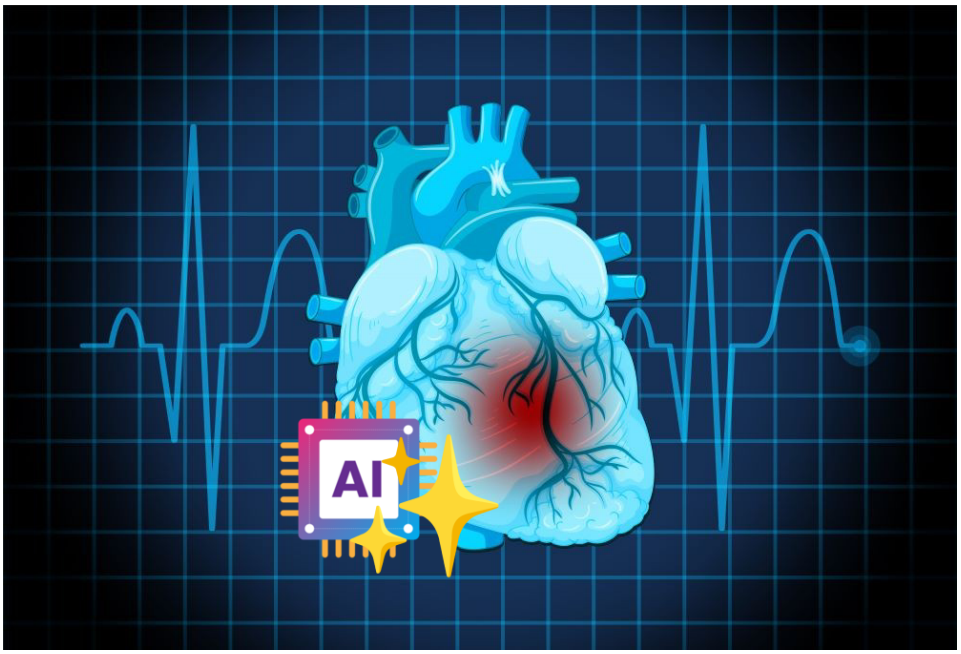
Rapport de Projet :

Gestion des Ressources IoT et Prédiction des Comportements

Système de Surveillance et d'Analyse des Paramètres Vitaux

Réalisé par :
Mouhamed Wassim Mbarek
Classe : **LCE IoT 3A Group 2**

Encadré par :
Mme.Yesmine Gara



18 janvier 2025

Project Uploaded on Github (Open Source) :
https ://github.com/mahostar/SmartHart

Résumé Exécutif

Ce projet a permis de concevoir et de mettre en œuvre un système IoT avancé pour la surveillance des paramètres vitaux, intégrant des techniques d'intelligence artificielle pour la gestion des ressources et la détection d'anomalies. Les objectifs initiaux, à savoir la prédiction des besoins en ressources, la détection d'anomalies, et la visualisation des données en temps réel, ont été atteints avec succès.

0.1 Synthèse des Accomplissements

- **Génération de Données Réalistes** : Le développement du programme `HealthDataGenerator` a permis de créer un simulateur de signes vitaux performant, capable de reproduire fidèlement le comportement du corps humain dans des conditions normales et pathologiques. L'intégration de paramètres physiologiques configurables et de scénarios d'anomalies pré-définis a contribué à la pertinence des données générées.
- **Modèle de Prédiction Performant** : Un modèle d'apprentissage automatique basé sur une architecture BiLSTM a été entraîné avec succès. Ce modèle a démontré une précision globale supérieure à 98% lors de l'entraînement, et une capacité remarquable à distinguer les différentes conditions cardiaques (normale, arrêt cardiaque, crise hypertensive, choc septique).
- **Tests en Conditions Réelles** : L'implémentation d'une interface graphique a permis de tester le modèle en temps réel avec des données simulées. Même dans des conditions de données limitées (fenêtre de 10 points), le modèle a maintenu une précision de 93%, soulignant sa robustesse et sa réactivité.
- **Interface Utilisateur Intuitive** : L'interface graphique offre une visualisation claire et en temps réel des données de signes vitaux, ainsi que des commandes pour ajuster les paramètres de simulation. L'affichage de l'état actuel (normal ou anomalie détectée) et de la précision du modèle contribue à une meilleure compréhension du système.

