooling2D`):** These layers reduce the spatial dimensions of the feature maps, making the model more robust to small shifts and distortions in the input. `MaxPooling2D` takes the maximum value within a specified window (e.g., 2x2).
* **Flatten Layer:** Converts the 2D feature maps from the convolutional and pooling layers into a 1D vector, preparing them for the fully connected layers.
* **Dense (Fully Connected) Layers:** These are standard neural network layers that take the flattened features and perform classification.
* **Dropout Layer:** A regularization technique that randomly sets a fraction of input units to zero at each update during training. This helps prevent overfitting.
* **Output Layer:** A `Dense` layer with `num_classes` (10 for MNIST) neurons and a `softmax` activation function. Softmax outputs a probability distribution over the classes.

<!-- end list -->

```python
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5), # Add dropout for regularization
    Dense(num_classes, activation='softmax')
])

model.summary() # Print a summary of the model's architecture
```

.......Step 5: Compile the Model

Before training, you need to compile the model by specifying:

  * **Optimizer:** An algorithm to adjust the weights during training (e.g., 'adam', 'rmsprop', 'sgd'). Adam is a good general-purpose optimizer.
  * **Loss Function:** A function that quantifies the difference between the model's predictions and the true labels. For multi-class classification with one-hot encoded labels, `categorical_crossentropy` is appropriate.
  * **Metrics:** Metrics to evaluate the model's performance (e.g., 'accuracy').

<!-- end list -->

```python
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

........Step 6: Train the Model

Train the model using your training data.

  * **Epochs:** The number of times the model will iterate over the entire training dataset.
  * **Batch Size:** The number of samples per gradient update.

<!-- end list -->

```python
history = model.fit(x_train, y_train,
                    epochs=10, # You can adjust the number of epochs
                    batch_size=32,
                    validation_split=0.1) # Use 10% of training data for validation
```

........Step 7: Evaluate the Model

Evaluate the trained model on the unseen test data to assess its generalization ability.

```python
loss, accuracy = model.evaluate(x_test, y_test, verbose=0)
print(f"Test Loss: {loss:.4f}")
print(f"Test Accuracy: {accuracy:.4f}")

# Plot training history (accuracy and loss)
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.show()
```

.........Step 8: Make Predictions

Use the trained model to make predictions on new, unseen data (or a portion of your test set).

```python
predictions = model.predict(x_test[:10]) # Predict on the first 10 test images
predicted_classes = np.argmax(predictions, axis=1)
true_classes = np.argmax(y_test[:10], axis=1)

print("\nSample Predictions:")
for i in range(10):
    print(f"True: {true_classes[i]}, Predicted: {predicted_classes[i]}")

# Visualize some predictions
plt.figure(figsize=(12, 6))
for i in range(10):
    plt.subplot(2, 5, i + 1)
    plt.imshow(x_test[i].reshape(28, 28), cmap='gray')
    plt.title(f"True: {true_classes[i]}\nPred: {predicted_classes[i]}",
              color='green' if true_classes[i] == predicted_classes[i] else 'red')
    plt.axis('off')
plt.tight_layout()
plt.show()
```

.........Step 9: Save the Model (Optional but Recommended)

Once you're satisfied with your model's performance, save it for future use.

```python
model.save('digit_recognition_cnn_model.h5')
print("Model saved as digit_recognition_cnn_model.h5")

