University Mohammed VI Polytechnic

Lab: Model testing
Professor : Fahd KALLOUBI
Year : 2025/2026
Major : CS3 – College of computing

Dans cet atelier, nous allons tester la performance de plusieurs déploiements à savoir :

- Un déploiement en utilisant FastAPI
- Un déploiement en utilisant Docker pour conteneuriser une API
- Un déploiement en utilisant Tensorflow Extended (TFX) pour servir un model

Pour de plus amples information, essayez de consulter les ressources suivantes :

- Locust documentation: https://docs.locust.io
- TFX documentation: https://www.tensorflow.org/tfx/guide
- FastAPI documentation: https://fastapi.tiangolo.com
- Docker documentation: https://docs.docker.com
- HuggingFace TFX serving: https://huggingface.co/blog/tf-serving

Les packages suivants doivent être installés :

- TensorFlow
- PyTorch
- transformers >=4.00
- fastAPI
- Docker
- Locust

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 allégé de BERT) pré-entrainé 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. Nous allons commencer par la création d'un modèle de données en utilisant Pydantic pour modéliser les entrées (Question-Réponse).

```
from pydantic import BaseModel
class QADataModel(BaseModel):
    question: str
    context: str
```

2. Après avoir créé une instance de fastAPI, nous allons charger le modèle pré-entrainé en utilisant le module « pipeline » de la bibliothèque « transformers »

```
from transformers import pipeline
model_name = 'distilbert-base-cased-distilled-squad'
```

```
model = pipeline(model=model_name, tokenizer=model_name, task='question-answering')
```

3. Ensuite, nous allons créer un endpoint pour notre API, en créant une fonction en mode asynchrone.

```
@app.post("/question_answering")
async def qa(input_data: QADataModel):
    result = model(question = input_data.question, context=input_data.context)
    return {"answer": result["answer"]}
```

4. Finalement, et en utilisant uvicorn, nous pouvons démarrer notre API.

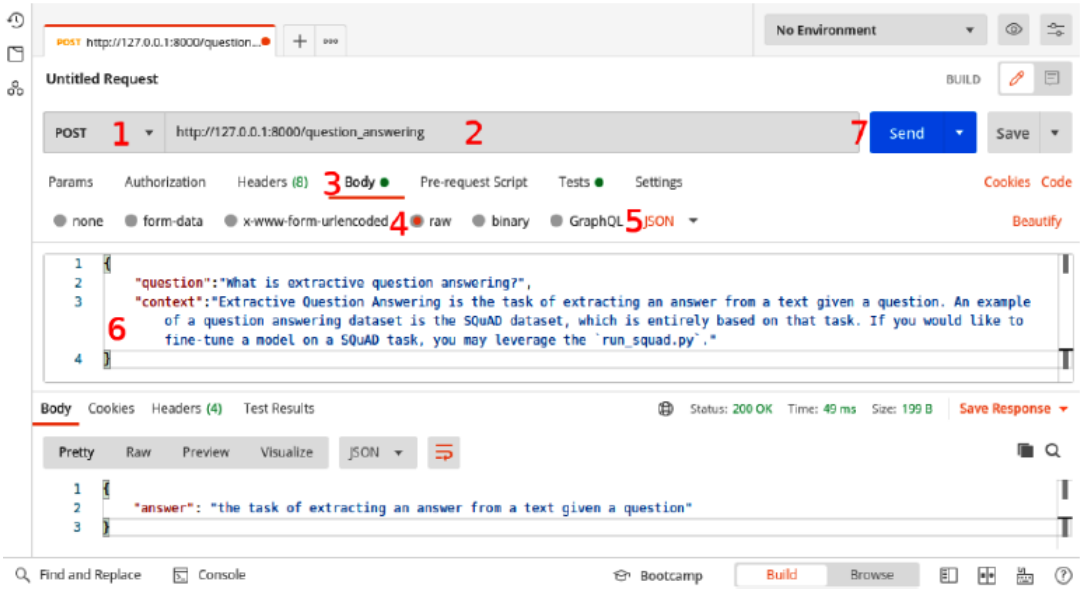
```
if __name__ == '__main__':
    uvicorn.run('main:app', workers=1)
```

Exécutez le fichier en tapant la commande « python main.py », et accédez ensuite à la documentation http://127.0.0.1:8000/docs.