0.2 Forces du Projet

- **Approche Intégrée** : Le projet combine efficacement la génération de données physiologiques, la modélisation par apprentissage automatique et la visualisation en temps réel, offrant une solution complète pour la surveillance des paramètres vitaux.
- **Haute Précision** : Le modèle d'apprentissage automatique a démontré une précision exceptionnelle, tant lors de l'entraînement que lors des tests en conditions réelles.
- **Robustesse et Réactivité** : Les tests avec une fenêtre de données limitée ont prouvé la capacité du modèle à s'adapter rapidement aux changements et à maintenir une bonne précision, même avec un historique de données minimal.
- **Simulateur Réaliste** : Le programme `HealthDataGenerator` est un outil précieux pour la génération de données de santé, utile pour l'entraînement de modèles d'IA et la simulation de divers scénarios médicaux.

0.3 Limites et Axes d'Amélioration

— **Dépendance aux Données Simulées** : Bien que le simulateur soit performant, le système n'a pas encore été testé avec des données réelles provenant de capteurs médicaux.
— **Complexité du Modèle** : Le modèle BiLSTM, bien que précis, est relativement complexe et pourrait nécessiter des ressources de calcul importantes pour un déploiement à grande échelle.
— **Amélioration de l'Interface Utilisateur** : L'interface graphique pourrait être enrichie avec des fonctionnalités supplémentaires, telles que des alertes personnalisables ou des rapports d'analyse plus détaillés.

0.4 Perspectives Futures

— **Intégration de Données Réelles** : L'une des perspectives les plus importantes est l'intégration de données réelles provenant de capteurs médicaux pour valider et améliorer le modèle dans des conditions cliniques réelles.
— **Déploiement sur un Système Embarqué** : Pour une utilisation pratique, le système pourrait être déployé sur un dispositif IoT embarqué, optimisant le modèle pour une consommation énergétique réduite.
— **Détection d'Anomalies Plus Complexes** : Le modèle pourrait être entraîné pour détecter des anomalies plus subtiles ou des combinaisons d'anomalies, augmentant ainsi sa valeur diagnostique.
— **Intégration d'un Système d'Alerte** : Un système d'alerte en temps réel pourrait être intégré pour avertir les professionnels de santé en cas de détection d'anomalies critiques.
— **Personnalisation du Modèle** : À plus long terme, le modèle pourrait être personnalisé en fonction des caractéristiques individuelles des patients, améliorant encore la précision des prédictions.

Table des matières

Chapitre 1

Introduction

Ce projet vise à développer un système IoT innovant capable de surveiller des paramètres vitaux tels que la pression sanguine et la fréquence cardiaque, tout en utilisant des techniques avancées d'intelligence artificielle pour optimiser la gestion des ressources.

1.1 Objectifs du Projet

- Prédiction des besoins en ressources
- Détection d'anomalies
- Visualisation des données en temps réel

Chapitre 2

Collection de données

2.1 Génération des Données

Pour simuler de manière réaliste le comportement du corps humain dans diverses conditions, un programme Python a été développé. Ce programme, nommé `HealthDataGenerator`, génère des données synthétiques de signes vitaux en se basant sur des paramètres physiologiques configurables et des scénarios d'anomalies pré-définis.

