Comment créer et déployer une API de Machine Learning avec FastAPI ?

Auteurs : Victor Bigand

Temps de lecture : 12 minutes

La modélisation scientifique constitue le cœur du métier de data scientist, du cadrage à la création du modèle en passant par la préparation des données. D'après Venturebeat, 87% des projets IA en entreprise ne dépassent pas le stade du POC. Parmi les raisons de ces échecs, on trouve la séparation en silos des compétences, notamment les Data Scientists qui développent les modèles et les équipes en charge du déploiement de ces modèles. De ce constat est né le profil de Machine Learning Engineer, avec un rôle centré sur le déploiement de modèles en production. Ce tutoriel vous guide à travers un exemple simple de partage, au sein de votre équipe ou d'un groupe métier par exemple, d'un modèle de machine learning en s'appuyant en particulier sur la brique applicative FastAPI.

Il existe plusieurs frameworks pour créer une API en Python, les plus connus étant Django et Flask. FastAPI est un framework plus récent, inspiré de Flask, qui reprend la plupart de ses fonctionnalités et présente quelques nouveautés, notamment le fait de pouvoir gérer les appels asynchrones, valider les données d'entrée ou de sortie, sérializer les JSON par défaut ou encore une rédaction automatique de documentation standardisée.

Si vous n’êtes pas familiers avec ces notions, ne vous en faites pas, nous les reverrons plus loin dans cet article.

Voici le schéma représentant ce que nous allons mettre en place le long de ce tutoriel :

Nous allons donc développer une API avec FastAPI comprenant un modèle de Machine Learning (grâce à Tensorflow) et Uvicorn comme serveur web. Le tout sera packagé au sein d’une image Docker et déployé sur le cloud grâce à un service nommé Render, rendant l’API accessible à n’importe qui disposant d’une connexion web.

Voici les différentes parties que nous allons aborder:

- Spécifications de l’API et tests associés

- Ajout du modèle de classification d'images
- Test de l'API
- Déploiement de l'API
- Wrap-up et pistes d'améliorations

Avant de débuter, nous devons choisir quel est le but de l'API et comment elle sera manipulée par nos utilisateurs. Pour ce tutoriel, nous allons utiliser un modèle pré-entraîné de classification d'images. Les utilisateurs pourront envoyer une requête à l'API avec une image en paramètre, et recevront en retour une liste des deux classes les plus probables détectées par notre modèle.

Spécifications de l'API et tests associés

La première étape consiste à créer un squelette de code qui va initialiser l'API et contiendra une méthode de prédiction. Pour ce faire, nous allons créer un projet python avec votre éditeur de code préféré, ainsi qu'un environnement virtuel python associé. Ensuite, nous y ajoutons le fichier *requirements.txt* suivant :

```
fastapi==0.61.1
uvicorn==0.11.8
tensorflow==2.3.1
pillow==8.0.1
python-multipart==0.0.5
pytest==6.1.2
```

On y trouve:

- *Uvicorn*, un serveur *ASGI* (Asynchronous Server Gateway Interface) qui va permettre à notre application de communiquer de manière asynchrone avec le serveur que l'on va déployer. C'est une vraie différence par rapport à Flask, qui utilise *WSGI*, son prédécesseur, qui ne permettait nativement que l'exécution de requêtes synchrones. Dans le cas où une API effectue des tâches qui prennent du temps (dans notre cas, la lecture d'image n'est pas aussi rapide que l'envoi d'un nombre comme paramètre), l'asynchrone permet de traiter plusieurs requêtes en même temps en dissociant le traitement de la tâche du thread principal.
- *Tensorflow*, librairie phare de Deep Learning, pour charger notre modèle de classification d'images et effectuer des prédictions.

- *Pillow*, pour effectuer des manipulations sur les images (notamment les lire).
- *Python-multipart* pour que FastAPI puisse recevoir des images en paramètre.

