Deployment on Flask:

Digit Recognition Using the MNIST dataset

Name: Zohra Bouchamaoui
Batch Code: LISUM21
Submission date: 27/05/2023
Submitted to: Data Glacier

1. Introduction:

In this project, our objective is to deploy a machine learning model for digit recognition using the MNIST dataset. We will leverage the Flask Framework to create a web application that allows users to submit handwritten digit images and receive predictions from our trained model.

Our focus will be twofold: firstly, building a machine learning model capable of accurately recognizing handwritten digits, and secondly, integrating this model into a web application using Flask, a lightweight and flexible Python micro-framework for web development. Through this project, we will demonstrate the potential of Flask in deploying machine learning models and creating user-friendly interfaces for real-time predictions.

The MNIST dataset is widely recognized as a benchmark in the field of machine learning. It consists of a large collection of 60,000 training images and 10,000 test images, each representing a handwritten digit from 0 to 9. Our goal is to train a model that can accurately classify these digits based on the input images.

By leveraging Flask's simplicity and versatility, we will develop a web application that allows users to upload an image of a handwritten digit. The application will pre-process the image, feed it into our trained model, and provide the predicted digit as the output. This interactive interface will enable users to easily test the performance of our digit recognition model using their own input.

Throughout the project, we will explore the steps involved in data pre-processing, model training, and model evaluation using the MNIST dataset. We will then demonstrate how to transform our trained model into a functional web application using Flask. By the end of the project, we will have a deployable web application that can somewhat accurately recognize handwritten digits in real-time.

Now, let's delve into the details of the project, including the dataset, model architecture, and the process of turning our model into a functional Flask-based web application.

2. Data:

2.1. Overview

The toy dataset we are using for this project is the MNIST dataset (Fig. 2), which is widely recognized as a benchmark in the field of image classification. It consists of a large collection of grayscale images, each representing a handwritten digit from 0 to 9. The dataset is split into two main parts: a training set and a test set.

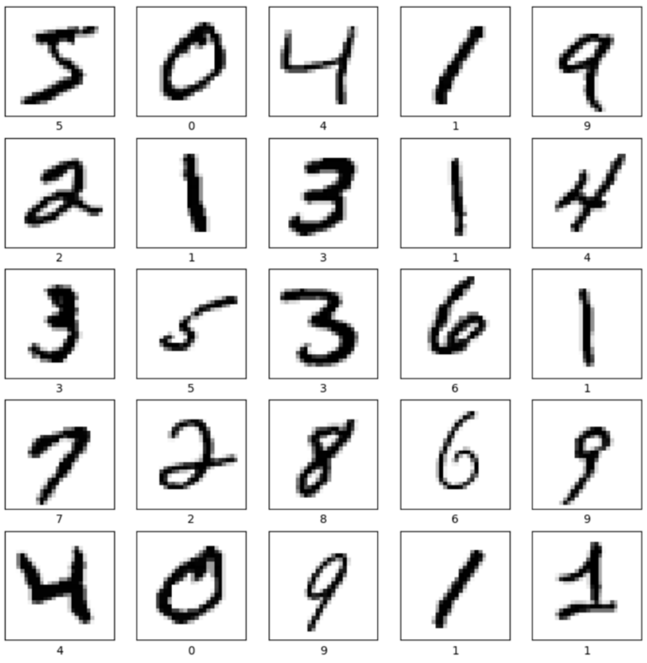


Fig. 2: MNIST dataset

The training set contains 60,000 images, while the test set contains 10,000 images. This division allows us to train our model on a large amount of labelled data and evaluate its performance on unseen examples. The MNIST dataset has become a standard dataset for evaluating the performance of machine learning models in the task of digit recognition.

Each image in the MNIST dataset has a resolution of 28x28 pixels, resulting in a total of 784 pixels per image. Each pixel represents the intensity of the grayscale, ranging from 0 (black) to 255 (white). The goal of our model is to learn patterns and features from these pixel values to accurately classify the digits.

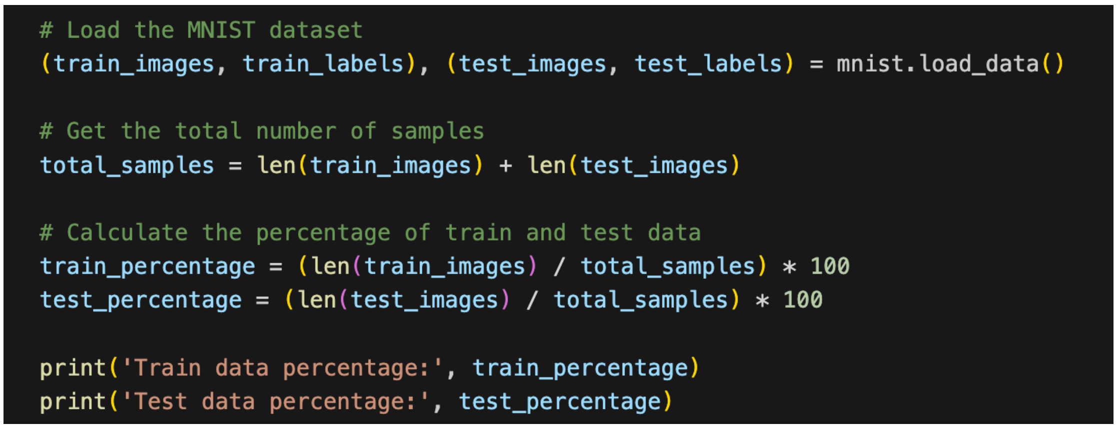
In addition to the image data, the MNIST dataset also provides corresponding labels for each image, indicating the true digit represented by the image. These labels range from 0 to 9 and serve as the ground truth for evaluating the accuracy of our model's predictions.

