Laboratory - Deploying a Deep Learning model (Local setup)

Owner	Ⓓ Dennis Dosso
Tags	

Repository GitHub

La mia repository GiutHub con i miei commenti.

La repository fa parte di una repository relativa a tutta la specializzazione, quindi 4 corsi, e le diverse settimane per ogni corso. Questo laboratorio è relativo alla prima settimana del primo corso.

Il clone della repository si fa come segue:

```
git clone https://github.com/https-deeplearning-ai/machine-learning-engineering-for-production-public.git
cd machine-learning-engineering-for-production-public/course1/week1-ungraded-lab
```

Consiglio **VIVAMENTE** di usare Anaconda come indicato da loro, in quanto il modello di ML da loro usato appartiene ad una libreria che, importata non tramite conda, sembra dare molti problemi per i quali non è molto chiaro come dovremmo fare per risolverli.

```
conda create --name mlep-w1-lab python=3.8
conda activate mlep-w1-lab
pip install -r requirements.txt
jupyter lab
```

A questo punto, abbiamo 2 notebook Python, uno che funge da client ed un'altro che funge da server. Partiamo dal server.

Chiariamo alcuni concetti

Quando di parla di **deploying,** quello che si intende tipicamente è *mettere tutto il software che serve per eseguire la predizione all'interno di un server.* In questo modo, un client può interagire con il modello mandando richieste (requests) al server. Per quel che ci riguarda, quello che importa qui è che il modello ML **vive dentro il server attendendo che il client sottometta delle richieste di predizione.** Il client quindi dovrebbe provvedere le informazioni necessarie che servono al modello per poter fare una predizione. È spesso cosa comune, inoltre, fare un **batch** di richieste di predizione e metterle in una unica request. A quel punto il server farà le sue predizioni, e le manderà al client.

FastAPI

Si parte creando una istanza della classe FastAPI.

```
app = FastAPI()
```

A questo punto, lo step successivo è usare l'istanza per creare degli **endpoint** che gestiranno la logica di predizione. Una volta che tutto il codice viene messo in piedi, il server si può far correre con il comando:

```
uvicorn.run(app)
```

La API è codificata usando fastAPI, mentre il serving è performato usando **uvicorn**, che è una possibile implementazione di una Asynchronous Server Gateway Interface (ASGI).

> 🔑 **Asynchronous Server Gateway Interface (ASGI)** is a specification ffor building web applications in Python that can handle both synchronous and asynchronous programming models. ASGI is designed to improve the performance and scalability of web applications by allowing them to handle multiple simultaneous connections efficiently. In traditional synchronous web frameworks, each incoming request is handled sequentially, blocking the execution of subsequent requests until the current one is complete. This can lead to performance bottlenecks, especially in scenarios where there are long-running tasks or high concurrency. ASGI introduces an asynchronous programming model that enables web servers to handle multiple requests concurrently without blocking other requests. It allows web applications to utilize asynchronous frameworks and libraries, such as asyncio, to handle I/O-bound operations efficiently. This is particularly useful for applications that involve tasks like interacting with databases, making API calls, or performing computationally expensive operations.

FastAPI e Uvicorn sono due componenti separate che sono usate spesso assieme nello sviluppo web in Python. FastAPI è un **framework web** per costruire API con Python. È creato con l'obiettivo di essere di facile utilizzo, altamente efficiente, e di provvedere una grande produttività per gli sviluppatori. FastAPI sfrutta i type hints di Python per provvedere validazione alle response e request HTTP, così anche per generare delle documentazioni per le API. Uvicorn invece è il **web server** ad alte prestazioni che permette di far correre applicazioni ASGI. Si preoccupa di gestire le connessioni HTTP, gestire la concorrenza, e fare il dispatching delle richieste alle appropriate applicazioni ASGI. FastAPI, come web framework, sfrutta le capacità di uvicorn per gestire le chiamate HTTP in ingresso e provvedere il processing. Insieme, permettono di creare API web in Python.

Ricordiamo, per maggiore chiarezza, che un **web server** serve per gestire le chiamate HTTP che arrivano dai vari client e restituire loro le risposte associate. La sua funzione primaria è quella di gestire le comunicazioni tra un client e una applicazione web. Di fatto, lo fanno gestendo il routing delle richieste al corretto **endpoint**, facendo il parsing degli header HTTP, gestendo le connessioni, e mandando le risposte indietro ai client. Un **web framework** è una collezione di librerie, tool e pattern che mettono a disposizione una base di partenza per costruire applicazioni web. Mettono cioè a disposizione delle astrazioni, utilities e dei pattern pre-definiti per gestire task comuni come il routing, il request handling, la data persistence, autenticazione, ecc.