Une fois le fichier créé, on peut installer ces dépendances avec la commande *pip install -r requirements.txt*.

Les librairies étant installées, nous allons définir un test vérifiant que l'API n'accepte que des images en entrée. Le fait de tester les données d'entrée, de sortie, ou le bon fonctionnement de l'API en amont nous permet d'éviter de tomber sur une erreur inattendue une fois l'API développée. Pour ce tutoriel, nous testerons uniquement le type des données reçues.

Pour cela, nous allons créer un dossier *tests* à la racine du projet, avec un fichier *test_predict.py*.

Nous allons ensuite ajouter un dossier *files* au sein de *tests*, contenant un fichier image et un fichier texte que vous pouvez ajouter manuellement (ex: un fichier *.jpg* et un fichier *.txt*)

Au sein du fichier test_predict, nous allons ajouter deux fonctions, qui vont chacune tester un appel de l'API avec un des fichiers.

```
from fastapi.testclient import TestClient
from app.main import app
client = TestClient(app)
def test_predict_image():
filepath = "tests/files/picture.jpg"
response = client.post(
"/predict", files={"file": ("filename", open(filepath, "rb"), "image/jpeg")}
)
assert response.status_code == 200
def test_predict_text():
filepath = "tests/files/text.txt"
response = client.post(
"/predict", files={"file": ("filename", open(filepath, "rb"), "text/plain")}
)
assert response.status_code == 400
assert response.json() == {"detail": "File provided is not an image."}
```

On importe le *TestClient*, qui nous permet de requêter l'API au sein de tests, ainsi que notre API qui n'a pas encore été créée.

Chacun des tests effectue une requête à l'API avec un des fichiers, et vérifie la réponse. Les requêtes sont envoyées sur la route */predict*. Une réponse 200 signifie que la requête s'est bien déroulée, 400 que le fichier envoyé n'est pas au bon format.

Il nous reste maintenant à créer l'API. Pour cela, on va ajouter un fichier *main.py* au sein d'un nouveau dossier *app* situé à la racine du projet.

```
from fastapi import FastAPI, File, HTTPException, UploadFile
app = FastAPI()
@app.post("/predict")
def prediction(file: UploadFile = File(...)):
# Initialize the data dictionnary that will be returned
response = {"success": False}
# Ensure that the file is an image
if not file.content_type.startswith("image/"):
raise HTTPException(status_code=400, detail="File provided is not an image.")
return response
```

Dans ce fichier, on retrouve la création de l'API à la ligne 4, ensuite on va définir notre méthode POST, accessible au endpoint */predict,* qui nous permet d'envoyer des données au serveur (contrairement à la méthode GET). Un simple décorateur @app.post suivi du endpoint nous permet d'associer la fonction *prediction* à l'API.

Pour l'instant, on ne fait que spécifier le paramètre *file* qui indique que l'on attend un fichier, et on vérifie que ce fichier est bien une image, sinon on renvoie un code d'erreur 400 et un message associé.

Avant de lancer les tests, il faut ajouter *pytest*, la librairie qui nous permet de lancer les tests, aux requirements. Pytest étant utilisée pour la phase de développement, on peut l'ajouter dans un autre fichier appelé *requirements_dev.txt* à la racine du projet, puis lancer la commande *pip install -r requirements_dev.txt*.

```
-r requirements.txt
pytest==6.1.2
```

On peut à présent lancer les tests avec la commande *python -m pytest* et vous devriez voir que nos deux tests s'exécutent correctement.