By leveraging the MNIST dataset, we have a diverse and representative collection of handwritten digits to train and test our model. This dataset has been extensively studied and serves as a reliable benchmark for evaluating the performance of various machine learning algorithms in the domain of digit recognition.

In the next section, we will delve into the process of building our machine learning model using the MNIST dataset, exploring the architecture, training procedure, and evaluation metrics.

2.2. Data Pre-processing

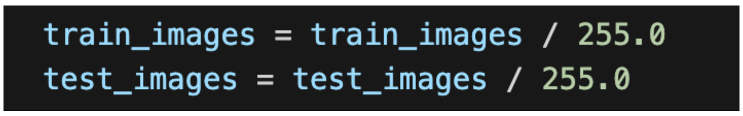
The dataset used here is split into 85% for the training set and the remaining 15% for the test set (Fig. 3).



```python
# Load the MNIST dataset
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Get the total number of samples
total_samples = len(train_images) + len(test_images)

# Calculate the percentage of train and test data
train_percentage = (len(train_images) / total_samples) * 100
test_percentage = (len(test_images) / total_samples) * 100

print('Train data percentage:', train_percentage)
print('Test data percentage:', test_percentage)
```

Fig. 3: Load the dataset.

The pixel values of the images (between 0 and 255) are then normalised (Fig. 4) to be between 0 and 1.



```
train_images = train_images / 255.0
test_images = test_images / 255.0
```

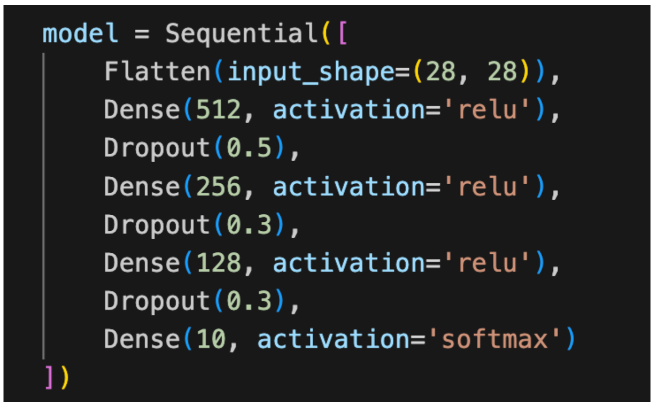
Fig. 4: Normalise the pixel values.

3. Building a Model:

In this section, we will focus on building a machine learning model for digit recognition using the MNIST dataset. Our goal is to develop a model that can accurately classify handwritten digits from 0 to 9.

3.1. Model Architecture:

For this task, we will utilize a convolutional neural network (CNN) architecture, which is known for its effectiveness in image classification tasks. The CNN architecture is well-suited for capturing spatial patterns and features in images, making it a suitable choice for digit recognition.



```
model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(512, activation='relu'),
    Dropout(0.5),
    Dense(256, activation='relu'),
    Dropout(0.3),
    Dense(128, activation='relu'),
    Dropout(0.3),
    Dense(10, activation='softmax')
])
```

Fig. 5: Model architecture

Our model architecture (Fig. 5) consists of several layers:

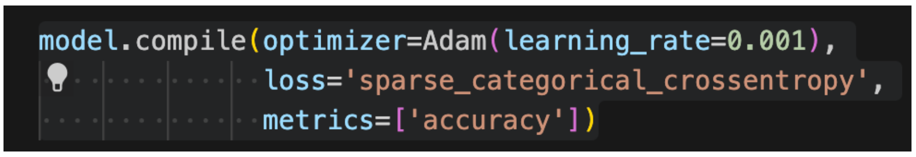
- The first layer is a convolutional layer that applies a set of filters to the input image. These filters help extract relevant features from the image.

- Next, we apply a pooling layer to reduce the spatial dimensions of the features and retain the most important information.
- We then flatten the pooled features and pass them through fully connected (dense) layers to learn the relationships between the extracted features.
- Finally, we have an output layer with 10 nodes, each representing a digit from 0 to 9. We use the softmax activation function to obtain the probability distribution over the classes.

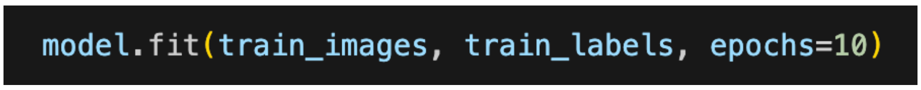
By combining convolutional, and dense layers, our model can effectively learn discriminative features and make predictions for each digit class.

3.2. Training Procedure:

To train our model (Fig. 6), we used the training set of the MNIST dataset, which consists of 60,000 labelled images. We employed the categorical cross-entropy loss function, which is suitable for multi-class classification tasks, and the Adam optimizer to optimize the model parameters.



