

ECOLE CENTRALE CASABLANCA



Compte rendu du TP Deep Learning

Réalisé par :
Oussama Lafdil

Encadré par :
Dr. Oumayma Banouar



Table des matières

1	Étape 1 : Préparation et analyse des données	3
1.1	Chargement des données	3
1.2	Analyse de la distribution des classes	3
1.3	Division des données en jeux d'entraînement et de test	3
1.4	Questions de compréhension	4
2	Étape 2 : Conception et entraînement du modèle	4
2.1	Création du modèle	4
2.2	Entraînement du modèle	4
2.3	Évaluation du modèle	4
2.4	Questions de compréhension	5
3	Étape 3 : Entraînement dans un conteneur Docker	5
3.1	Script Python pour l'entraînement et la sauvegarde du modèle	5
3.2	Fichier Dockerfile	6
3.3	Construction et exécution du conteneur Docker	7
3.4	Lien vers l'image Docker	7
3.5	Questions de compréhension	8
4	Étape 4 Déploiement avec Flask dans un conteneur Docker	8
4.1	Développement de l'API Flask	8
4.2	Conteneurisation de l'application Flask	10
4.3	Exécution et test du conteneur	10
4.4	Test de l'API avec l'interface utilisateur	11
4.5	Lien vers l'image Docker sur Docker Hub	11
4.6	Questions de compréhension	12
5	Arborescence finale du projet	12
5.1	Description des fichiers	13
5.2	Conclusion	13

Introduction

Dans ce compte rendu, nous allons détailler les étapes nécessaires pour concevoir, entraîner et déployer un modèle de deep learning dédié à la classification des électrocardiogrammes (ECG). Chaque observation est décrite par 140 caractéristiques, et le modèle prédit deux états possibles pour le patient :

- **0 : ECG normal** (patient sain),
- **1 : ECG anormal** (présence d'une anomalie).

L'objectif de ce travail est de développer une solution complète, allant de l'exploration des données à leur déploiement dans un environnement conteneurisé via Docker. Les étapes suivantes ont été réalisées :

1. **Préparation et analyse des données** : identification de la distribution des classes et création de jeux d'entraînement et de test.
2. **Conception et entraînement d'un modèle de réseau de neurones** : utilisation de Keras et TensorFlow pour la mise en œuvre d'un modèle MLP.
3. **Entraînement dans un conteneur Docker** : garantissant un environnement reproductible pour l'entraînement.
4. **Déploiement avec Flask** : développement d'une API pour rendre le modèle accessible via une interface web.

Ce rapport inclut des analyses des résultats, des codes sources et des captures d'écran des différentes étapes. Les images Docker créées sont également référencées pour permettre la reproductibilité.

1 Étape 1 : Préparation et analyse des données

1.1 Chargement des données

Le fichier `ecg.csv` contenant les données des électrocardiogrammes a été chargé dans l'environnement Python à l'aide de la bibliothèque `pandas`. Voici le code utilisé pour cette étape :

```
1 # Importation des bibliothèques nécessaires
2 import pandas as pd
3
4 # Charger le fichier ecg.csv
5 data = pd.read_csv('ecg.csv')
```

Listing 1 – Chargement des données

Les données contiennent des mesures associées à l'état des patients :

- **0** : ECG normal (patient sain),
- **1** : ECG anormal (présence d'une anomalie).

1.2 Analyse de la distribution des classes

La distribution des classes a été examinée pour vérifier si le dataset est équilibré. Voici le code utilisé pour cette analyse :

```
1 # Examiner la distribution des classes
2 class_distribution = data.iloc[:, -1].value_counts()
3 print("Distribution des classes:")
4 print(class_distribution)
```

Listing 2 – Analyse de la distribution des classes

Les résultats obtenus sont les suivants :

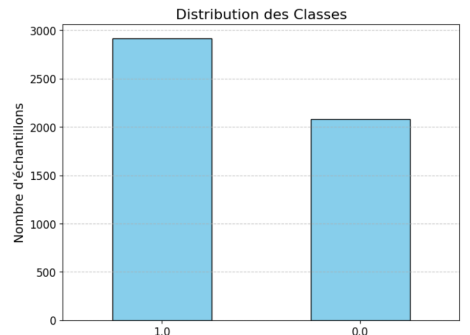


FIGURE 1 – Distribution des classes dans le dataset.