Voici l'arborescence du projet final, afin que vous puissiez correctement situer les différents fichiers que nous allons à présent rajouter:

```
.
├── Dockerfile
├── app
│   ├── __init__.py
│   ├── main.py
│   └── model.py
├── requirements.txt
├── requirements_dev.txt
└── tests
    ├── files
    │   ├── picture.jpg
    │   └── text.txt
    └── test_predict.py
```

Ajout du modèle de classification d'image

Il est temps d'ajouter à notre API notre modèle de classification d'image. Dans notre cas, nous allons directement utiliser un réseau de neurones pré-entraîné pour de la classification d'images. On pourrait également le *fine-tuner* sur un cas d'usage spécifique ou bien le remplacer par un autre algorithme, le principe reste le même : avoir une fonction *predict* qui sera appelée par notre API. Libre à vous d'insérer ici votre propre modèle, votre fonction train, predict et la préparation des données.

On va ajouter un fichier model.py dans le dossier *app* qui va contenir 3 fonctions essentielles.

```
import numpy as np
from PIL import Image
from tensorflow.keras.applications import ResNet50, imagenet_utils
from tensorflow.keras.applications.imagenet_utils import (decode_predictions,
preprocess_input)
from tensorflow.keras.preprocessing.image import img_to_array
def load_model():
"""
Loads and returns the pretrained model
"""
model = ResNet50(weights="imagenet")
print("Model loaded")
return model
```

```
def prepare_image(image, target):
# resize the input image and preprocess it
image = image.resize(target)
image = img_to_array(image)
image = np.expand_dims(image, axis=0)
image = preprocess_input(image)
return image
def predict(image, model):
# We keep the 2 classes with the highest confidence score
results = decode_predictions(model.predict(image), 2)[0]
response = [
{"class": result[1], "score": float(round(result[2], 3))} for result in results
]
return response
```

La première fonction va nous permettre de charger le modèle, ici un réseau ResNet50. Ce type de réseau de neurones, basé sur des couches de convolution, est particulièrement efficace pour traiter des images. Il est pré-entraîné sur imagenet, un dataset composé de 1000 classes différentes.

La fonction *prepare_image* apporte quelques modifications basiques à l'image de base : la passer à la shape définie par le paramètre target (ex: 224×224), convertir l'image en array numpy, ajouter une dimension qui correspond au numéro de batch que l'on fournit au modèle et enfin une fonction *preprocess_input* qui va s'assurer que nos données sont optimales pour être traitées par le réseau (normalisation des valeurs des pixels).

Enfin, la fonction *predict* prend en entrée une image et le modèle pour ressortir les deux classes ayant le score le plus haut.

Il ne nous reste plus qu'à modifier notre fonction *prediction* dans *main.py* pour prendre en compte ce modèle.

```
from io import BytesIO
from typing import List
import uvicorn
from fastapi import FastAPI, File, HTTPException, UploadFile
from model import load_model, predict, prepare_image
from PIL import Image
from pydantic import BaseModel
app = FastAPI()
model = load_model()
```

```
# Define the response JSON
class Prediction(BaseModel):
filename: str
content_type: str
predictions: List[dict] = []
@app.post("/predict", response_model=Prediction)
async def prediction(file: UploadFile = File(...)):
# Ensure that the file is an image
if not file.content_type.startswith("image/"):
raise HTTPException(status_code=400, detail="File provided is not an image.")
content = await file.read()
image = Image.open(BytesIO(content)).convert("RGB")
# preprocess the image and prepare it for classification
image = prepare_image(image, target=(224, 224))
response = predict(image, model)
# return the response as a JSON
return {
"filename": file.filename,
"content_type": file.content_type,
"predictions": response,
}
if __name__ == "__main__":
uvicorn.run("main:app", host="0.0.0.0", port=5000)
```

On charge le modèle au départ grâce à la fonction *load_model* précédemment définie. Le modèle sera chargé une seule fois au démarrage de l'API et non à chaque prédiction.

Ensuite, on définit une classe Prediction, qui hérite de *BaseModel* de *pydantic*. Cela nous permet de définir un schéma de réponse pour notre API. Ce schéma va valider les données en sortie et construire automatiquement la documentation comme on le verra par la suite. On retourne donc le nom du fichier, son type et les prédictions du modèle.