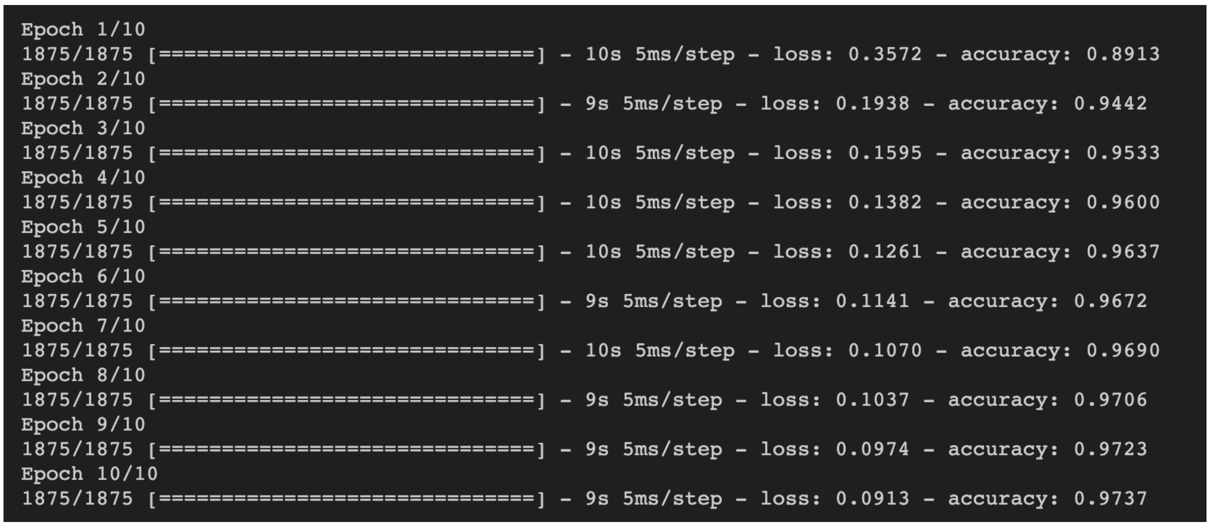
```
model.compile(optimizer=Adam(learning_rate=0.001),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```



```
model.fit(train_images, train_labels, epochs=10)
```

Fig. 6: Model training

During training, we iterate over the training set in multiple epochs (Fig. 7), where each epoch represents a complete pass through the entire dataset. The model learns from the training examples and adjusts its parameters to minimize the loss and improve its accuracy.



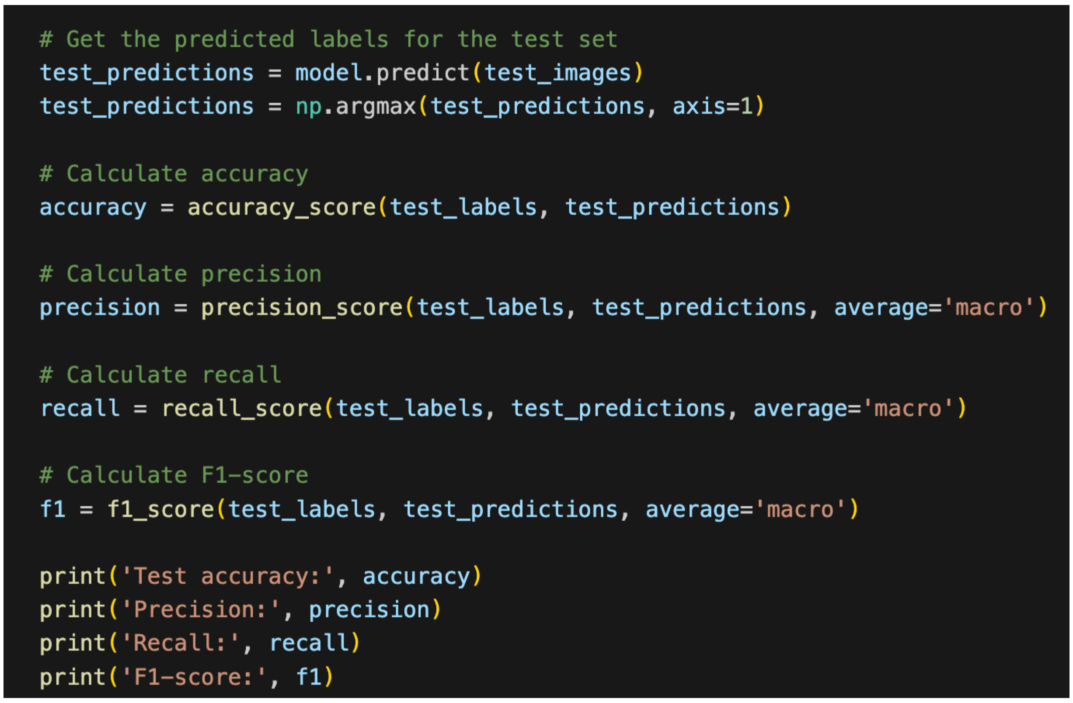
```
Epoch 1/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.3572 - accuracy: 0.8913
Epoch 2/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.1938 - accuracy: 0.9442
Epoch 3/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.1595 - accuracy: 0.9533
Epoch 4/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.1382 - accuracy: 0.9600
Epoch 5/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.1261 - accuracy: 0.9637
Epoch 6/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.1141 - accuracy: 0.9672
Epoch 7/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.1070 - accuracy: 0.9690
Epoch 8/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.1037 - accuracy: 0.9706
Epoch 9/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.0974 - accuracy: 0.9723
Epoch 10/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.0913 - accuracy: 0.9737
```

Fig. 7: Iteration over 10 epochs

We also incorporate techniques like batch normalization and dropout to enhance the generalization ability of our model and prevent overfitting. Batch normalization normalizes the intermediate activations within the network, while dropout randomly deactivates neurons during training to reduce reliance on specific features.

3.3. Evaluation Metrics:

To assess the performance of our model, we utilize evaluation metrics such as accuracy, precision, recall, and F1-score (Fig. 8). Accuracy measures the overall correctness of the model's predictions, while precision quantifies the model's ability to correctly classify positive instances. Recall measures the model's ability to identify all positive instances, and the F1-score is the harmonic mean of precision and recall.



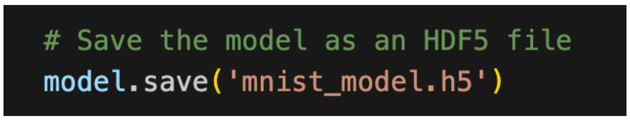
```python
# Get the predicted labels for the test set
test_predictions = model.predict(test_images)
test_predictions = np.argmax(test_predictions, axis=1)