Endpoints

Molti modelli di ML possono essere posti nello stesso server. Per poter lavorare correttamente però, serve assegnare un endpoint diverso a ciascuno di loro, dimodoché si possa sempre sapere quale è il modello che si sta usando. Un endpoint, ricordiamo, è identificato da un pattern nell'URL.

In FastAPI si definisce un endpoint creando una funzione che gestirà tutta la logica di quell'endpoint. Si sfruttano le decorazioni python per specificare il metodo HTTP richiesto e il pattern url che si sta usando per identificar el'endpoint stesso.

```
@app.get("/my-endpoint")
def handle_endpoint():
    ...
```

HTTP Requests

Se un client esegue una richiesta GET all'endpoint di un server, tipicamente è per richiedere dell'informazioni da quell'endpoint **senza la necessità di aggiungere informazioni addizionali** da parte sua. Questo è solo parzialmente vero, perché si possono passare delle richieste GET con dei parametri nell'url, ma sticazzi qui.

Quando si interagisce con dei modelli di ML che vivono all'interno di endpoints solitamente vengono fatte tramite una request POST, dato che serve dare informazioni al modello affinché possa fare la sua predizione. Pertanto, una request di tipo POST avrà questa forma:

```
@app.post("/my-other-endpoint")
def handle_other_endpoint(param1: int, param2: str):
    ...
```

Si nota che la funzione associata alla richiesta POST ha dei parametri nella sua firma.

FastAPI

Ci permette di creare il web server molto velocemente e facilmente. Inoltre, ha anche un **client built-in** che permette un testing altrettanto veloce dell'applicazione, messo a disposizione nell'endpoint /docs, senza che ce lo dobbiamo creare noi from scratch ogni volta che si fa prototipazione. Per accedere a questo endpoint, basta attivarlo e poi navigare all'url http://localhost:8000/docs.

Con FastAPI, è possibile all'inizio creare una istanza della classe FastAPI, la quale rappresenta l'applicazione API e viene anche usata per definire gli endpoints e gestire le richieste HTTP.

```
app = FastAPI(title='Deploying a ML Model with FastAPI')
```

A questo punto, si possono usare i decorator per definire gli endpoint. Ad esempio, per l'endpoint Home, si definisce una funzione **home()** e la si decora come segue:

```
# By using @app.get("/") you are allowing the GET method to work for the / endpoint.
@app.get("/")
def home():
    return "Congratulations! Your API is working as expected. Now head over to http://localhost:8000/doc"
```

uesto fa sì che si possa accedere, tramite l'url http://localhost:8000/, ad un messaggio di testo.Chiaramente, prima serve far partire l'applicazione, qui è solo definito il server.

Per l'endpoint POST, invece:

```
# This endpoint handles all the logic necessary for the object detection to work.
# It requires the desired model and the image in which to perform object detection.
@app.post("/predict")
def prediction(model: Model, file: UploadFile = File(...)):

    # 1. VALIDATE INPUT FILE
    filename = file.filename
    fileExtension = filename.split(".")[-1] in ("jpg", "jpeg", "png")
    if not fileExtension:
        raise HTTPException(status_code=415, detail="Unsupported file provided.")

    # 2. TRANSFORM RAW IMAGE INTO CV2 image

    # Read image as a stream of bytes
    image_stream = io.BytesIO(file.file.read())

    # Start the stream from the beginning (position zero)
    image_stream.seek(0)

    # Write the stream of bytes into a numpy array
    file_bytes = np.asarray(bytearray(image_stream.read()), dtype=np.uint8)

    # Decode the numpy array as an image
    image = cv2.imdecode(file_bytes, cv2.IMREAD_COLOR)
```

```
    # 3. RUN OBJECT DETECTION MODEL

    # Run object detection
    bbox, label, conf = cv.detect_common_objects(image, model=model)

    # Create image that includes bounding boxes and labels
    output_image = draw_bbox(image, bbox, label, conf)

    # Save it in a folder within the server
    cv2.imwrite(f'images_uploaded/{filename}', output_image)


    # 4. STREAM THE RESPONSE BACK TO THE CLIENT

    # Open the saved image for reading in binary mode
    file_image = open(f'images_uploaded/{filename}', mode="rb")

    # Return the image as a stream specifying media type
    return StreamingResponse(file_image, media_type="image/jpeg")
```

La funzione si occupa di validare l'input dell'immagine, assicurandosi he essa sia nella corretta estensione file. Trasforma il file in una immagine CV2, convertendola cioè in un array numpy e poi ricodificandola in una immagine. Il metodo detect_common_objects viene poi fatto correre per individuare nell'immagine gli oggetti che vi compaiono, assieme alle loro label e i confidence score. Infine, disegna le box sull'immagine, le salva su disco e le rimanda al client.