Troisièmement, dans la fonction *prediction* : on lit le fichier, le charge au format Image de la librairie *Pillow*, puis on le convertit au format RGB si jamais l'image était en échelle de gris.

Il nous reste à prétraiter l'image puis la fournir à la fonction prédiction et enfin retourner le résultat.

Vous remarquerez l'ajout du *async* avant la définition de la fonction, et le *await* avant la lecture du fichier. C'est ici qu'on bénéficie du mode asynchrone de FastAPI: notre process étant bloqué pendant la lecture du fichier (I/O operation), await lui dit qu'il a le temps de faire autre chose jusqu'à ce que le fichier soit lu (comme traiter une autre requête). On pourrait également ajouter ce *await* sur d'autres tâches consommatrices en ressources comme le preprocessing de l'image ou sa prédiction.

Test de l'API

Il ne nous reste plus qu'à lancer le serveur pour pouvoir tester une vraie requête. Avec le *uvicorn.run()* définit dans le *main* (ligne 47 de l'exemple de code précédent), nous pouvons lancer notre script avec la commande suivante : *python app/main.py*.

Comme évoqué en introduction, FastAPI génère une documentation automatique, qui de plus va nous permettre de tester l'API. Pour cela, rendez-vous sur http://0.0.0.0:5000/docs.

On y trouve une documentation interactive avec Swagger UI, respectant les spécifications OpenAPI. Avec OpenAPI, on a une documentation interactive, mais on peut aussi générer le code de l'API à partir de spécifications au format YAML ou JSON. Le format OpenAPI devient un standard pour la création d'APIs. Il est agnostique au langage de programmation, ce qui permet aux humains comme aux machines de comprendre le fonctionnement de l'API sans accéder au code source ou à la documentation.

On peut tester une prédiction très simplement comme sur la photo ci-dessous. Vous pouvez aussi tester une prédiction en ligne de commande avec CURL.

POST /predict Prediction

Parameters

No parameters

Request body required

file * required

string($binary) Choisir le fichier car

Execute Clear

Responses

Curl

```
curl -X POST "http://0.0.0.0:5000/predict" -H  "accept: application/json" -H  "Co
part/form-data" -F "file=@car.jpg;type=image/jpeg"
```

Request URL

```
http://0.0.0.0:5000/predict
```

Server response

Code	Details
200	Response body


```
{
  "filename": "car.jpg",
  "content_type": "image/jpeg",
  "predictions": [
    {
      "class": "sports_car",
      "score": 0.2750000059604645
    },
    {
      "class": "racer",
      "score": 0.17399999499320984
    }
  ]
}
```

Déploiement de l'API

Notre API fonctionnant localement, il nous reste à la déployer sur le cloud. On peut par exemple utiliser un service managé d'un cloud provider (ex: API Gateway d'AWS ou Google App Engine). L'avantage du serverless est multiple : L'infrastructure est gérée par le cloud provider (notamment les patchs de sécurité), l'élasticité des ressources en fonction de la demande (pas de crash de votre API si vous recevez un pic de requêtes) et enfin le prix puisque l'on paye uniquement les ressources que l'on consomme. D'autres plateformes proposent ce type de service, comme Heroku ou Render. Pour ce tutoriel, nous allons utiliser Render, qui va nous permettre de déployer notre API en quelques clics seulement.

Pour cela, nous packageons notre code sous la forme de conteneur afin qu'il fonctionne de la même manière qu'importe le serveur de déploiement, et qu'il soit scalable.