La distribution des classes montre une répartition raisonnablement équilibrée avec 58% pour la classe 1 (ECG anormal) et 42% pour la classe 0 (ECG normal). Cette différence n'est pas critique pour l'entraînement du modèle, car les proportions restent suffisamment proches.

1.3 Division des données en jeux d'entraînement et de test

Les données ont été divisées en jeux d'entraînement (80%) et de test (20%) tout en conservant la distribution des classes grâce au paramètre `stratify`. Voici le code utilisé :

```
1 # Diviser le dataset en jeux d'entraînement et de test
2 from sklearn.model_selection import train_test_split
3
4 X = data.iloc[:, :-1] # Toutes les colonnes sauf la dernière
5 y = data.iloc[:, -1] # La dernière colonne
6
7 X_train, X_test, y_train, y_test = train_test_split(
8     X, y, test_size=0.2, random_state=42, stratify=y)
```

1.4 Questions de compréhension

Pourquoi est-il important de vérifier si le jeu de données est équilibré ?

Un jeu de données déséquilibré peut entraîner un modèle biaisé en faveur de la classe majoritaire, réduisant ainsi sa capacité à bien prédire les observations de la classe minoritaire. En équilibrant ou en prenant en compte ce déséquilibre, nous améliorons la robustesse et l’équité du modèle.

2 Étape 2 : Conception et entraînement du modèle

2.1 Création du modèle

Un réseau de neurones a été conçu en utilisant Keras et TensorFlow. Il comporte :

- Deux couches cachées avec 16 et 8 neurones respectivement, utilisant la fonction d’activation `sigmoid`.
- Une couche de sortie avec un seul neurone et une fonction d’activation `sigmoid` pour produire une sortie binaire.

Le code utilisé pour cette étape est présenté ci-dessous :

```

1 # tape 2 : Conception et entraînement du modèle
2
3 # 1. Créer un réseau de neurones
4 model = Sequential()
5 model.add(Dense(16, input_dim=X_train.shape[1], activation='sigmoid')) # Première
6     couche cachée
7 model.add(Dense(8, activation='sigmoid')) # Deuxième couche cachée
8 model.add(Dense(1, activation='sigmoid')) # Couche de sortie
9
10 # Compilation du modèle
11 model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

```

Listing 4 – Création du modèle

2.2 Entraînement du modèle

Le modèle a été entraîné sur 50 itérations (*epochs*) avec une taille de lot (*batch size*) de 4. Voici le code utilisé :

```

1 # 2. Entraîner le modèle
2 history = model.fit(X_train, y_train, epochs=50, batch_size=4, verbose=1)

```

Listing 5 – Entraînement du modèle

2.3 Évaluation du modèle

Le modèle a été testé sur le jeu de test, et sa précision a été mesurée. Voici le code correspondant :

```

1 # 3. Tester le modèle
2 predictions = model.predict(X_test)
3 predictions = (predictions > 0.5).astype(int) # Conversion en valeurs binaires
4
5 accuracy = accuracy_score(y_test, predictions)
6 print(f"Précision du modèle sur le jeu de test : {accuracy*100:.2f}%")

```

Listing 6 – Évaluation du modèle

Le résultat obtenu est une précision de **99.30%**, comme illustré dans la capture d’écran suivante :

La précision obtenue sur le jeu de test indique que le modèle est capable de prédire correctement l’étiquette (normal ou anormal) dans la grande majorité des cas. Ce résultat suggère une performance élevée du modèle.

32/32 ————— 0s 1ms/step
Précision du modèle sur le jeu de test : 99.30%

FIGURE 2 – Précision du modèle sur le jeu de test.

2.4 Questions de compréhension

Pourquoi utilisons-nous une fonction d'activation non linéaire comme la sigmoïde dans les couches cachées ?

Les fonctions d'activation non linéaires comme la sigmoïde permettent au réseau de modéliser des relations non linéaires entre les données d'entrée et les sorties. Sans elles, le réseau ne pourrait apprendre que des relations linéaires.

Quel est l'effet des paramètres epochs et batch size sur l'entraînement du modèle ?