Infine, serve far partire il server:

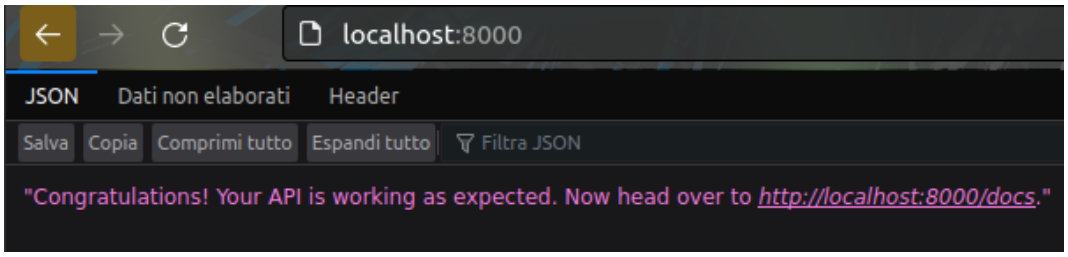
```
# Allows the server to be run in this interactive environment
nest_asyncio.apply()

# Host depends on the setup you selected (docker or virtual env)
host = "0.0.0.0" if os.getenv("DOCKER-SETUP") else "127.0.0.1"

# Spin up the server!
uvicorn.run(app, host=host, port=8000)
```

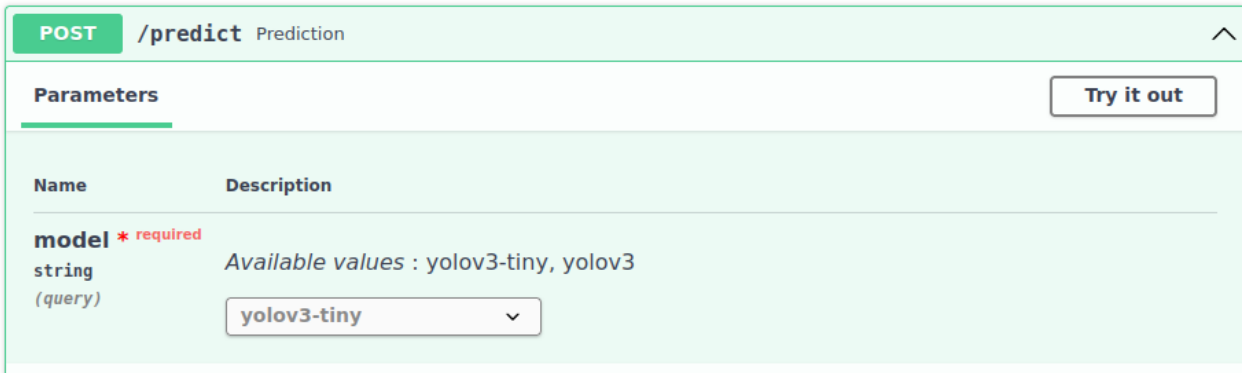
In questo caso, si permette al server di correre in un ambiente interattivo, come un notebook Jupyter. Si controlla se stiamo correndo su Docker, in quel caso si ascolta su tutte le porte, o in altro ambiente, in quel caso si ascolta solo sul localhost. Infine, si fa partire il server usando la funzione run, che richiede l'application instance di FastAPi creata prima, l'indirizzo host al quale il server ascolterà, e la port che si userà, settata qui a 8000.