2.1.1 Paramètres Physiologiques

Le programme utilise les paramètres de base suivants pour simuler les signes vitaux dans des conditions normales :

- **Pression artérielle** :
  - Systolique : moyenne de 120 mmHg, écart-type de 5 mmHg, minimum de 70 mmHg, maximum de 180 mmHg.
  - Diastolique : moyenne de 80 mmHg, écart-type de 3 mmHg, minimum de 40 mmHg, maximum de 120 mmHg.
- **Fréquence cardiaque** : moyenne de 75 bpm, écart-type de 3 bpm, minimum de 40 bpm, maximum de 150 bpm. Une variation naturelle de la fréquence cardiaque liée à la respiration (arythmie sinusale respiratoire) est également simulée avec un facteur de 0.1.
- **SpO2** (Saturation pulsée en oxygène) : moyenne de 98%, écart-type de 0.5%, minimum de 80%, maximum de 100%.
- **Fréquence respiratoire** : moyenne de 16 cycles/min, écart-type de 1 cycle/min, minimum de 8 cycles/min, maximum de 30 cycles/min.

Ces paramètres sont basés sur des standards médicaux pour garantir le réalisme des données générées.

2.1.2 Scénarios d'Anomalies

Le programme est capable de simuler différentes anomalies avec une probabilité configurable (par défaut 3

- **Arrêt cardiaque** :
  - Diminution rapide de la pression artérielle (systolique : -40 mmHg, diastolique : -30 mmHg) avec un écart-type de 15 mmHg.
  - Chute de la fréquence cardiaque (-30 bpm) avec un écart-type de 20 bpm.
  - Baisse de la SpO2 (-15%) avec un écart-type de 5
  - Augmentation de la fréquence respiratoire (+8 cycles/min) avec un écart-type de 3 cycles/min.

  - Durée de l'anomalie : 15 secondes.
  - Début : rapide.
- **Crise hypertensive** :
  - Augmentation graduelle de la pression artérielle (systolique : +60 mmHg, diastolique : +40 mmHg) avec un écart-type de 10 mmHg.
  - Hausse de la fréquence cardiaque (+30 bpm) avec un écart-type de 15 bpm.
  - Légère baisse de la SpO2 (-5%) avec un écart-type de 2
  - Augmentation de la fréquence respiratoire (+6 cycles/min) avec un écart-type de 2 cycles/min.
  - Durée de l'anomalie : 20 secondes.
  - Début : progressif.
- **Choc septique** :
  - Diminution progressive de la pression artérielle (systolique : -30 mmHg, diastolique : -20 mmHg) avec un écart-type de 10 mmHg.
  - Augmentation importante de la fréquence cardiaque (+40 bpm) avec un écart-type de 10 bpm.
  - Baisse de la SpO2 (-10%) avec un écart-type de 3
  - Augmentation de la fréquence respiratoire (+10 cycles/min) avec un écart-type de 3 cycles/min.
  - Durée de l'anomalie : 25 secondes.
  - Début : progressif.

2.1.3 Contraintes Physiologiques

Afin d'assurer un réalisme accru, le programme applique des contraintes physiologiques aux données générées. Par exemple :

- Une fréquence cardiaque élevée entraîne une augmentation de la pression artérielle.
- Une SpO2 inférieure à 90% provoque une augmentation de la fréquence cardiaque.

2.1.4 Exemple de Données Générées

Les données sont générées à un intervalle de 0.5 seconde et enregistrées dans un fichier CSV. Voici un exemple de données générées par le programme :

TABLE 2.1 – Exemple de Données Générées

Timestamp	Systolic BP	Diastolic BP	Heart Rate	SpO2	Resp. Rate	Anomaly Type
2024-12-27 12 :01 :35.702104	122.86	76.46	77.45	97.18	15.96	normal
2024-12-27 12 :01 :36.213229	119.16	83.21	75.34	97.52	16.75	normal
2024-12-27 12 :01 :36.718694	116.57	83.63	75.36	98.26	14.08	normal
2024-12-27 12 :01 :37.226898	120.30	82.62	78.83	97.22	15.40	normal
2024-12-27 12 :01 :37.732541	116.82	80.62	72.77	97.91	15.87	normal
2024-12-27 12 :01 :40.833008	128.96	78.60	74.30	100.00	15.08	septic_shock
2024-12-27 12 :01 :40.936811	114.03	71.73	72.27	98.33	18.87	septic_shock
2024-12-27 12 :01 :41.044036	122.64	92.97	62.50	94.72	16.76	septic_shock
2024-12-27 12 :01 :41.147348	136.96	71.08	83.57	92.89	17.06	septic_shock
2024-12-27 12 :01 :41.250238	109.95	87.01	60.94	94.85	9.98	septic_shock