Il nous suffit de rajouter le fichier *Dockerfile* suivant à la racine de notre projet :

```
FROM tiangolo/uvicorn-gunicorn-fastapi:python3.8-slim
WORKDIR /app
COPY requirements.txt /tmp/
RUN pip install -r /tmp/requirements.txt
COPY app/ /app/
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "5000"]
```

- La première ligne définit l'image docker de base que l'on utilise. Dans notre cas, c'est une image FastAPI optimisée pour adapter la charge en fonction des ressources matérielles détectées. La version choisie est en python 3.8, dans une configuration slim. Le slim indique une image plus légère, dénuée de certaines fonctionnalités dont nous n'avons pas besoin ici.
- On définit le dossier /app comme répertoire de travail.
- On copie puis installe les requirements.
- Finalement, on lance uvicorn en ligne de commande, une alternative au lancement du script *main.py*

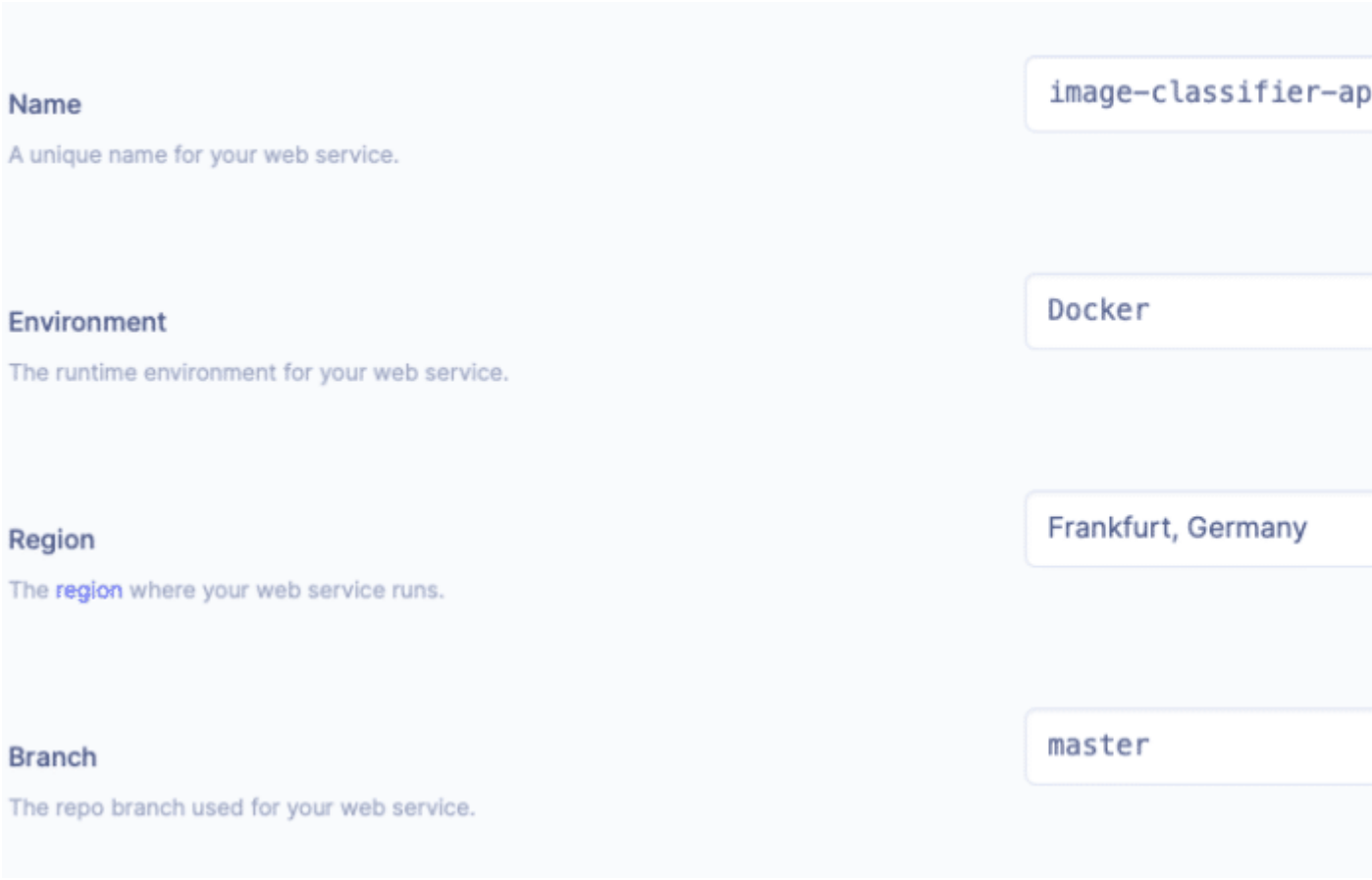
Il faut maintenant créer un compte sur Render et y connecter son compte github (ou gitlab), sur lequel vous allez devoir ajouter le projet.