Il risultato sarà questa semplice schermata:

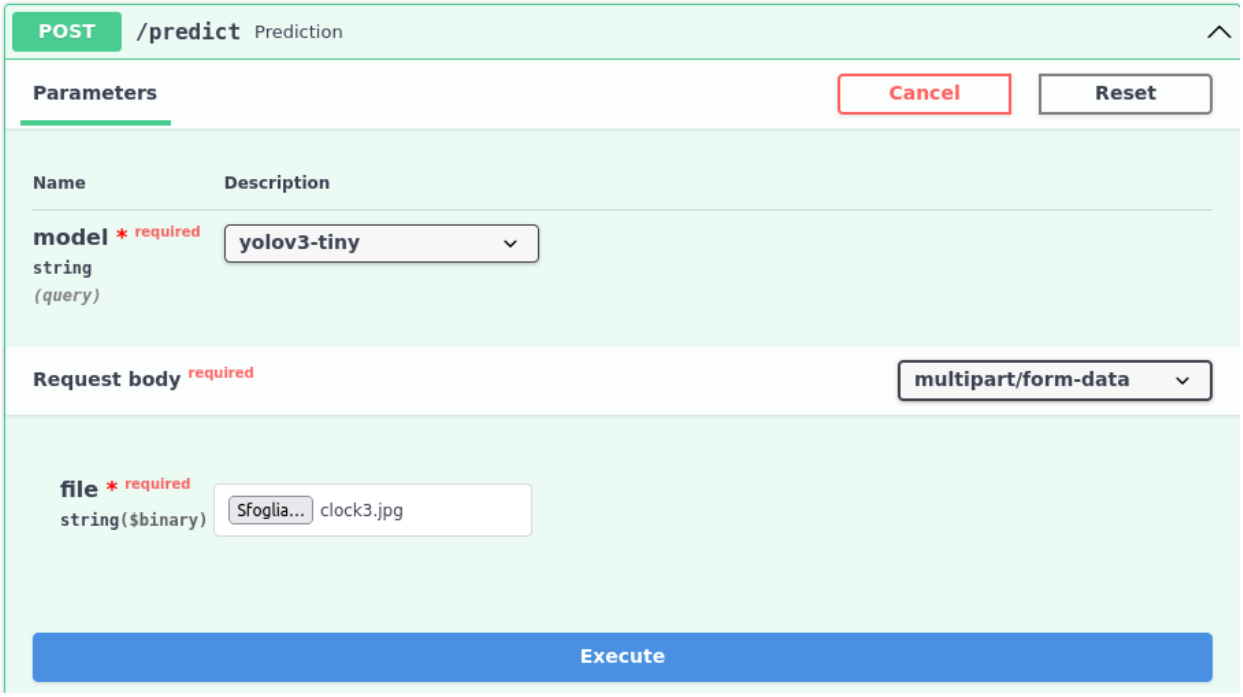


A questo punto si può usare il **client di default** reso disponibile dall'api docs. Questo, al quale si arriva seguendo l'url indicato qui sopra, appare come segue:

Ora, quello che vediamo è che ci sono i 2 **endpoint** da noi definiti precedentemente, la home e la funzione **predict**. Cliccando sull'espansione, appare la documentazione swagger con il nome dei parametri che ci sono richiesti. Sono **model:**



Come si nota, richiesto e sceglibile tramite un menu a tendina. Poi vi è il **request body**, tipico di una richiesta REST, dove possiamo solo scegliere il *multipart/form-data* e infine il path del file richiesto. Per eseguire, ad esempio:



A questo punto non si fa altro che pigiare Execute. Il risultato è il seguente (inoltre il server ha salvato l'immagine in se stesso).

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:8000/predict?model=yolov3-tiny' \
  -H 'accept: application/json' \
  -H 'Content-Type: multipart/form-data' \
  -F 'file=@clock3.jpg;type=image/jpeg'
```

Request URL

```
http://localhost:8000/predict?model=yolov3-tiny
```

Server response

Code	Details
200	**Response body**

Response headers

```
content-type: image/jpeg
date: Tue,03 Oct 2023 14:04:38 GMT
server: uvicorn
transfer-encoding: chunked
```

Si vede prima la richiesta fatta, tramutata in curl, e il response body dal server, che consiste nell'immagine elaborata e negli header.

In the context of REST (Representational State Transfer) and HTTP (Hypertext Transfer Protocol), `multipart/form-data` is a type of content type or media type used for encoding and transmitting binary or non-textual data as part of an HTTP request. It is commonly used when you need to upload files or send complex data structures, such as forms with file attachments, via an HTTP POST request.

When you use `multipart/form-data`, the data in the request body is divided into multiple parts or sections, each with its own set of headers and data. Each part can represent a field and its associated value or a file attachment. This format allows for the transmission of binary data, making it suitable for file uploads.

Di seguito un file HTML contenente il notebook:

server.html