- **Epochs** : Un nombre élevé d'epochs permet au modèle de s'entraîner plus longtemps, améliorant potentiellement ses performances si les données ne sont pas sur-appriées.
- **Batch size** : Une petite taille de lot permet un entraînement plus précis mais plus lent, tandis qu'une grande taille accélère l'entraînement mais peut être moins précise.

Quelle est la fonction coût utilisée dans le MLP ? Quelle est sa signification ?

La fonction coût utilisée est `binary_crossentropy`, qui mesure la différence entre les probabilités prédites et les vraies étiquettes (0 ou 1). Elle guide l'optimiseur dans l'ajustement des poids pour réduire cette différence et améliorer la précision du modèle.

3 Étape 3 : Entraînement dans un conteneur Docker

3.1 Script Python pour l'entraînement et la sauvegarde du modèle

Un script Python nommé `train_model.py` a été préparé pour charger les données, entraîner le modèle, et le sauvegarder dans un fichier `.pkl` à l'aide de la bibliothèque `pickle`. Voici le code correspondant :

```
1  # Importation des bibliothèques nécessaires
2  import pandas as pd
3  from sklearn.model_selection import train_test_split
4  from tensorflow.keras.models import Sequential
5  from tensorflow.keras.layers import Dense
6  from sklearn.metrics import accuracy_score
7  import pickle
8
9  # Charger le fichier ecg.csv
10 data = pd.read_csv('ecg.csv')
11
12 # Diviser les données en jeux d'entraînement et de test
13 X = data.iloc[:, :-1]
14 y = data.iloc[:, -1]
15
16 # Définir le modèle de réseau de neurones
17 model = Sequential()
18 model.add(Dense(16, input_dim=X.shape[1], activation='sigmoid'))
19 model.add(Dense(8, activation='sigmoid'))
20 model.add(Dense(1, activation='sigmoid'))
21
22 # Compiler le modèle
23 model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
24
25 # Entraîner le modèle
26 model.fit(X, y, epochs=50, batch_size=4, verbose=1)
27
28 # Sauvegarder le modèle avec Pickle
29 with open('ecg_model.pkl', 'wb') as model_file:
30     pickle.dump(model, model_file)
```

Listing 7 – Script Python pour l'entraînement et la sauvegarde du modèle

Le script `train_model.py` est structuré pour répondre aux exigences d'entraînement et de sauvegarde du modèle dans un environnement conteneurisé :

- **Utilisation des bibliothèques essentielles** : Les bibliothèques comme `TensorFlow`, `pandas`, et `pickle` ont été sélectionnées pour leur robustesse et leur compatibilité avec le format requis. Par exemple, `pickle` permet de sauvegarder efficacement le modèle dans un format standardisé pour le déploiement ultérieur.
- **Absence de la division train/test** : Contrairement aux étapes précédentes, le dataset complet est utilisé pour l'entraînement. Cela est justifié dans un cadre d'entraînement final où la validation croisée ou le test n'est pas nécessaire dans le contexte d'un pipeline intégré.
- **Configuration pour Docker** : Le script est conçu pour une exécution autonome dans Docker. Par exemple, la sauvegarde du modèle avec `pickle` garantit que le modèle peut être partagé ou déployé facilement après l'entraînement.

3.2 Fichier Dockerfile

Un fichier `Dockerfile` a été créé pour conteneuriser l'environnement d'entraînement. Ce fichier inclut les bibliothèques nécessaires et configure le conteneur pour exécuter le script Python. Voici son contenu :

```
1 # Utiliser une image Python l g re comme base
2 FROM python:3.9-slim
3
4 # Définir le répertoire de travail dans le conteneur
5 WORKDIR /app
6
7 # Installer les dépendances système nécessaires
8 RUN apt-get update && apt-get install -y --no-install-recommends \
9     build-essential \
10     python3-pip \
11     python3-dev \
12     && apt-get clean \
13     && rm -rf /var/lib/apt/lists/*
14
15 # Installer les bibliothèques Python nécessaires
16 RUN pip install --upgrade pip && \
17     pip install tensorflow pandas scikit-learn pickle5
18
19 # Copier les fichiers du projet dans le conteneur
20 COPY train_model.py /app/
21 COPY ecg.csv /app/
22
23 # Commande par défaut pour exécuter le script Python
24 CMD ["python", "train_model.py"]
```

