

Réalisé par : Mouad GUEDAD

1. Servir un modèle NLP (transformer) en utilisant FastAPI

Nous allons utiliser un modèle de QA (Question answering) et le servir en utilisant FastAPI. Pour ce faire, nous allons utiliser la bibliothèque « transformers » pour en utiliser des modèles de langages basés sur une architecture Transformer-Encoder et plus spécifiquement nous allons utiliser DistillBERT qui est une version allégée de BERT « Bidirectional Encoder Representations from Transformers » qui appartient à une famille nommée « autoencoding language models ».

Dans cette partie, nous allons utiliser le modèle DistillBERT (une version allégée de BERT) pré-entraîné sur le dataset SQUAD « The Stanford Question Answering Dataset » (<https://rajpurkar.github.io/SQuAD-explorer/>).

Dans le dossier de l'atelier, ouvrez le dossier « 1. fastAPI_Transformer_model_serving » et ensuite le fichier « main.py » :

```
1 import uvicorn
2 from fastapi import FastAPI
3 from pydantic import BaseModel
4
5 class QADataModel(BaseModel):
6     question: str
7     context: str
8
9 app = FastAPI()
10
11 from transformers import pipeline
12 model_name = 'distilbert-base-cased-distilled-squad'
13 model = pipeline(model=model_name, tokenizer=model_name, task='question-answering')
14
15 @app.post("/question_answering")
16 async def qa(input_data: QADataModel):
17     result = model(question = input_data.question, context=input_data.context)
18     return {"answer": result["answer"]}
19 if __name__ == '__main__':
20     uvicorn.run('main:app', workers=1)
```

En utilisant uvicorn, nous pouvons démarrer notre API.

```
{
  "question": "What is extractive question answering?",
  "context": "Extractive Question Answering is the task of extracting an answer from a text given a question. An example of a question answering dataset is the SQuAD dataset, which is entirely based on that task. If you would like to fine-tune a model on a SQuAD task, you may leverage the 'run_squad.py'."
}
```

Execute

Clear

Successful Response

Media type

application/json

Controls Accept header.

Example Value | Schema

"string"

Nous allons essayer maintenant un outil pour interroger notre API à l'aide des requêtes curl. Postman est une interface graphique facile à utiliser (lien de téléchargement : <https://www.postman.com/downloads/>).

New Collection / <http://127.0.0.1:8000/home> Save ... 🔗 💬

POST http://127.0.0.1:8000/question_answering Send

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** Beautify

```
1 {
2   "question": "What is extractive question answering?",
3   "context": "Extractive Question Answering is the task of extracting an answer from a text given a question.
      An example of a question answering dataset is the SQuAD dataset, which is entirely based on that task.
      If you would like to fine-tune a model on a SQuAD task, you may leverage the 'run_squad.py'."
4 }
```

Body Cookies Headers (4) Test Results 🌐 200 OK 231 ms 198 B Save Response

Pretty Raw Preview Visualize **JSON** 🔗

```
1 {
2   "answer": "the task of extracting an answer from a text given a question"
3 }
```

2. La conteneurisation de notre API en utilisant Docker

Pour gagner du temps en production et faciliter le processus de déploiement, il est essentiel d'utiliser Docker. Il est très important d'isoler votre service et votre application. Notez également que le même code peut être exécuté n'importe où, quel que soit le système d'exploitation.

Ouvrez le dossier « 2. Dockerizing_API », les étapes de dockerisation de notre API peuvent se résumer comme suit :

1. Mettez le fichier main.py dans le dossier « app ».
2. Ensuite, vous devez éliminer la dernière partie du fichier main.y :

```
if __name__ == '__main__':  
    uvicorn.run('main:app', workers=1)
```

```
1 import uvicorn  
2 from fastapi import FastAPI  
3 from pydantic import BaseModel  
4  
5 class QADataModel(BaseModel):  
6     question: str  
7     context: str  
8  
9 app = FastAPI()  
10  
11 from transformers import pipeline  
12 model_name = 'distilbert-base-cased-distilled-squad'  
13 model = pipeline(model=model_name, tokenizer=model_name, task='question-answering')  
14  
15 @app.post("/question_answering")  
16 async def qa(input_data: QADataModel):  
17     result = model(question = input_data.question, context=input_data.context)  
18     return {"answer": result["answer"]}
```