# Calculate accuracy
accuracy = accuracy_score(test_labels, test_predictions)

# Calculate precision
precision = precision_score(test_labels, test_predictions, average='macro')

# Calculate recall
recall = recall_score(test_labels, test_predictions, average='macro')

# Calculate F1-score
f1 = f1_score(test_labels, test_predictions, average='macro')

print('Test accuracy:', accuracy)
print('Precision:', precision)
print('Recall:', recall)
print('F1-score:', f1)
```

Fig. 8: Evaluation metrics

In the next section, we will explore the process of deploying our trained model using the Flask framework, allowing us to create a web application for digit recognition.

4. Save the model

To save the trained model we used the 'save()' method provided by the Keras API (Fig. 9).



```python
# Save the model as an HDF5 file
model.save('mnist_model.h5')
```

Fig. 9: Save the model.

5. Flask Deployment

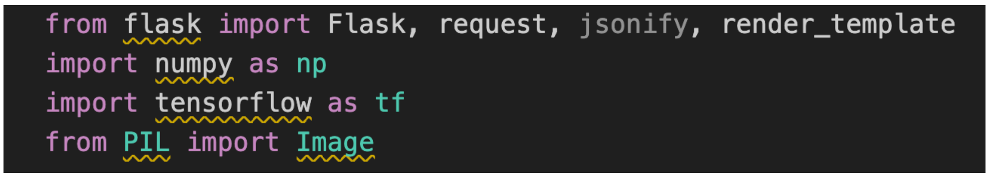
To turn our machine learning model into a web application (Fig. 10), we will utilize the Flask framework, which provides a simple and efficient way to build web applications in Python. By integrating our model with Flask, we can create an interactive web interface for users to input data and receive predictions. Below are the steps involved in turning our model into a web application using Flask:

Week 4
images examples/
    1-black.png
    5-red.png
    9-gold.png
    MNIST_0.png
    sample_image.png
templates/
    index.html
    prediction.html
uploads/
    file.jpg
app.py
DigitRecognMNIST.ipynb
requirements.txt

Fig. 10: Folder directory

- **Install Flask:** we started by installing Flask by running command '*pip install flask*'.

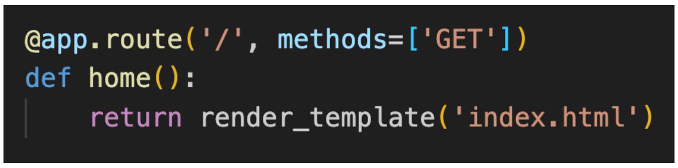
- **Create a Flask Application:** we imported the necessary modules from the Flask library and create an instance of the Flask application (Fig. 11).