Listing 8 – Contenu du Dockerfile

Le `Dockerfile` est un élément central du projet, car il permet de construire un environnement isolé et reproductible. Voici une analyse des décisions de conception et leur justification :

- **Base légère (python:3.9-slim)** : L'utilisation d'une image légère réduit la taille de l'image Docker et accélère le déploiement, tout en fournissant les outils essentiels.
- **Installation des dépendances système** : Les packages comme `build-essential` et `python3-dev` sont nécessaires pour compiler certaines bibliothèques Python, comme `TensorFlow`, dans un environnement minimaliste.
- **Gestion des dépendances Python** : En spécifiant les bibliothèques (`tensorflow`, `pandas`, etc.) et en utilisant `pip`, le `Dockerfile` garantit que l'environnement est compatible avec le script Python, même sur une machine où ces bibliothèques ne sont pas installées.
- **Isolation des fichiers du projet** : En définissant un répertoire de travail `/app` et en copiant uniquement les fichiers nécessaires, on minimise les conflits avec d'autres fichiers ou projets.
- **Commande d'exécution par défaut** : Le choix de `CMD ["python", "train_model.py"]` garantit que le conteneur exécute automatiquement le script lors de son lancement, simplifiant ainsi l'utilisation pour l'utilisateur final.

3.3 Construction et exécution du conteneur Docker

Le conteneur a été construit et exécuté à l'aide des commandes suivantes :

```
D:\DL> docker build -f Dockerfile -t lafdil_oussama_env .
```

FIGURE 3 – Construction de l'image Docker.

Cette commande construit une image Docker en utilisant le fichier **Dockerfile** présent dans le répertoire courant. Les éléments suivants sont spécifiés :

- **-f Dockerfile** : Indique le fichier Dockerfile à utiliser pour la construction.
- **-t lafdil_oussama_env** : pour faciliter son identification et son exécution ultérieure.
- **.** : Spécifie que le contexte de construction se trouve dans le répertoire courant.

Une fois cette commande exécutée avec succès, une image Docker contenant toutes les dépendances nécessaires à l'entraînement du modèle est prête à être utilisée.

```
D:\DL> docker run lafdil_oussama_env
```

FIGURE 4 – Exécution du conteneur Docker.

Cette commande exécute un conteneur à partir de l'image Docker précédemment construite. Voici les détails de cette commande :

- **docker run** : Cette commande permet de créer et d'exécuter un conteneur basé sur une image Docker.
- **lafdil_oussama_env** : Le nom de l'image utilisée pour créer le conteneur. Ce nom fait référence à l'image construite lors de la commande précédente (**docker build**).

3.4 Lien vers l'image Docker

L'image Docker construite localement a été taguée et poussée sur Docker Hub pour la rendre accessible et partageable. Voici les commandes utilisées :

```
D:\DL> docker tag lafdil_oussama_env oussamalatardo/lafdil_oussama_env
```

FIGURE 5 – Taguer l'image Docker.

Cette commande permet d'ajouter un nouveau tag (nom ou alias) à une image Docker existante. Les détails sont les suivants :

- **lafdil_oussama_env** : Le nom de l'image Docker locale qui a été créée précédemment avec **docker build**.
- **oussamalatardo/lafdil_oussama_env** : Le nouveau tag attribué à l'image Docker.

L'utilisation de cette commande est une étape préalable à la publication de l'image sur Docker Hub.

```
D:\DL> docker push oussamalatardo/lafdil_oussama_env
```

FIGURE 6 – Pousser l'image Docker sur Docker Hub.

Cette commande permet de pousser l'image Docker taguée localement vers un registre distant, comme Docker Hub. Les détails sont les suivants :

- **oussamalatardo/lafdil_oussama_env** : Le nom complet de l'image Docker incluant le nom d'utilisateur sur Docker Hub et le nom de l'image.

Une fois cette commande exécutée avec succès, l'image Docker sera disponible dans mon espace personnel. Cette image pourra être récupérée et utilisée par d'autres utilisateurs ou systèmes via la commande **docker pull oussamalatardo/lafdil_oussama_env**.