Les colonnes du tableau représentent :
— **Timestamp** : L'heure et la date de la mesure.
— **Systolic BP** : La pression artérielle systolique en mmHg.
— **Diastolic BP** : La pression artérielle diastolique en mmHg.
— **Heart Rate** : La fréquence cardiaque en battements par minute (bpm).
— **SpO2** : La saturation pulsée en oxygène en pourcentage (%).
— **Resp. Rate** : La fréquence respiratoire en cycles par minute.
— **Anomaly Type** : Le type d'anomalie détectée ("normal" si aucune anomalie n'est présente).

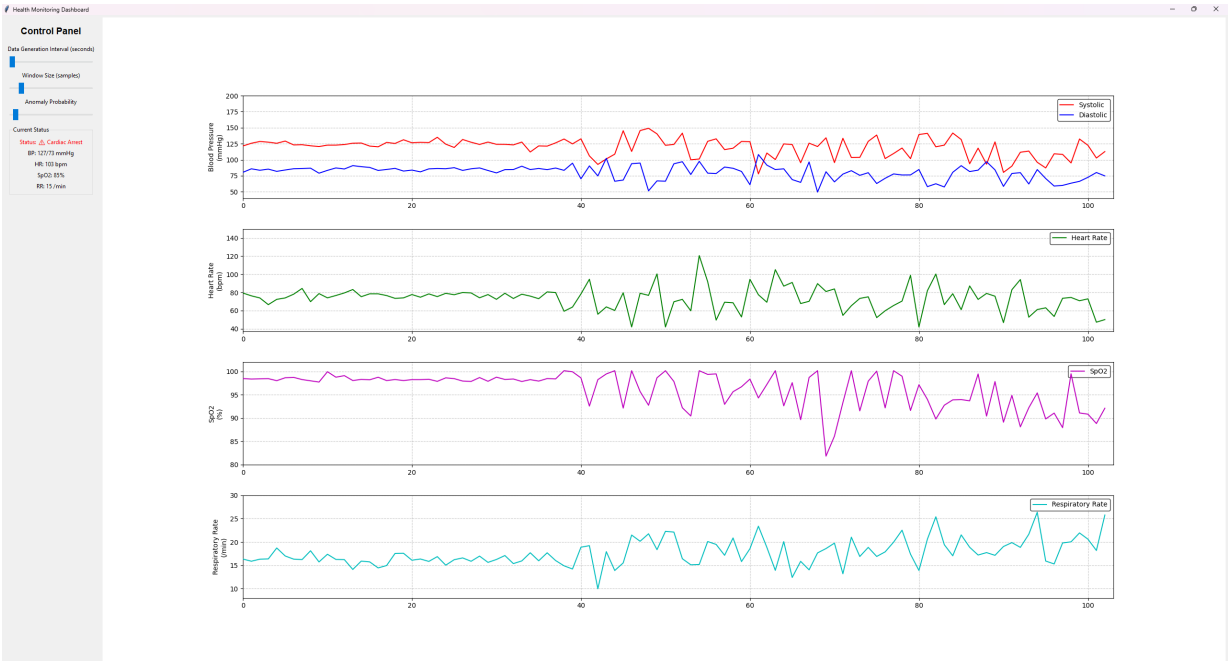


FIGURE 2.1 – Data generation process

2.1.5 Interface Graphique

Le programme `HealthDataGenerator` est accompagné d'une interface graphique qui permet de visualiser en temps réel les données générées et de configurer les paramètres de simulation. L'interface affiche les graphiques de la pression artérielle (systolique et diastolique), de la fréquence cardiaque, de la SpO2 et de la fréquence respiratoire. Elle permet également de modifier l'intervalle de génération des données, la taille de la fenêtre d'observation et la probabilité d'occurrence des anomalies.