```python
from flask import Flask, request, jsonify, render_template
import numpy as np
import tensorflow as tf
from PIL import Image
```

Fig. 11: Importing modules.

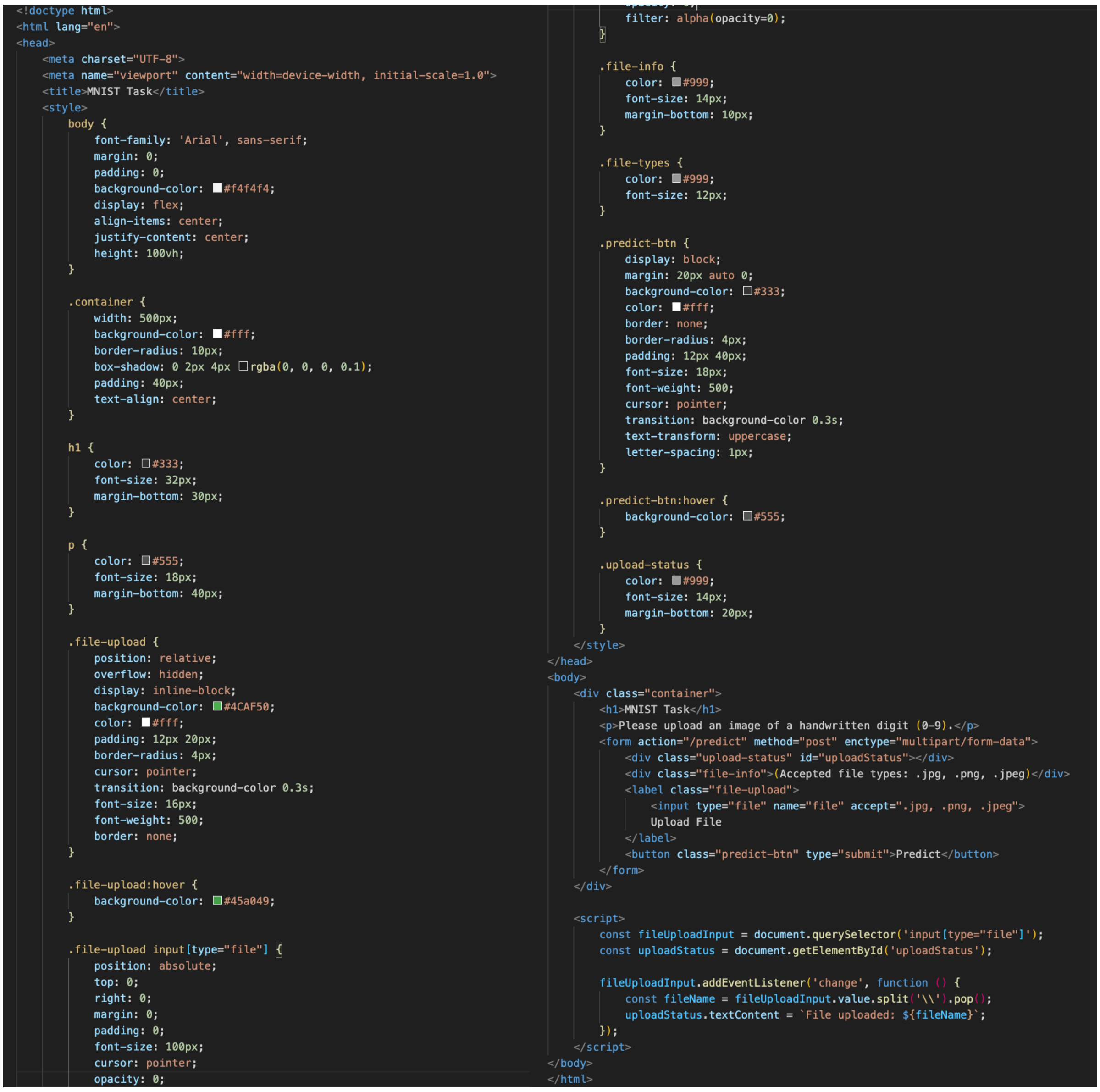
- **Define Routes:** Routes define the different pages and functionalities of our web application. We defined routes using the @app.route() decorator (Fig. 12). For instance, we defined a route to our home page:



```python
@app.route('/', methods=['GET'])
def home():
    return render_template('index.html')
```

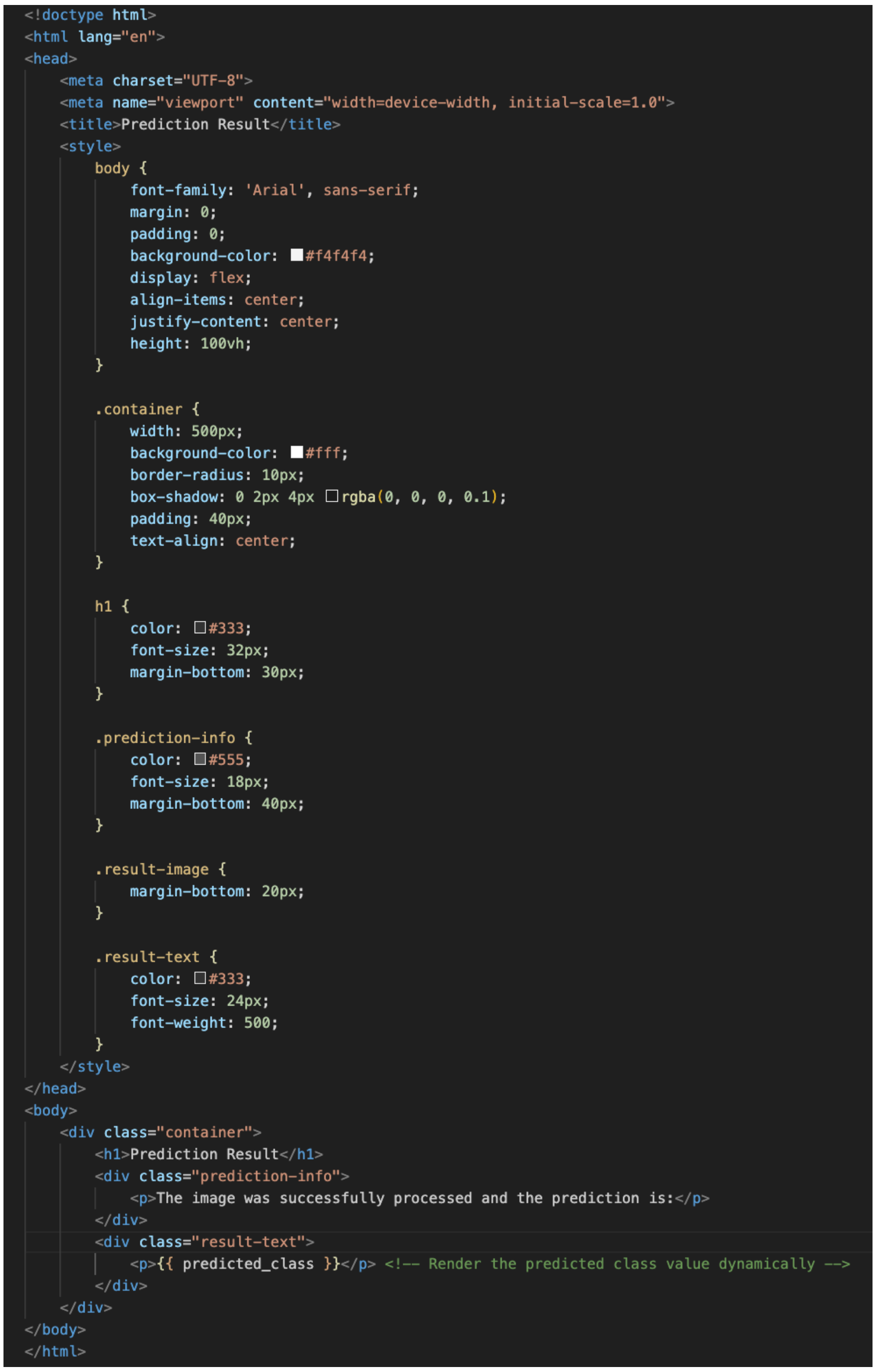
Fig. 12: Defining routes on Flask.

- **Create HTML Templates:** HTML templates define the structure and layout of our web pages. We can use the Jinja2 templating engine to include dynamic content and variables in our templates. We created HTML templates for each page of our web application, such as the home (Fig. 13) and result (Fig. 14) pages.



```html
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>MNIST Task</title>
    <style>
        body {
            font-family: 'Arial', sans-serif;
            margin: 0;
            padding: 0;
            background-color: #f4f4f4;
            display: flex;
            align-items: center;
            justify-content: center;
            height: 100vh;
        }

        .container {
            width: 500px;
            background-color: #fff;
            border-radius: 10px;
            box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
            padding: 40px;
            text-align: center;
        }

        h1 {
            color: #333;
            font-size: 32px;
            margin-bottom: 30px;
        }

        p {
            color: #555;
            font-size: 18px;
            margin-bottom: 40px;
        }

        .file-upload {
            position: relative;
            overflow: hidden;
            display: inline-block;
            background-color: #4CAF50;
            color: #fff;
            padding: 12px 20px;
            border-radius: 4px;
            cursor: pointer;
            transition: background-color 0.3s;
            font-size: 16px;
            font-weight: 500;
            border: none;
        }

        .file-upload:hover {
            background-color: #45a049;
        }

        .file-upload input[type="file"] {
            position: absolute;
            top: 0;
            right: 0;
            margin: 0;
            padding: 0;
            font-size: 100px;
            cursor: pointer;
            opacity: 0;
            filter: alpha(opacity=0);
        }

        .file-info {
            color: #999;
            font-size: 14px;
            margin-bottom: 10px;
        }

        .file-types {
            color: #999;
            font-size: 12px;
        }

        .predict-btn {
            display: block;
            margin: 20px auto 0;
            background-color: #333;
            color: #fff;
            border: none;
            border-radius: 4px;
            padding: 12px 40px;
            font-size: 18px;
            font-weight: 500;
            cursor: pointer;
            transition: background-color 0.3s;
            text-transform: uppercase;
            letter-spacing: 1px;
        }

        .predict-btn:hover {
            background-color: #555;
        }

        .upload-status {
            color: #999;
            font-size: 14px;
            margin-bottom: 20px;
        }
    </style>
</head>
<body>
    <div class="container">
        <h1>MNIST Task</h1>
        <p>Please upload an image of a handwritten digit (0-9).</p>
        <form action="/predict" method="post" enctype="multipart/form-data">
            <div class="upload-status" id="uploadStatus"></div>
            <div class="file-info">(Accepted file types: .jpg, .png, .jpeg)</div>
            <label class="file-upload">
                <input type="file" name="file" accept=".jpg, .png, .jpeg">
                Upload File
            </label>
            <button class="predict-btn" type="submit">Predict</button>
        </form>
    </div>

    <script>
        const fileUploadInput = document.querySelector('input[type="file"]');
        const uploadStatus = document.getElementById('uploadStatus');

        fileUploadInput.addEventListener('change', function () {
            const fileName = fileUploadInput.value.split('\\').pop();
            uploadStatus.textContent = `File uploaded: ${fileName}`;
        });
    </script>
</body>
</html>
```

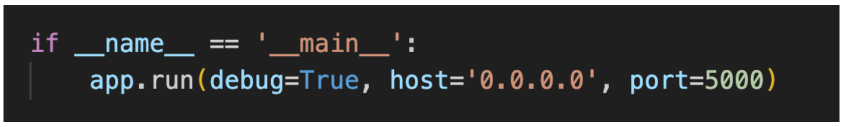
Fig. 13: HTML template for home page (index.html)