Cette étape finalise le processus de publication de l'image Docker, rendant le projet accessible et partageable via le registre Docker Hub.

L'image est désormais accessible via le lien suivant :

— https://hub.docker.com/r/oussamalatardo/lafdil_oussama_env/

3.5 Questions de compréhension

Quels sont les avantages d'entraîner un modèle directement dans un conteneur Docker ? Docker garantit un environnement isolé et reproductible, éliminant les problèmes liés aux différences entre machines (systèmes d'exploitation, versions des bibliothèques). Cela facilite le partage et le déploiement.

Quel est le rôle de Pickle ?

Pickle est une bibliothèque Python qui permet de sérialiser des objets complexes (comme un modèle de machine learning) pour les sauvegarder dans un fichier et les recharger ultérieurement sans réentraînement.

Comment le fichier Dockerfile permet-il de gérer les dépendances d'un projet ?

Le Dockerfile spécifie toutes les dépendances nécessaires (bibliothèques, versions, etc.) et les installe dans l'image Docker, garantissant que le projet s'exécute dans un environnement stable.

Quel est le rôle du Docker Hub et du Docker Desktop ?

Docker Hub permet de stocker et partager des images Docker en ligne. Docker Desktop est une interface locale pour gérer et exécuter des conteneurs Docker sur une machine.

4 Étape 4 Déploiement avec Flask dans un conteneur Docker

4.1 Développement de l'API Flask

Une API Flask a été développée pour permettre de charger le modèle sauvegardé (`ecg_model.pkl`) et de faire des prédictions sur de nouvelles données ECG. Voici le code du fichier `app.py` :

```
1 from flask import Flask, request, render_template_string, jsonify
2 import pickle
3 import numpy as np
4 import json
5
6 # Load the saved model
7 with open('ecg_model.pkl', 'rb') as f:
8     model = pickle.load(f)
9
10 # Initialize Flask app
11 app = Flask(__name__)
12
13 # HTML Template for Upload Form
14 UPLOAD_HTML = """
15 <!DOCTYPE html>
16 <html lang="en">
17 <head>
18     <meta charset="UTF-8">
19     <meta name="viewport" content="width=device-width, initial-scale=1.0">
20     <title>ECG Prediction</title>
21     <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha3/dist/css/bootstrap
22     .min.css" rel="stylesheet">
23 </head>
24 <body>
25     <div class="container mt-5">
26         <h1 class="text-center">ECG Prediction</h1>
27         <p class="text-muted text-center">Upload a JSON file containing the features
28         for prediction</p>
29         <form method="POST" action="/predict" enctype="multipart/form-data">
30             <div class="mb-3">
31                 <label for="json_file" class="form-label">Upload JSON File</label>
32                 <input type="file" class="form-control" id="json_file" name="json_file"
33                 " accept=".json" required>
34             </div>
35             <button type="submit" class="btn btn-primary">Submit</button>
36         </form>
37     </div>
38 </body>
39 </html>
40 """
```

```

34     </div>
35 </body>
36 </html>
37 """
38
39 # HTML Template for Result Page
40 RESULT_HTML = """
41 <!DOCTYPE html>
42 <html lang="en">
43 <head>
44     <meta charset="UTF-8">
45     <meta name="viewport" content="width=device-width, initial-scale=1.0">
46     <title>Prediction Result</title>
47     <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha3/dist/css/bootstrap
      .min.css" rel="stylesheet">
48 </head>
49 <body>
50     <div class="container mt-5 text-center">
51         <h1>Prediction Result</h1>
52         <p class="display-4">Prediction: <strong>{{ prediction }}</strong></p>
53         <p class="display-6">{{ interpretation }}</p>
54         <a href="/" class="btn btn-primary mt-3">Try Again</a>
55     </div>
56 </body>
57 </html>
58 """
59
60 # Routes
61 @app.route('/')
62 def home():
63     """Render the file upload form."""
64     return render_template_string(UPLOAD_HTML)
65
66 @app.route('/predict', methods=['POST'])
67 def predict():
68     """Handle prediction requests from a JSON file."""
69     try:
70         # Check if a JSON file is uploaded
71         json_file = request.files.get('json_file')
72         if not json_file:
73             return jsonify({'error': 'No JSON file uploaded'}), 400
74
75         # Load and validate JSON file
76         json_data = json.load(json_file)
77         if "features" not in json_data or len(json_data["features"]) != 140:
78             return jsonify({'error': 'JSON file must contain a "features" key with
              exactly 140 values'}), 400
79
80         # Extract features from JSON
81         features = np.array(json_data["features"]).reshape(1, -1)
82
83         # Make a prediction
84         prediction = model.predict(features)[0][0]
85         rounded_prediction = round(prediction)
86
87         # Interpret the prediction
88         if rounded_prediction == 0:
89             interpretation = "ECG normal (patient is a)"
90         elif rounded_prediction == 1:
91             interpretation = "ECG abnormal (presence of an anomaly)"
92         else:
93             interpretation = "Prediction unclear"
94
95         return render_template_string(RERESULT_HTML, prediction=rounded_prediction,
          interpretation=interpretation)
96     except Exception as e:
97         return jsonify({'error': str(e)}), 400
98
99
100 if __name__ == '__main__':
101     app.run(host='0.0.0.0', port=3000, debug=True)