Vous pouvez créer un projet vierge sur github, et ensuite le lier avec le code avec les commandes suivantes :

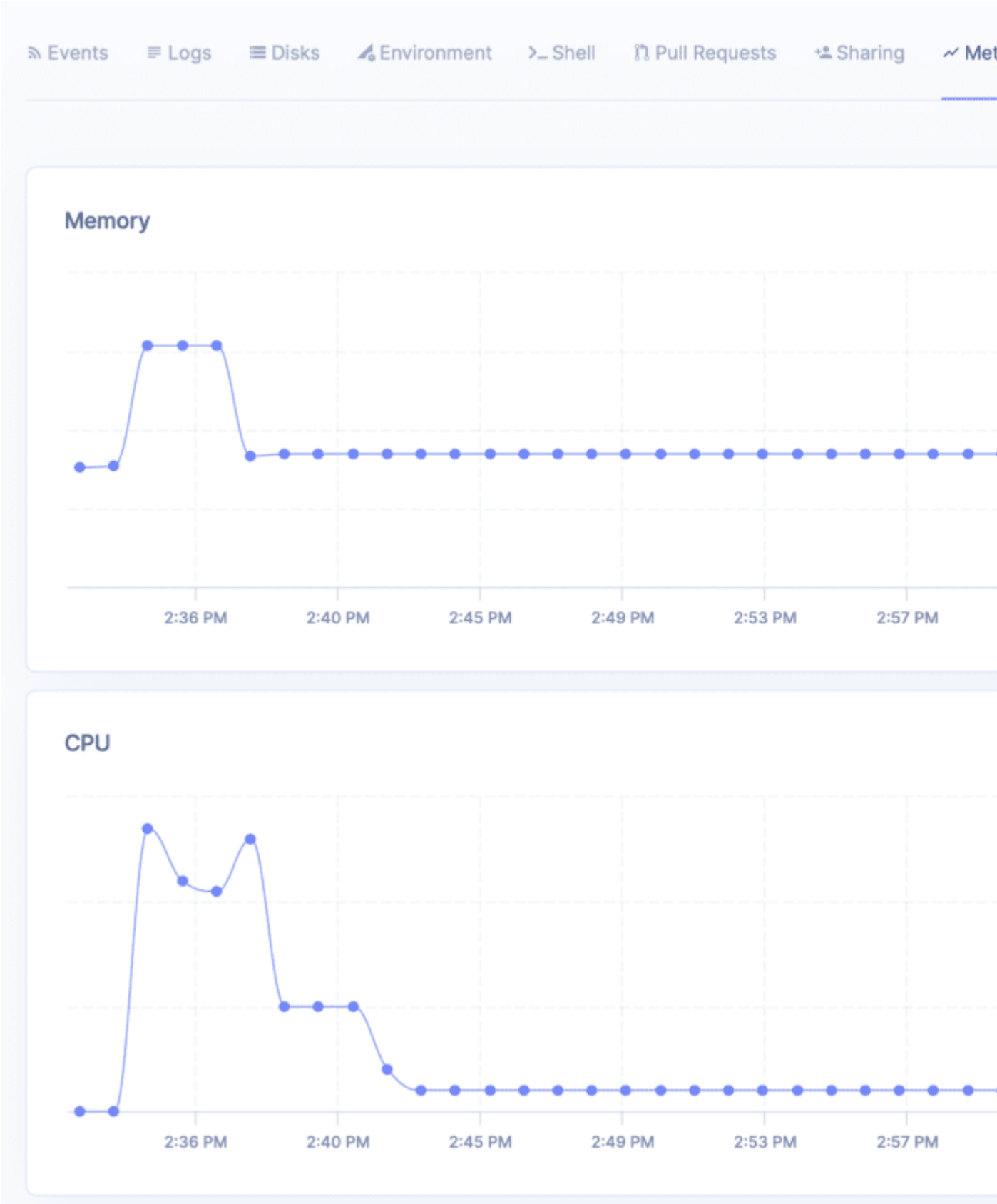
```
git init
git remote add origin link_ssh_to_your_repo
git add --all
git commit -m "Initialize API"
git push -u origin master
```