```html
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Prediction Result</title>
    <style>
        body {
            font-family: 'Arial', sans-serif;
            margin: 0;
            padding: 0;
            background-color: #f4f4f4;
            display: flex;
            align-items: center;
            justify-content: center;
            height: 100vh;
        }

        .container {
            width: 500px;
            background-color: #fff;
            border-radius: 10px;
            box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
            padding: 40px;
            text-align: center;
        }

        h1 {
            color: #333;
            font-size: 32px;
            margin-bottom: 30px;
        }

        .prediction-info {
            color: #555;
            font-size: 18px;
            margin-bottom: 40px;
        }

        .result-image {
            margin-bottom: 20px;
        }

        .result-text {
            color: #333;
            font-size: 24px;
            font-weight: 500;
        }
    </style>
</head>
<body>
    <div class="container">
        <h1>Prediction Result</h1>
        <div class="prediction-info">
            <p>The image was successfully processed and the prediction is:</p>
        </div>
        <div class="result-text">
            <p>{{ predicted_class }}</p> <!-- Render the predicted class value dynamically -->
        </div>
    </div>
</body>
</html>
```

Fig. 14: HTML template for results page (prediction.html)

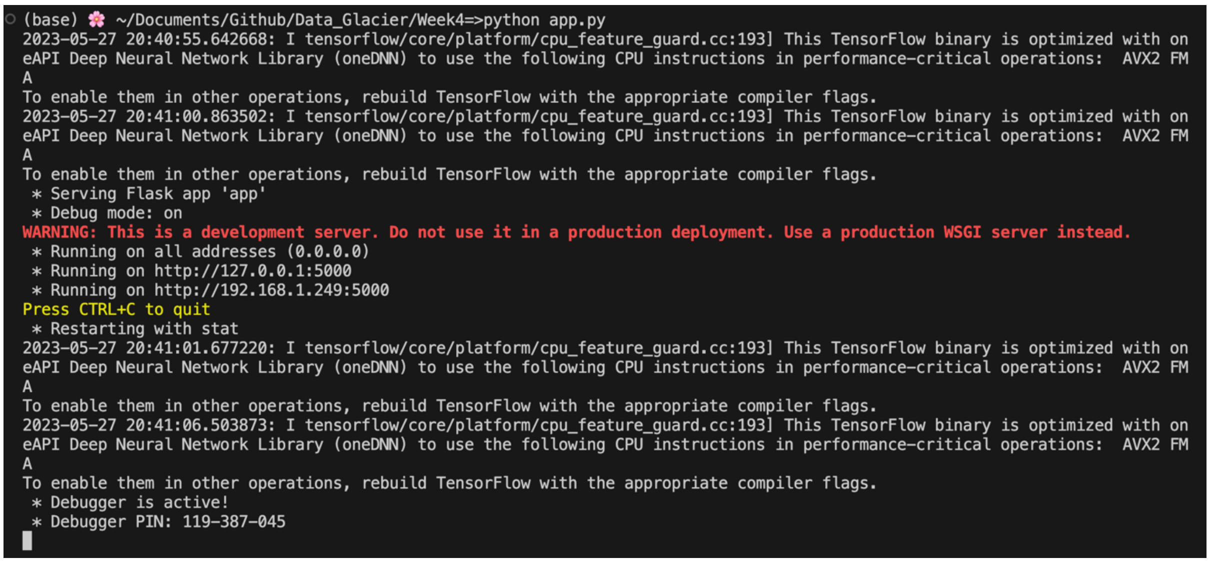
- **Run the Application:** Finally, we can run our Flask application and serve it on a local server using the app.run() method (Fig. 15).



```python
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=5000)
```

Fig. 15: Method to run Flask Application

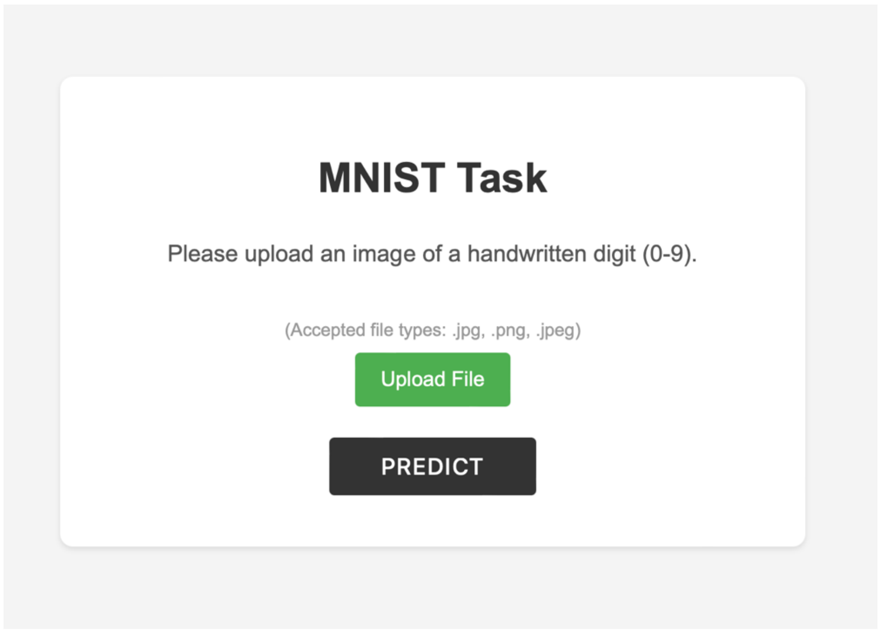
Once we have completed all the above steps, we can run the API by executing the following command on the Terminal, '*python app.py'* (Fig. 16):