```

Listing 9 – Code de l’API Flask

Ce script Python est une application web développée avec Flask pour effectuer des prédictions à partir d’un modèle de machine learning pré-entraîné. Voici les principales parties du code :

- **Chargement du modèle** : Le modèle de machine learning sauvegardé (`ecg_model.pkl`) est chargé à l’aide de la bibliothèque `pickle`.
- **Initialisation de Flask** : L’application web est initialisée avec Flask pour gérer les requêtes HTTP et servir des pages web.
- **Templates HTML** :
 - `UPLOAD_HTML` : Fournit un formulaire permettant aux utilisateurs de charger un fichier JSON contenant les données ECG.
 - `RESULT_HTML` : Affiche le résultat de la prédiction.
- **Routes Flask** :
 - `/` : Affiche le formulaire pour permettre le téléchargement d’un fichier JSON.
 - `/predict` : Traite le fichier JSON téléchargé, effectue une prédiction avec le modèle, et renvoie le résultat avec une interprétation.
- **Gestion des erreurs** : Si le fichier JSON est invalide ou mal formaté, un message d’erreur clair est retourné à l’utilisateur.
- **Exécution de l’application** : L’application s’exécute sur le port 3000 et est accessible via n’importe quelle interface réseau (0.0.0.0).

4.2 Conteneurisation de l’application Flask

Pour exécuter l’API dans un environnement isolé et reproductible, un fichier Dockerfile (`a.Dockerfile`) a été créé pour conteneuriser l’application Flask. Voici le contenu du Dockerfile :

```
1 # Utiliser une image Python l g re comme base
2 FROM python:3.9-slim
3
4 # Définir le répertoire de travail
5 WORKDIR /app
6
7 # Installer les dépendances système
8 RUN apt-get update && apt-get install -y --no-install-recommends \
9     build-essential \
10     python3-pip \
11     && apt-get clean \
12     && rm -rf /var/lib/apt/lists/*
13
14 # Installer les bibliothèques Python nécessaires
15 RUN pip install --upgrade pip && \
16     pip install flask tensorflow pandas pickle5 numpy
17
18 # Copier les fichiers du projet
19 COPY app.py /app/
20 COPY ecg_model.pkl /app/
21
22 # Exposer le port utilisé par Flask
23 EXPOSE 3000
24
25 # Commande pour exécuter l’application Flask
26 CMD ["python", "app.py"]
```

Listing 10 – Fichier a.Dockerfile

4.3 Exécution et test du conteneur

Le conteneur a été construit, exécuté, et l’API a été testée en envoyant des requêtes avec des données ECG. Voici les commandes utilisées :

```
D:\DL> docker build -f a.Dockerfile -t lafdil_oussama_dep .
```

FIGURE 7 – Construction de l'image Docker pour l'application Flask.

```
D:\DL> docker run --rm -p 3000:3000 lafdil_oussama_dep
```

FIGURE 8 – Exécution du conteneur Docker pour l'application Flask.