Render est une plateforme cloud permettant de déployer du code Python ou des images docker, et de bénéficier entre autres du déploiement continue (votre API sera mise à jour à chaque push sur votre projet), et de certificats SSL pour sécuriser les données qui transitent vers vos services.

Il ne vous reste qu'à créer un *web service* depuis le dashboard Render et la création de l'image docker se fera automatiquement. On indique le nom du service et la branche git sur laquelle Render va se baser. Il faut ensuite choisir les ressources dont vous avez besoin. La première semaine d'utilisation est gratuite (pas besoin de spécifier vos informations bancaires). Vous aurez ensuite un lien permettant d'accéder à l'API, et vous pourrez la tester de la même manière qu'en local précédemment, en ajoutant le */docs* après l'url. (Avec le nom indiqué sur l'image ci-dessous, mon url est donc https://image-api-classifier.onrender.com/docs).



Render met également à disposition des informations sur l'API, comme les logs des requêtes ou les ressources utilisées (RAM, CPU) :



Nous avons donc déployé simplement une API sur le cloud, en s’affranchissant des contraintes d’infrastructure.

Wrap-up et pistes d'améliorations

Dans ce tutoriel, nous avons vu comment déployer un modèle de Machine Learning avec une API. Bien sûr, FastAPI propose (tout comme Flask ou Django) de nombreuses autres fonctionnalités qu'il reste à découvrir, mais vous avez les clefs pour partager tout modèle d'IA sous la forme d'une API.

Voici quelques pistes pour la suite:

- Suivre les performances techniques (usage de la RAM ou du CPU, temps de réponse) mais aussi fonctionnelles : est-ce que les performances du modèle sont bonnes dans le temps ? Y a-t-il une dérive des données ? Pour cela, on peut déjà simplement logger les prédictions dans des des fichiers, et les analyser a posteriori. Pour une analyse plus fine du cycle de vie du modèle, vous trouverez cet article issu du blog Quantmetry.

- Comparer un nouveau modèle à celui actuellement en production. La plupart des plateformes cloud permettent de faire cela grâce au *canary release* (Un pourcentage des requêtes est dirigé vers le nouveau modèle afin d'évaluer ses performances en production).

- Dans la continuité du point précédent, si on modifie notre API, il est parfois nécessaire de la versionner (ex: *http://api/v1/predict* vs *http://api/v2/predict*) pour que l'ancienne version reste accessible.

- Pour améliorer la sécurité de l'API, on peut ajouter de l'authentification pour s'assurer que les personnes qui requêtent l'API en ont bien le droit. Oauth2 est le standard en la matière et est présent au sein de FastAPI.

J'espère que ce tutoriel vous a donné envie d'expérimenter le développement de votre API, avec votre propre modèle de Machine Learning. Pour aller plus loin, je ne peux que vous conseiller la documentation très complète de FastAPI dont nous n'avons qu'à peine effleuré les fonctionnalités