Date: 02/04/2024

Topics: Flask and Fast API

**Flask API:**

Flask is a micro web framework for Python that allows developers to build web applications quickly and with minimal boilerplate code. It is lightweight, flexible, and easy to use, making it a popular choice for creating APIs (Application Programming Interfaces) in Python.

Flask provides the tools needed to handle HTTP requests and responses, routing, session management, and more, while still giving developers the freedom to structure their applications as they see fit. It follows the WSGI (Web Server Gateway Interface) specification, making it compatible with a wide range of web servers and deployment options.

When building APIs with Flask, developers typically define routes that map HTTP methods and URLs to Python functions, known as view functions. These view functions generate HTTP responses based on the incoming requests, allowing developers to define custom logic for handling API endpoints.

Overall, Flask provides a solid foundation for building APIs in Python, offering a balance of simplicity, flexibility, and performance that makes it well-suited for a wide range of web development tasks.

WSGI – Web Sever Gateway Interface, its helps to communicate between the server and web applications.

Jinja 2 – Web templating system

In VS Code:

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/')

def welcome():

return 'Welcome to the AI course!'

@app.route('/helo')

def helo():

return 'hello world'

if \_\_name\_\_ == '\_\_main\_\_' :

app.run(debug= True)

app = Flask(\_\_name\_\_) – Create instance of the flask class

@app.route(‘/’) – decorator (use to open a web page)

@app.route(‘/’)

@app.route(‘/home’) – it represents the name came after the local host

app.run(debug=True) – no need to refresh the page every time.

**FlaskAPI in Image processing: file name image\_flask**

from transformers import AutoFeatureExtractor, ViTForImageClassification

from PIL import Image

import io

import requests

import torch

from flask import Flask, request, jsonify

# Define the model and feature extractor (place outside any function)

model\_name = "facebook/deit-tiny-patch16-224"

feature\_extractor = AutoFeatureExtractor.from\_pretrained(model\_name)

model = ViTForImageClassification.from\_pretrained(model\_name)

app = Flask(\_\_name\_\_)

@app.route('/', methods=['POST'])

def predict():

# Check if an image file is present in the request

#if 'image' not in request.files:

#return jsonify({'error': 'No image file found in request'}), 400

response = request.get\_json()

image = Image.open(requests.get(response["url"], stream=True).raw)

# Preprocess the image

inputs = feature\_extractor(images=image, return\_tensors="pt")

# Make prediction

# with torch.no\_grad(): # Disable gradient calculation for efficiency

outputs = model(\*\*inputs)

logits = outputs.logits

predicted\_class\_idx = logits.argmax(-1).item()

result = {'result': model.config.id2label[predicted\_class\_idx], 'message': 'Success'}

return result

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True) # Run the Flask app in debug mode

**Fast API:**

FastAPI is a modern, fast (high-performance) web framework for building APIs with Python 3.7+ based on standard Python type hints. It was created to address the need for a framework that combines high performance with ease of use and developer productivity. FastAPI leverages Python's type system to provide automatic data validation, serialization, and documentation, making it a powerful tool for building robust and well-documented APIs.

Key features of FastAPI include:

**Fast:** FastAPI is built on top of Starlette and Pydantic, which are two asynchronous-friendly libraries known for their performance. This allows FastAPI to handle high loads and deliver responses with low latency.

**Type Safety:** FastAPI uses Python type hints to automatically validate request data and generate documentation. By annotating function parameters and return types with Python types, FastAPI can automatically validate incoming requests and convert data to the expected types, reducing the likelihood of runtime errors.

**Automatic Documentation:** FastAPI generates interactive API documentation automatically based on the Python type hints used in your code. This documentation is generated using OpenAPI (formerly Swagger), making it easy to understand and explore API endpoints without the need for manual documentation.

**Asynchronous Support:** FastAPI fully supports asynchronous programming, allowing you to write asynchronous code using Python's async and await syntax. This enables developers to build highly scalable and efficient applications that can handle concurrent requests with ease.

**Dependency Injection:** FastAPI provides a built-in dependency injection system that allows you to easily manage dependencies and share common resources across different parts of your application. This makes it easy to modularize your code and keep it organized.

First install the required packages:

Pip install fastapi

Pip install uvicorn

In VS Code:

from fastapi import FastAPI

app = FastAPI()

@app.get("/")

def root():

return {"message": "Hello World"}

For run this code: uvicorn main:app –reload (write this code in terminal)

**FastAPI in Image Processing: file name: image\_fast.py**

from transformers import AutoFeatureExtractor, ViTForImageClassification

from PIL import Image

import io

import requests

import torch

from fastapi import FastAPI, Request

# Define the model and feature extractor (place outside any function)

model\_name = "facebook/deit-tiny-patch16-224"

feature\_extractor = AutoFeatureExtractor.from\_pretrained(model\_name)

model = ViTForImageClassification.from\_pretrained(model\_name)

app = FastAPI()

@app.post('/', status\_code= 200)

async def predict(info: Request):

# Check if an image file is present in the request

#if 'image' not in request.files:

#return jsonify({'error': 'No image file found in request'}), 400

#response = Request.get\_json()

response = await info.json()

print(response)

image = Image.open(requests.get(response["url"], stream=True).raw)

# Preprocess the image

inputs = feature\_extractor(images=image, return\_tensors="pt")

# Make prediction

# with torch.no\_grad(): # Disable gradient calculation for efficiency

outputs = model(\*\*inputs)

logits = outputs.logits

predicted\_class\_idx = logits.argmax(-1).item()

result = {'result': model.config.id2label[predicted\_class\_idx], 'message': 'Success'}

return result

To run this: uvicorn image\_fast:app –reload

Async def fun() 🡪 In Python, the async def syntax is used to define an asynchronous function. An asynchronous function is a function that can perform non-blocking I/O operations, allowing it to execute concurrently with other functions and tasks without blocking the event loop.

Await: In Python, await is used with asynchronous functions to pause the execution of the current coroutine (async function) until the awaited asynchronous operation completes.

When you use await, the coroutine will wait for the result of the asynchronous operation, allowing other coroutines to run concurrently in the meantime. Without await, the coroutine would continue executing immediately without waiting for the asynchronous operation to complete, which could lead to incorrect or incomplete results.