Essayez avec cette entrée pour valider le fonctionnement de l'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`."
}
```

Nous allons essayez 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/).



Chaque étape de mise en place du service sur Postman est numérotée dans le figure :

1. Choisissez Post comme méthode
2. Entrez votre endpoint complet « http://127.0.0.1:8000/question_answering »
3. Sélectionnez Body
4. Ensuite Raw
5. Choisissez Json comme type de données
6. Les données doivent être entrées sous type Json
7. Cliquez sur Send

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)
```

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

```
FROM python:3.7

RUN pip install torch

RUN pip install fastapi uvicorn transformers

EXPOSE 80

COPY ./app /app

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8005"]
```

4. Finalement, vous pouvez construire votre conteneur Docker

```
docker build -t qaapi .
```

5. Et le lancer avec la commande suivante :

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

En conséquence, vous pouvez désormais accéder à votre API en utilisant le port 8000. Cependant, vous pouvez toujours utiliser Postman, comme décrit dans la section précédente.

3. Servir un modèle de type transformer en utilisant TFX

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

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

```
docker pull tensorflow/serving
```

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

```
docker run -d --name serving_base tensorflow/serving
```

```
docker cp tfx_model/saved_model serving_base:/models/bert
```

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

```
docker commit --change "ENV MODEL_NAME bert" serving_base
my_bert_model
```

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

```
docker kill serving_base
```

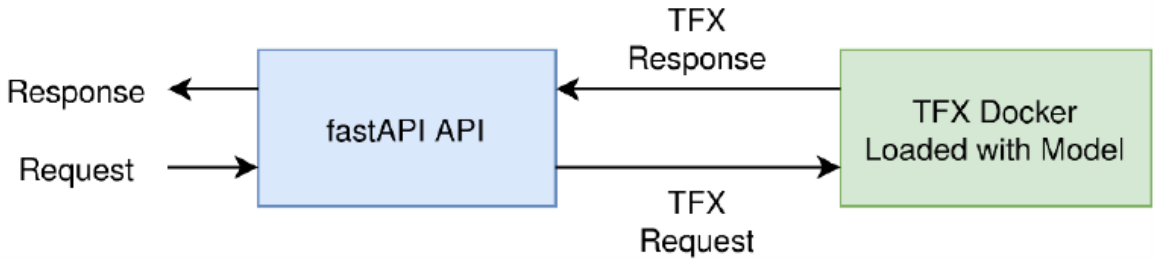
Cela arrêtera l'exécution du conteneur.
Maintenant que le modèle est prêt et peut être servi par TFX Docker, vous pouvez simplement l'utiliser avec un autre service. La raison pour laquelle nous avons besoin d'un autre service pour appeler TFX est que les modèles basés sur Transformer ont un format d'entrée 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 appelera 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.

```
docker run -p 8501:8501 -p 8500:8500 --name bert my_bert_model
```

- le port 8500 est exposé pour gRPC
- le port 8501 est exposé pour l'API REST

Gardez cette fenêtre ouverte pour que le service reste en marche car nous allons l'utiliser avec Fast API. L'architecture globale de la nouvelle API est la suivante :