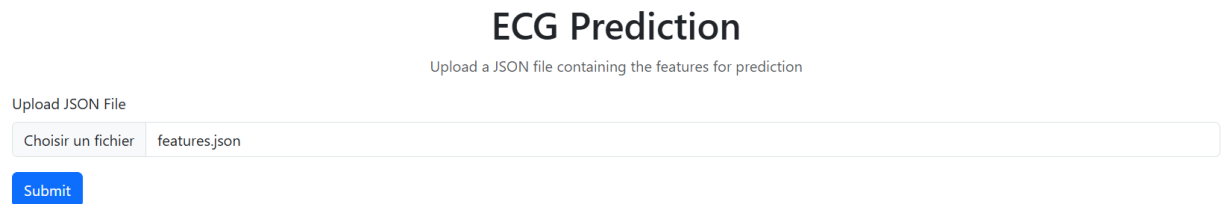
4.4 Test de l'API avec l'interface utilisateur

L'API a été testée à l'aide de l'interface utilisateur développée avec Flask et Bootstrap. Cette interface permet à l'utilisateur de charger un fichier JSON contenant les données nécessaires pour effectuer une prédiction.

Les étapes du test sont les suivantes :

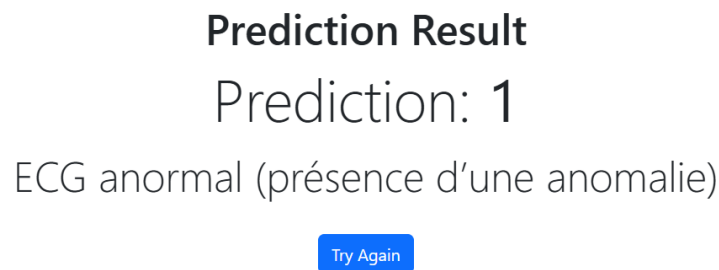
- L'utilisateur charge un fichier JSON contenant les caractéristiques (**features**) via le formulaire disponible sur l'interface.
- L'application effectue une prédiction à l'aide du modèle pré-entraîné.
- Le résultat est affiché sur une page dédiée, avec une interprétation claire de la prédiction (ECG normal ou anormal).

Les captures d'écran suivantes illustrent le test :



The screenshot shows a web interface titled "ECG Prediction" with the subtitle "Upload a JSON file containing the features for prediction". Below this, there is a section labeled "Upload JSON File" containing a file input field with the placeholder text "Choisir un fichier" and the filename "features.json" displayed. To the right of the input field is a blue "Submit" button.

FIGURE 9 – Interface utilisateur pour charger un fichier JSON.



The screenshot shows a web page titled "Prediction Result" with the text "Prediction: 1" in a large font. Below this, it says "ECG anormal (présence d'une anomalie)". At the bottom of the page, there is a blue button labeled "Try Again".

FIGURE 10 – Résultat de la prédiction affiché sur l'interface utilisateur.

4.5 Lien vers l'image Docker sur Docker Hub

L'image Docker construite localement a été taguée et poussée sur Docker Hub pour la rendre accessible et partageable. Voici les commandes utilisées :

```
D:\DL> docker tag lafdil_oussama_dep oussamalatardo/lafdil_oussama_dep:latest
```

FIGURE 11 – Taguer l'image Docker pour Docker Hub.

```
D:\DL> docker push oussamalatardo/lafdil_oussama_dep
```

FIGURE 12 – Pousser l'image Docker sur Docker Hub.

L'image Docker est désormais disponible sur Docker Hub via le lien suivant :
— https://hub.docker.com/r/oussamalatardo/lafdil_oussama_dep

4.6 Questions de compréhension

Comment une API Flask permet-elle de rendre un modèle de deep learning accessible ?

Une API Flask expose des routes HTTP permettant de communiquer avec le modèle via des requêtes. Par exemple, une requête `POST` contenant les données d'entrée peut déclencher une prédiction par le modèle, et le résultat est renvoyé au client sous forme de réponse JSON. Cela permet d'intégrer le modèle dans n'importe quelle application.

Quelle est la signification de RESTful API ?

Une API RESTful respecte les principes REST (Representational State Transfer), en utilisant des méthodes HTTP standard (`GET`, `POST`, etc.) et des formats de données comme JSON pour échanger des informations de manière stateless.