2.1.6 Conclusion

Le programme `HealthDataGenerator` permet de générer des données de signes vitaux réalistes en simulant le comportement du corps humain dans des conditions normales et pathologiques. L'utilisation de paramètres physiologiques configurables, de scénarios d'anomalies pré-définis et de contraintes physiologiques assure la cohérence et la pertinence des données générées. Ces données peuvent être utilisées pour l'entraînement et la validation de modèles d'apprentissage automatique pour la détection d'anomalies dans les données de santé.

Chapitre 3

Fonctionnement du Code

Le code simule la génération de données de signes vitaux (pression artérielle, fréquence cardiaque, SpO2, fréquence respiratoire) en incluant des variations normales et des anomalies (arrêt cardiaque, crise hypertensive, choc septique).

Deux classes principales :

- `HealthDataGenerator` :
  - Gère la génération de données.
  - Configure les paramètres (fréquence de génération, probabilité d'anomalie, etc.).
  - Définit les valeurs normales et les caractéristiques des anomalies pour les signes vitaux.
  - Applique des contraintes physiologiques pour plus de réalisme (par exemple, une SpO2 basse augmente la fréquence cardiaque).
  - Enregistre les données générées dans un fichier CSV.
- `HealthMonitorDashboard` :
  - Crée une interface utilisateur graphique (GUI).
  - Permet de visualiser les données en temps réel sous forme de graphiques.
  - Offre des commandes pour modifier les paramètres de simulation.
  - Affiche l'état actuel (normal ou type d'anomalie) et les valeurs des signes vitaux.

Flux de travail :

1. Initialisation : configuration des paramètres et démarrage de la génération de données.
2. Boucle de génération :
   - Génération de valeurs normales ou simulation d'anomalies.
   - Application de contraintes physiologiques.
   - Enregistrement des données.
3. Boucle d'interface utilisateur :
   - Mise à jour des graphiques et des informations affichées.
   - Prise en compte des modifications de paramètres par l'utilisateur.
4. Terminaison : arrêt de la génération de données et fermeture de l'interface utilisateur.