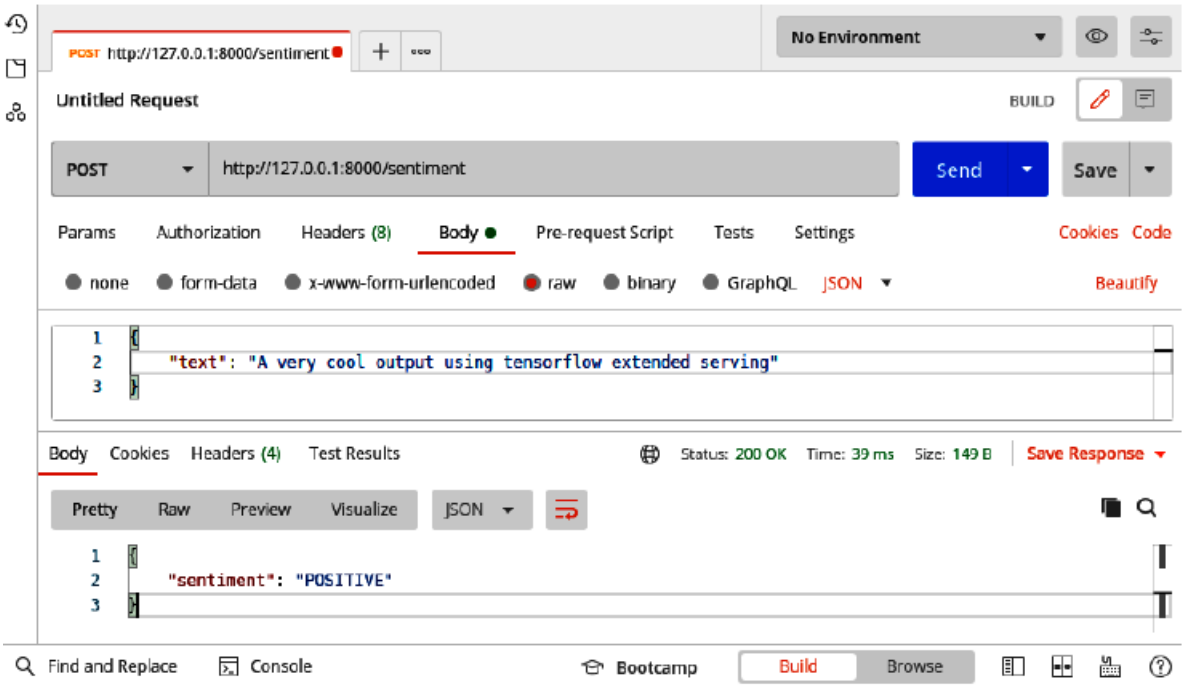


Maintenant que nous avons notre service TFX avec Docker prêt à être consommé en utilisant REST API (le port 8501), nous allons le consommer en utilisant FastAPI.

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

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

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



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.

Commencez d'abord par l'installation de Locust.

```
pip install locust
```

Vous pouvez 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)

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

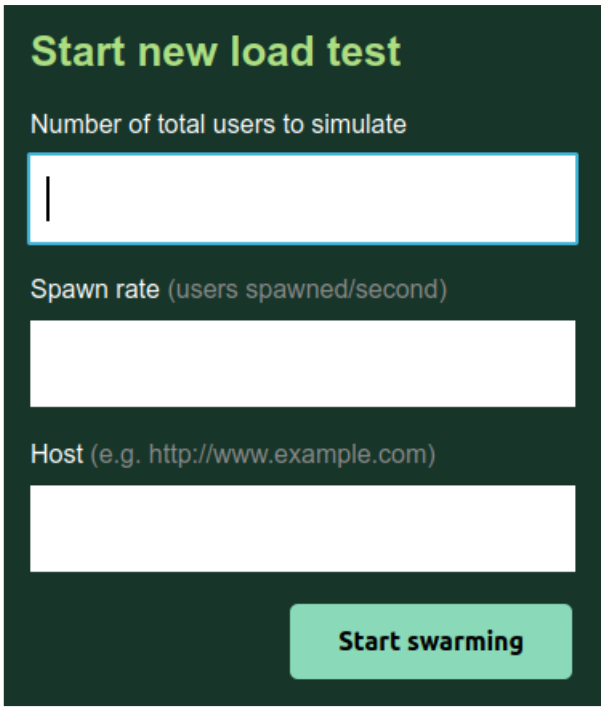
En utilisant HttpUser et en créant la classe User qui en hérite, nous pouvons définir une classe HttpUser. Le décorateur @task est essentiel pour définir la tâche que l'utilisateur doit effectuer. La fonction de prédiction est la tâche réelle que l'utilisateur effectuera à plusieurs reprises. Il générera une chaîne aléatoire d'une longueur de 20 et l'enverra à votre API.

Pour démarrer le test, lancez la commande « locust –f locust_file.py ».

Le service est disponible sur « http://localhost:8089/ ». une fois sur ce lien, vous allez trouver l'interface ci-dessous.

Nous allons définir le nombre total d'utilisateurs à simuler sur 10, le taux d'apparition sur 1 et l'hôte sur http://127.0.0.1:8000, c'est là que notre service s'exécute. Après avoir défini ces paramètres, cliquez sur « Start swarming ».

À ce stade, l'interface utilisateur changera et le test commencera. Pour arrêter le test à tout moment, cliquez sur le bouton Arrêter.



Vous pouvez cliquer sur « Charts » pour voir la visualisation des résultats.

Testez les trois versions et comparez les résultats pour voir lequel fonctionne le mieux. N'oubliez pas que les services doivent être testés indépendamment sur la machine sur laquelle vous souhaitez les servir. En d'autres termes, vous devez exécuter un service à la fois et le tester, fermer le service, exécuter l'autre et le tester, et ainsi de suite.

Remplissez le tableau ci-dessous avec les résultats approximatifs que vous avez eu.

Quel est le meilleur déploiement ?

	TFX-based FastAPI	FastAPI	Dockerized FastAPI
RPS			
Average RT(ms)			

Dans le tableau précédent, les requêtes par seconde (RPS) désignent le nombre de requêtes par seconde auxquelles l'API répond, tandis que le temps de réponse moyen (RT) désigne les millisecondes nécessaires au service pour répondre à un appel donné.