Pourquoi Docker est-il utile pour déployer une application web basée sur un modèle de deep learning ?

Docker permet de déployer une application avec toutes ses dépendances dans un conteneur isolé et reproductible, évitant les problèmes de compatibilité entre environnements.

Que signifie CI/CD dans le cadre d'un projet deep learning ? Quelle relation avec Docker ?

CI/CD (Intégration Continue / Déploiement Continu) automatise les tests, le déploiement et les mises à jour du projet. Docker facilite le CI/CD en offrant des conteneurs standardisés, garantissant que les applications fonctionnent de manière cohérente sur les différents systèmes.

5 Arborescence finale du projet

L'arborescence finale du projet est représentée dans la capture d'écran ci-dessous :

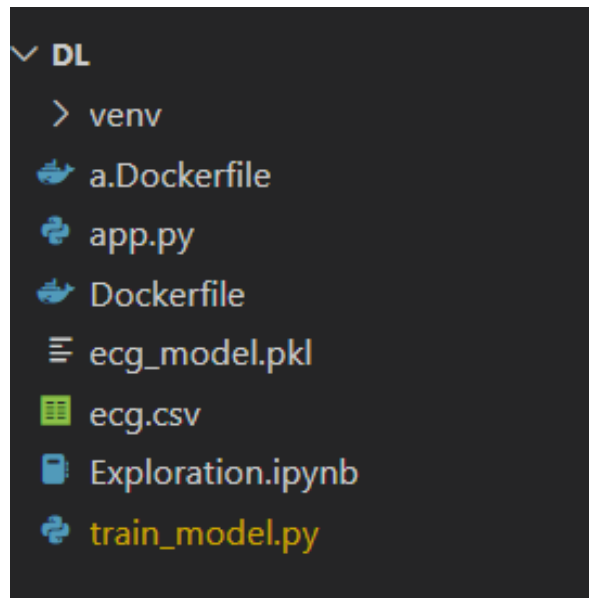


FIGURE 13 – Arborescence finale du projet.

5.1 Description des fichiers

- **a.Dockerfile** : Contient les instructions pour conteneuriser l'application Flask et exécuter l'API.
- **app.py** : Script Python contenant le code de l'API Flask. Ce fichier charge le modèle sauvegardé et expose une route HTTP pour effectuer des prédictions.
- **Dockerfile** : Fichier Docker permettant de conteneuriser le script d'entraînement du modèle.
- **ecg_model.pkl** : Fichier contenant le modèle de machine learning sauvegardé après entraînement, au format `pickle`.
- **ecg.csv** : Jeu de données utilisé pour entraîner le modèle.
- **Exploration.ipynb** : Notebook Google Colab utilisé pour effectuer une exploration initiale des données et entraîner un modèle simple.
- **train_model.py** : Script Python contenant le code pour charger les données, entraîner le modèle, et le sauvegarder. Ce script est conçu pour être exécuté dans un conteneur Docker.
- **venv/** : Répertoire contenant l'environnement virtuel Python utilisé pour exécuter le projet en local.

5.2 Conclusion

Cette arborescence est structurée pour séparer les étapes d'exploration, d'entraînement, et de déploiement du modèle. Chaque fichier a un rôle spécifique et contribue à la reproductibilité et au bon fonctionnement du projet.

Conclusion

Ce projet a permis de mettre en œuvre une solution end-to-end pour la classification des ECG, de l'analyse initiale des données au déploiement du modèle via une API Flask conteneurisée. Les étapes clés réalisées ont démontré l'efficacité et la flexibilité des outils modernes comme TensorFlow, Flask, et Docker. Voici les principaux résultats obtenus :

- Un modèle avec une précision de **99,3 %** sur le jeu de test, validant sa capacité à détecter les anomalies ECG.
- Une interface utilisateur intuitive permettant d'effectuer des prédictions en temps réel.
- Une infrastructure conteneurisée assurant portabilité et reproductibilité, essentielle dans des contextes industriels.

Les approches suivies et les résultats obtenus montrent la puissance des technologies modernes pour résoudre des problèmes complexes dans le domaine de la santé. Ce travail pourrait être étendu par des optimisations du modèle et une validation sur des jeux de données plus diversifiés.