```
○ (base) 🌸 ~/Documents/Github/Data_Glacier/Week4=>python app.py
2023-05-27 20:40:55.642668: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with on
eAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:  AVX2 FM
A
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2023-05-27 20:41:00.863502: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with on
eAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:  AVX2 FM
A
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://192.168.1.249:5000
Press CTRL+C to quit
 * Restarting with stat
2023-05-27 20:41:01.677220: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with on
eAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:  AVX2 FM
A
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2023-05-27 20:41:06.503873: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with on
eAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:  AVX2 FM
A
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
 * Debugger is active!
 * Debugger PIN: 119-387-045
```

Fig. 16: Executing the API on the terminal.

If we open the web browser and copy this link, http://192.168.1.249:5000 we should be able to see the following home (Fig. 17) and results pages (Fig. 18):

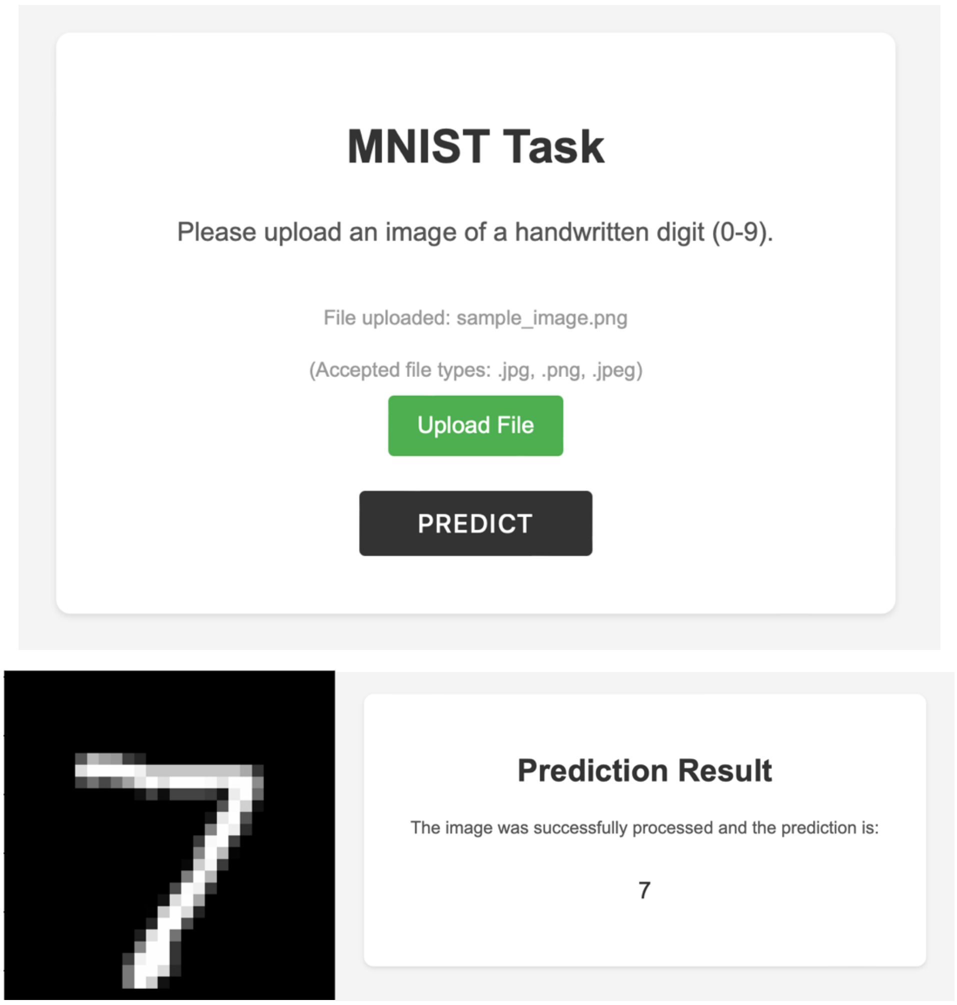


MNIST Task

Please upload an image of a handwritten digit (0-9).

(Accepted file types: .jpg, .png, .jpeg)

Fig. 17: Flask Web Home page


MNIST Task

Please upload an image of a handwritten digit (0-9).

File uploaded: sample_image.png

(Accepted file types: .jpg, .png, .jpeg)

Prediction Result

The image was successfully processed and the prediction is:

7

Fig. 18: Running the task by uploading a digit photo and getting the prediction result.