# To load the model later:
# from tensorflow.keras.models import load_model
# loaded_model = load_model('digit_recognition_cnn_model.h5').
```

4.Create GUI to predict digits:

Now for the GUI, we have created a new file in which we build an interactive window to draw digits on canvas and with a button, we can recognize the digit. The Tkinter library comes in the Python standard library. We have created a function predict_digit() that takes the image as input and then uses the trained model to predict the digit.

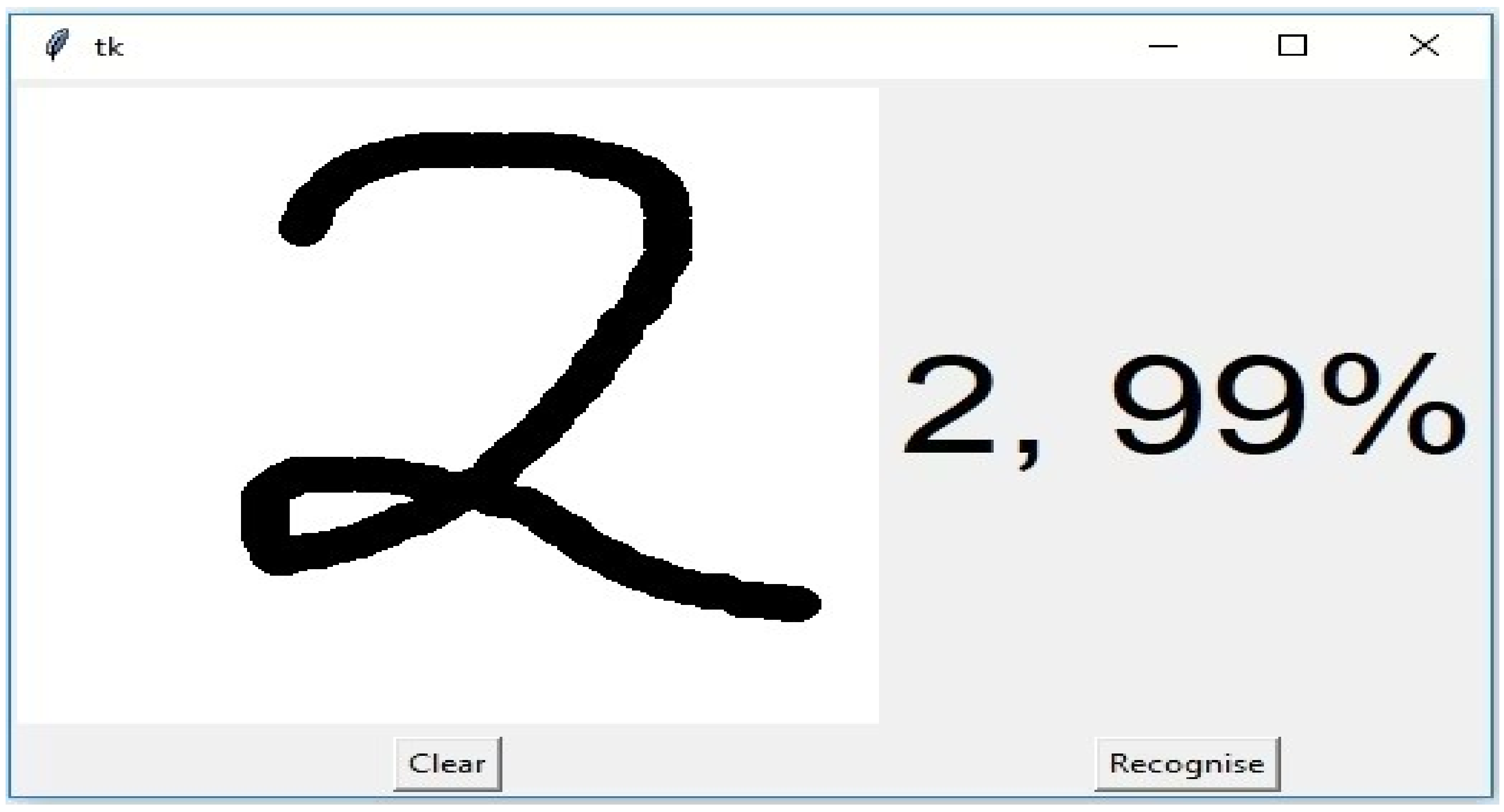
Then we create the App class which is responsible for building the GUI for our app. We create a canvas where we can draw by capturing the mouse event and with a button, we trigger the predict_digit() function and display the results.

Here’s the full code for our gui_digit_recognizer.py file:

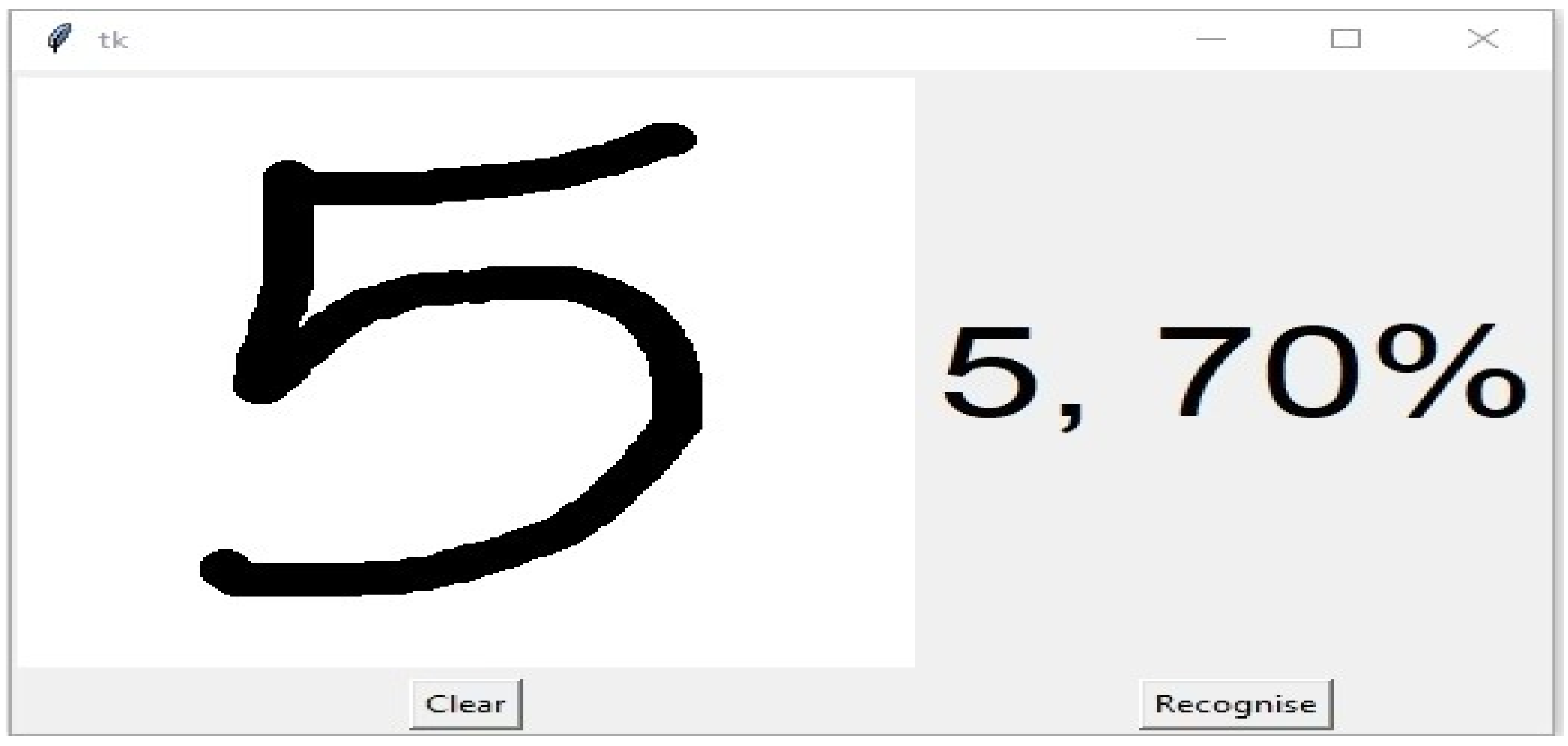
```
from keras.models import load_model
from tkinter import *
import tkinter as tk
import win32gui
from PIL import ImageGrab, Image
import numpy as np