3. Ensuite, vous devez créer un Dockerfile pour votre fastAPI :

```
1 FROM python:3.8  
2  
3 RUN pip install torch  
4  
5 RUN pip install fastapi uvicorn transformers  
6  
7 EXPOSE 80  
8  
9 COPY ./app /app  
10  
11 CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

4. Finalement, vous pouvez construire votre conteneur Docker

5. Et le lancer avec la commande suivante :

```
docker run -p 8000:8000 qaapi
```

3. Servir un modèle de type transformer en utilisant TFX

Jusqu'à maintenant, nous avons pu déployer notre modèle comme étant un modèle intégré dans une application (voir le 3 ème cours). Nous allons maintenant déployer notre modèle à

```
Removing intermediate container 6e25c81a73b8
---> 316c83997ff7
Step 4/6 : EXPOSE 80
---> Running in 1ae65d74e7bb
Removing intermediate container 1ae65d74e7bb
---> 85693b79fd9f
Step 5/6 : COPY ./app /app
---> 91065a07867d
Step 6/6 : CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
---> Running in fb331b0d4bb3
Removing intermediate container fb331b0d4bb3
---> d3c859a5da39
Successfully built d3c859a5da39
Successfully tagged qaapi:latest
```

travers une API dédiée.

TFX fournit un moyen plus rapide et plus efficace de servir des modèles deep learning. Mais il a quelques points clés importants que vous devez comprendre avant de l'utiliser. Le modèle doit être un type de modèle enregistré à partir de TensorFlow afin qu'il puisse être utilisé par TFX. Pour plus d'informations sur les modèles enregistrés TensorFlow, vous pouvez lire la documentation officielle : https://www.tensorflow.org/guide/saved_model

Ouvrez le dossier « 3. Faster_Transformer_model_serving_using_Tensorflow_Extended » et exécutez le notebook « saved_model.ipynb » afin de produire le modèle enregistré.

```
model.save_pretrained("tfx_model", saved_model=True)

WARNING:absl:Found untraced functions such as embeddings_layer_call_fn, embeddings_layer_call_and_return_conditional_losses, encoder_layer_call_fn, encoder_layer_call_and_return_conditional_losses, pooler_layer_call_fn while saving (showing 5 of 420). These functions will not be directly callable after loading.

INFO:tensorflow:Assets written to: tfx_model/saved_model/1/assets
INFO:tensorflow:Assets written to: tfx_model/saved_model/1/assets
```

Ensuite, nous allons extraire l'image Docker pour Tensorflow Extended (pour plus d'informations : <https://www.tensorflow.org/tfx/serving/docker>) :

Maintenant, nous allons exécuter le conteneur Docker et y copier le modèle enregistré.

Cela va copier le modèle enregistré vers le conteneur. Toutefois, nous devons valider le changement.

Maintenant que tout marche bien, nous pouvons arrêter le conteneur.

```

docker pull tensorflow/serving
[sudo] Mot de passe de youssef :
Using default tag: latest
latest: Pulling from tensorflow/serving
a404e5416296: Extracting [=====>] 26.71MB/26.71MB
a404e5416296: Pull complete
e0811bab055f: Pull complete
70566ae4c4ea: Pull complete
982266225230: Pull complete
cdb50cf1c58b: Pull complete
Digest: sha256:e66c1866bf6596473d56beb3ef77bf7b8b00baa0e1e80c3c428ede31da8ae066
Status: Downloaded newer image for tensorflow/serving:latest
docker.io/tensorflow/serving:latest

```