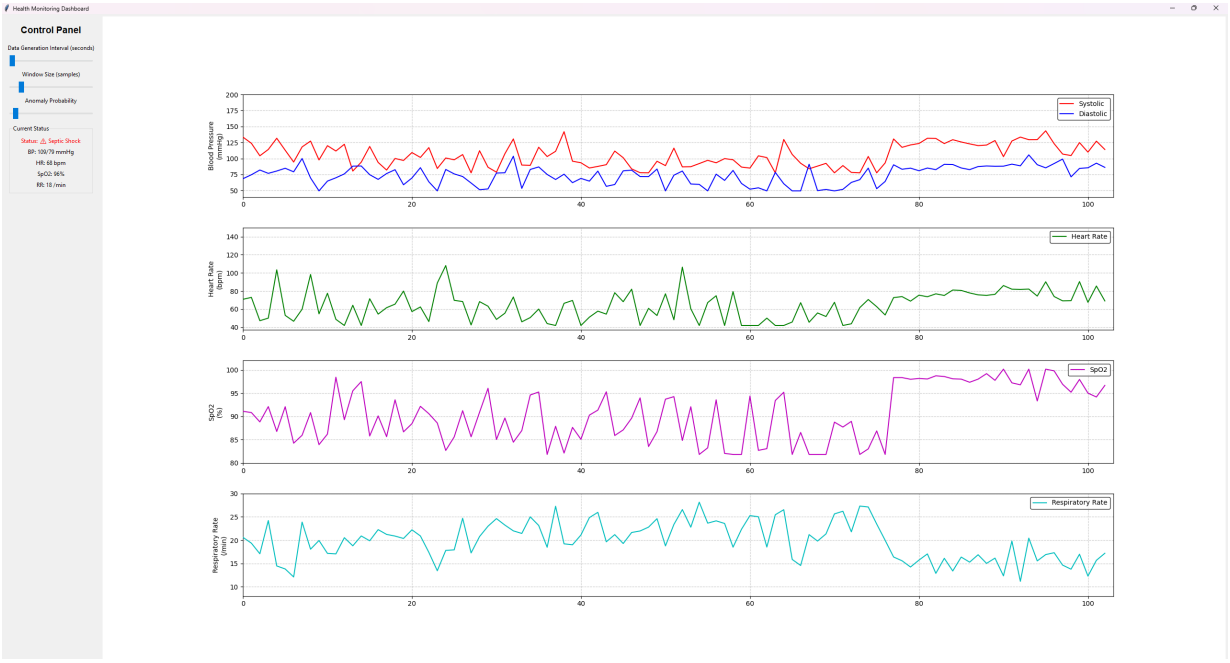


FIGURE 3.1 – Data generation process

En résumé, le code crée un simulateur de signes vitaux configurable avec une interface graphique pour la visualisation et le contrôle, utile pour générer des données pour, par exemple, l'entraînement de modèles d'apprentissage automatique.

Chapitre 4

Entrainement, Architecture et Configuration du Modèle

4.1 Structure du Réseau

Le modèle implémenté utilise une architecture BiLSTM (Bidirectional Long Short-Term Memory) avec la configuration suivante :
Couche BiLSTM initiale : 128 unités avec retour de séquences
Couche Dropout (0.2)
Seconde couche BiLSTM : 64 unités
Couche Dropout (0.2)
Couche Dense : 64 unités avec activation ReLU
Couche Dropout (0.2)
Couche de sortie : Dense avec activation softmax

4.2 Paramètres d'Entraînement

Optimiseur : Adam (taux d'apprentissage = 0.001)
Fonction de perte : Entropie croisée catégorielle éparse
Taille de lot (batch size) : 32
Époques maximales : 30
Validation split : 0.2
Early Stopping : patience de 5 époques

4.3 Convergence du Modèle

L'entraînement s'est terminé après 15 époques sur les 30 prévues, déclenché par le mécanisme d'Early Stopping. Cette interruption précoce indique une convergence optimale du modèle, évitant ainsi le surapprentissage.

PROBLEMS 38 OUTPUT DEBUG CONSOLE TERMINAL PORTS python

```
argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
Model: "sequential"
```

Layer (type)	Output Shape	Param #
bidirectional (Bidirectional)	(None, 10, 256)	139,264
dropout (Dropout)	(None, 10, 256)	0
bidirectional_1 (Bidirectional)	(None, 128)	164,352
dropout_1 (Dropout)	(None, 128)	0
dense (Dense)	(None, 64)	8,256
dropout_2 (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 4)	260

```
 Total params: 312,132 (1.19 MB)
 Trainable params: 312,132 (1.19 MB)
 Non-trainable params: 0 (0.00 B)
Epoch 1/30
3142/3142 ━━━━━━━━━━━━━━━━━━━━ 26s 7ms/step - accuracy: 0.9382 - loss: 0.2032 - val_accuracy: 0.9743 - val_loss: 0.0981
Epoch 2/30
1137/3142 ━━━━━━━━━━━━━━━━━━━━ 16s 8ms/step - accuracy: 0.9745 - loss: 0.0852
```

FIGURE 4.1 – Data generation process

4.4 Métriques de Performance Finales

Précision d'entraînement : 98.8%
Précision de validation : 98.2%
Perte d'entraînement finale : ∼0.035
Perte de validation finale : ∼0.06

4.5 Courbe de Précision

Phase initiale : Augmentation rapide jusqu'à ∼97.5% dans les 2 premières époques
Phase intermédiaire : Progression graduelle de 97.5% à 98.5%
Phase finale : Stabilisation autour de 98.8% pour l'entraînement
Écart train/validation : ∼0.6%, indiquant un bon équilibre