model = load_model('mnist.h5')
```

```
def predict_digit(img):
    #resize image to 28x28 pixels
    img = img.resize((28,28))
    #convert rgb to grayscale
    img = img.convert('L')
    img = np.array(img)
    #reshaping to support our model input and normalizing
    img = img.reshape(1,28,28,1)
    img = img/255.0
    #predicting the class
    res = model.predict([img])[0]
    return np.argmax(res), max(res)

class App(tk.Tk):
    def __init__(self):
        tk.Tk.__init__(self)

        self.x = self.y = 0

        # Creating elements
        self.canvas = tk.Canvas(self, width=300, height=300, bg = "white", cursor="cross")
        self.label = tk.Label(self, text="Thinking..", font=("Helvetica", 48))
        self.classify_btn = tk.Button(self, text = "Recognise", command =
self.classify_handwriting)
        self.button_clear = tk.Button(self, text = "Clear", command = self.clear_all)

        # Grid structure
        self.canvas.grid(row=0, column=0, pady=2, sticky=W, )
        self.label.grid(row=0, column=1,pady=2, padx=2)
        self.classify_btn.grid(row=1, column=1, pady=2, padx=2)
        self.button_clear.grid(row=1, column=0, pady=2)

        #self.canvas.bind("<Motion>", self.start_pos)
        self.canvas.bind("<B1-Motion>", self.draw_lines)

    def clear_all(self):
        self.canvas.delete("all")

    def classify_handwriting(self):
        HWND = self.canvas.winfo_id() # get the handle of the canvas
        rect = win32gui.GetWindowRect(HWND) # get the coordinate of the canvas
        im = ImageGrab.grab(rect)

        digit, acc = predict_digit(im)
```

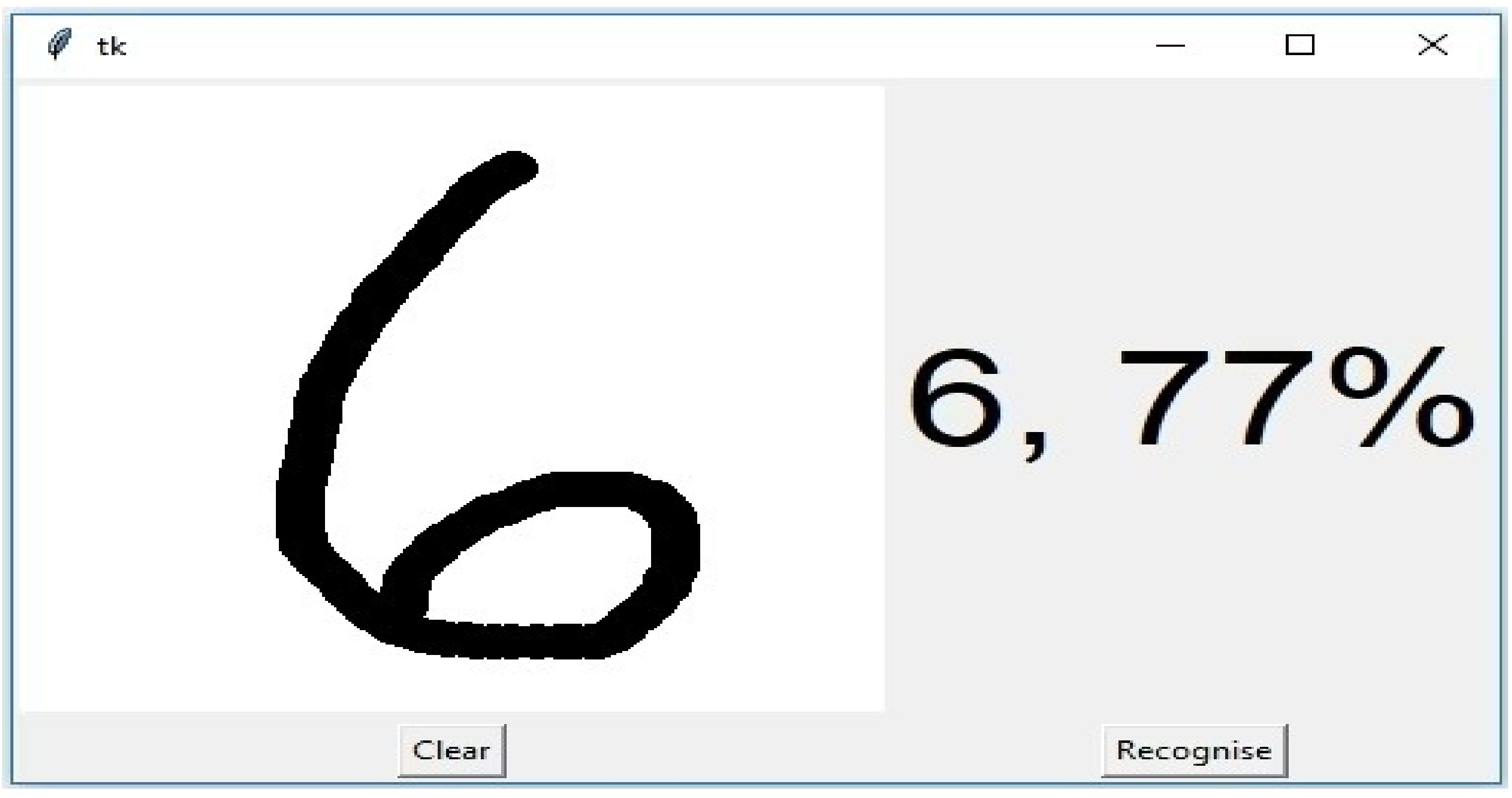
```
            self.label.configure(text= str(digit)+', '+ str(int(acc*100))+'%')

        def draw_lines(self, event):
            self.x = event.x
            self.y = event.y
            r=8
            self.canvas.create_oval(self.x-r, self.y-r, self.x + r, self.y + r, fill='black')

app = App()
mainloop()
```



2, 99%



5, 70%



6, 77%

Summary:

In this article, we have successfully built a Python deep learning project on handwritten digit recognition app. We have built and trained the Convolutional neural network which is very effective for image classification purposes. Later on, we build the GUI where we draw a digit on the canvas then we classify the digit and show the results.