```
$ docker kill serving_base
```

Maintenant que le modèle est prêt et peut être servi par TFX Docker, vous pouvez simplement l'utiliser avec un autre service. La raison pour laquelle nous avons besoin d'un autre service pour appeler TFX est que les modèles basés sur Transformer ont un format d'entrée spécial fourni par les tokenizers (ça veut dire le texte doit être traité avant qu'il soit passé au modèle). Pour ce faire, vous devez créer un service fastAPI qui appellera l'API qui a été servie par le conteneur TensorFlow serving. Avant de coder votre service, vous devez démarrer le conteneur Docker en lui donnant des paramètres pour exécuter le modèle d'analyse de sentiment basé sur BERT.

Images [Give Feedback](#)

LOCAL REMOTE REPOSITORIES

🔍 Search


☐ In use only

NAME ↑		TAG	IMAGE ID	CREATED	SIZE
fastapi		latest	be62adfbda59	25 days ago	995.38 MB
my_bert_model	IN USE	latest	ea80babc6ba3	2 minutes ago	906.49 MB
tensorflow/serving	IN USE	latest	3d0c9b293f81	about 1 month ago	459.21 MB

Containers [Give Feedback](#)

A container packages up code and its dependencies so the application runs quickly and reliably from one computing environment to another. [Learn more](#)

Showing 1 Items 🔍 ea80babc6ba3 ⋮

<input type="checkbox"/>	NAME	IMAGE	STATUS	PORT(S)	STARTED	ACTIONS
<input type="checkbox"/>	 bert f908f279f1f1	my_bert_model:latest	Created	8500,8...		▶ ⋮ 🗑️

Maintenant que nous avons notre service TFX avec Docker prêt à être consommé en utilisant REST API (le port 8501), nous allons le consommer en utilisant FastAPI.

Ouvrez maintenant le fichier « main.py » et regardez son contenu. **Qu'est-ce que vous**

remarquez ?

????????????????????????????????

Exécutons-le en utilisant la commande « python main.py ».

POST	/sentiment	Sentiment Analysis
Parameters		
No parameters		
Request body required		
<pre>{ "text": "i love you" }</pre>		
Response body		
<pre>{ "sentiment": "POSITIVE" }</pre>		

Le service est maintenant est prêt à être utilisé (127.0.0.1:8000/docs). Toutefois, vous pouvez utiliser Postman pour l’interroger.

New Collection / <http://127.0.0.1:8000/home> Save ...

POST ▼ <http://127.0.0.1:8000/sentiment>

Params Authorization Headers (8) **Body** ● Pre-request Script Tests Settings

● none ● form-data ● x-www-form-urlencoded ● **raw** ● binary ● GraphQL **JSON** ▼

```
1 {
2   "text": "i love you"
3 }
4
5
```

Body Cookies Headers (4) Test Results 200 OK 234 ms 149 B Sa

Pretty Raw Preview Visualize **JSON** ▼ ≡

```
1 {
2   "sentiment": "POSITIVE"
3 }
```

4. Le Test de chargement en utilisant Locust

Il existe de nombreuses applications que nous pouvons utiliser pour effectuer des tests de chargement. La plupart de ces applications et bibliothèques fournissent des informations utiles sur le temps de réponse et le délai du service. Ils fournissent également des informations sur le taux d'échec. Locust est l'un des meilleurs outils à cet effet. Nous l'utiliserons pour tester le chargement des trois méthodes vues précédemment :

- ☐ En utilisant fastAPI uniquement,
- ☐ En utilisant fastAPI dockerisé et
- ☐ En le servant le modèle avec TFX en utilisant fastAPI.

Commençons d'abord par l'installation de Locust.

```
$ sudo pip install locust
```

Nous pouvons maintenant tester vos APIs (services), vous devez préparer un fichier locust dans lequel vous allez définir votre utilisateur et son comportement. Ci-dessous un exemple de fichier « locust_file.py » dans lequel nous allons tester le dernier modèle déployé (i.e., TFX avec FastAPI pour l'analyse de sentiment)

Pour API question_Answering :