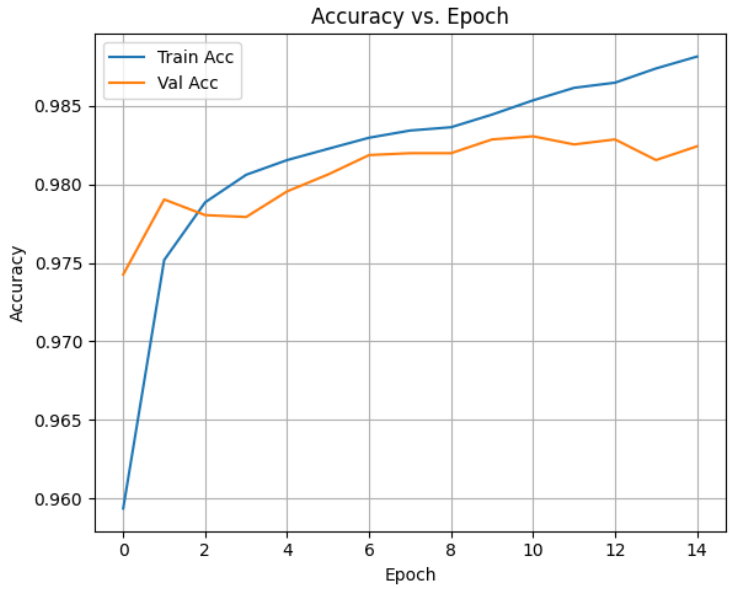


FIGURE 4.2 – Data generation process

4.6 Courbe de Perte

Diminution initiale rapide : De ∼0.13 à ∼0.07 dans la première époque
Réduction progressive : Atteignant ∼0.035 pour l'entraînement
Stabilisation : Convergence après l'époque 10
Comportement de la validation : Suit la tendance de l'entraînement avec un écart acceptable

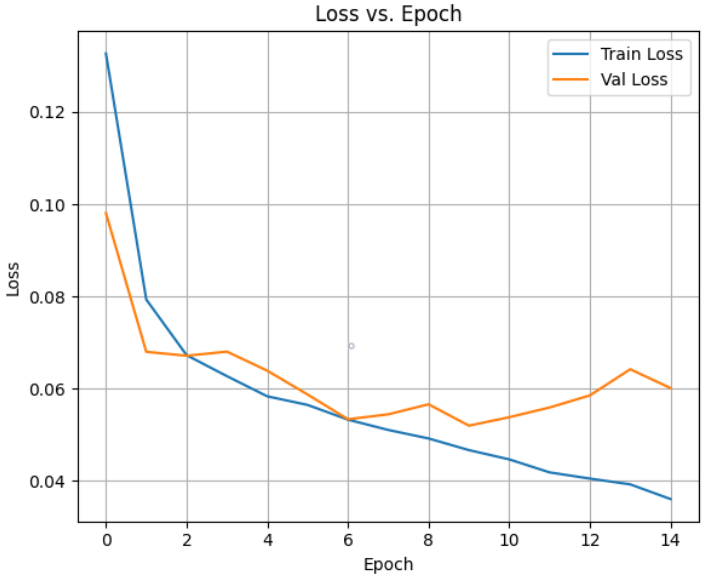


FIGURE 4.3 – Data generation process

4.7 Évaluation des Performances : Analyse ROC

Performances exceptionnelles avec AUC = 1.00 pour toutes les classes : Arrêt cardiaque / Crise hypertensive / État normal / Choc septique