```
from locust import HttpUser, task
from random import choice
from string import ascii_uppercase
class User(HttpUser):
    @task
    def predict(self):
        payload = {"question": ''.join(choice(ascii_uppercase) for i in range(20)),
                  "context": ''.join(choice(ascii_uppercase) for i in range(100))}
        self.client.post("/question_answering", json=payload)
```

Start new load test

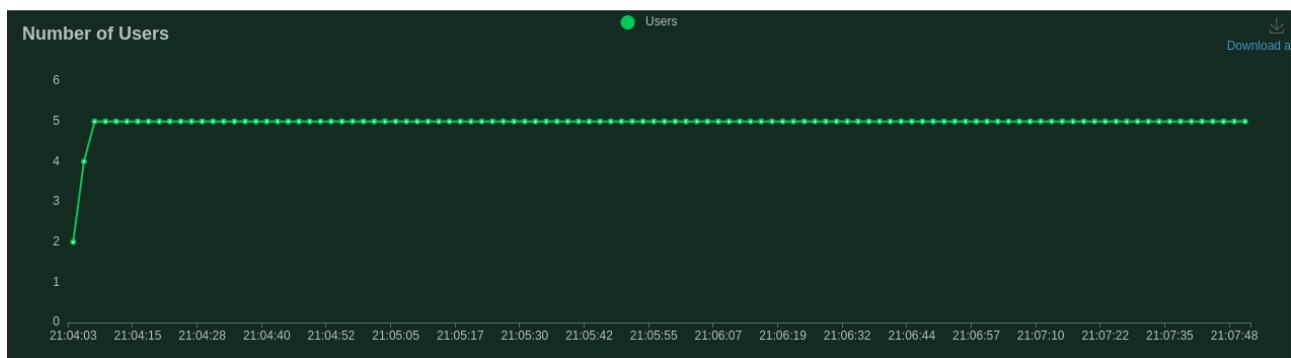
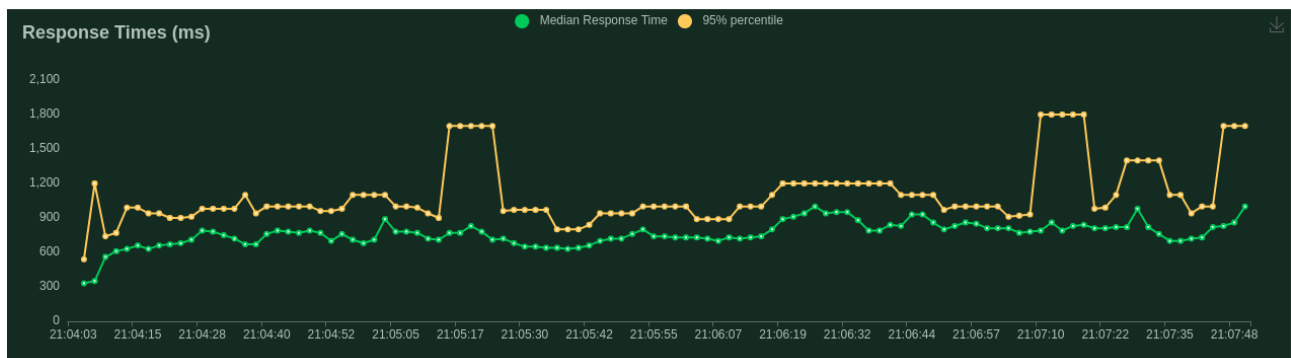
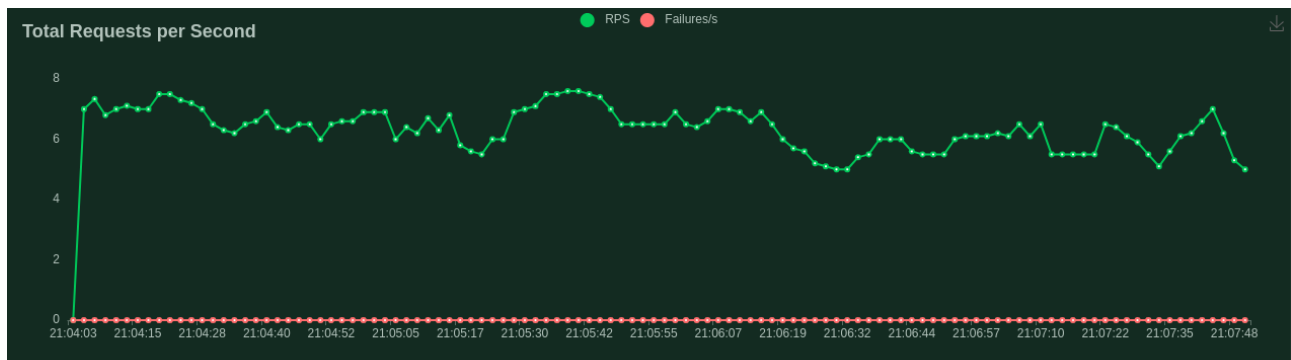
Number of users (peak concurrency)

Spawn rate (users started/second)

Host (e.g. http://www.example.com)

Advanced options

Start swarming



Maintenant, Testons les trois versions et comparons les résultats pour voir lequel fonctionne le mieux.

	TFX-based FastAPI	FastAPI	Dockerized FastAPI
RPS	5	7	7
Average RT(ms)	505	801	710

Conclusion :

Le meilleur déploiement est : TFX-based FastApi.