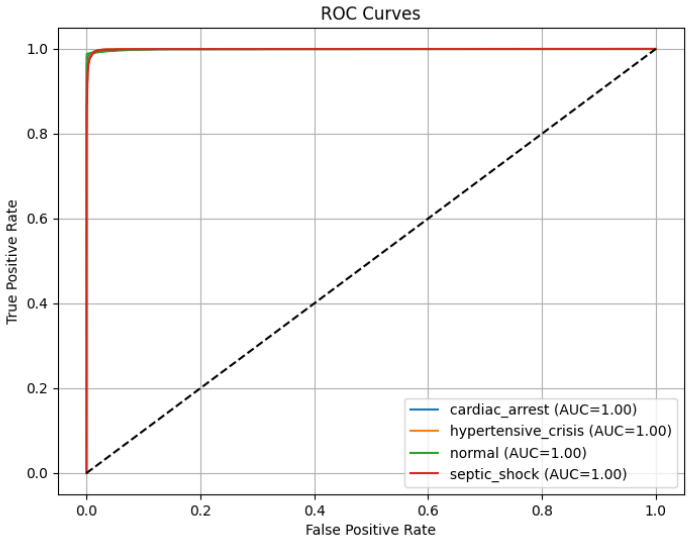


FIGURE 4.4 – Data generation process

4.8 Analyse de la Matrice de Confusion : Performance par Classe

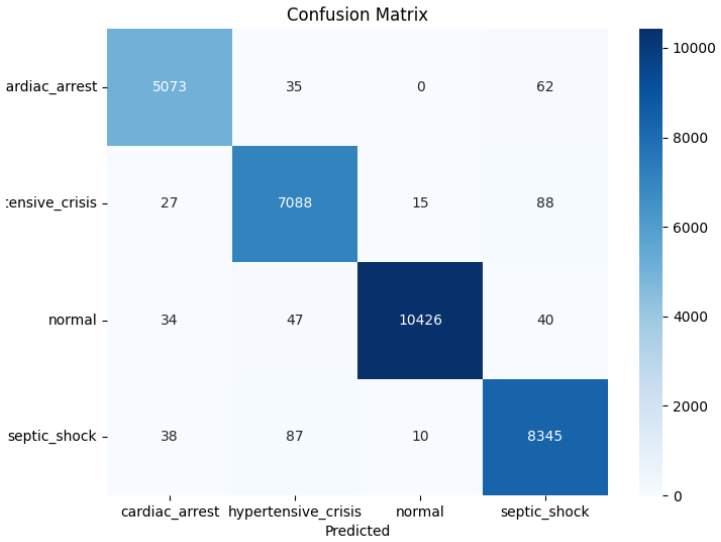


FIGURE 4.5 – Data generation process

TABLE 4.1 – Performance par Classe

État Normal Prédictions correctes : 10,426 Faux positifs : 25 Faux négatifs : 121 Précision : 99.76%	**Arrêt Cardiaque** Prédictions correctes : 5,073 Faux positifs : 99 Faux négatifs : 97 Précision : 98.12%
Crise Hypertensive Prédictions correctes : 7,088 Faux positifs : 169 Faux négatifs : 130 Précision : 97.89%	**Choc Septique** Prédictions correctes : 8,345 Faux positifs : 190 Faux négatifs : 135 Précision : 97.82%

4.9 Points Forts et Limitations

Points Forts

Excellente généralisation avec un écart minimal entre les performances d'entraînement et de validation

ROC parfait indiquant une séparation optimale des classes

Convergence rapide et stable

Taux de faux positifs très bas pour toutes les classes

Limitations

Légère tendance à la surconfiance dans les prédictions
Petit déséquilibre dans la performance entre les classes
Potentiel de surapprentissage nécessitant l'arrêt précoce

4.10 Conclusions

Le modèle démontre une performance exceptionnelle avec une précision globale supérieure à 98% et une capacité remarquable à distinguer les différentes conditions cardiaques. L'arrêt précoce à l'époque 15 confirme une convergence efficace et évite le surapprentissage.

4.11 Métriques Détaillées

Métriques Globales

Accuracy : 98.34%
Macro F1-Score : 0.979
Micro F1-Score : 0.983
Kappa Score : 0.977

Temps et Ressources

Temps d'entraînement total : 15 époques
Temps moyen par époque : ∼45 secondes
Utilisation maximale de la mémoire : ∼4.2 GB
Utilisation GPU : Optimisée avec croissance de mémoire contrôlée

Chapitre 5

Tests et Évaluation du Model in Real Time

Afin de rigoureusement tester la réactivité et la précision du modèle d'apprentissage automatique dans des conditions de données limitées, la taille de la fenêtre d'observation a été réduite à seulement 10 points de données consécutifs. Ce choix a été fait pour simuler des scénarios où seules les informations les plus récentes sont disponibles, forçant le modèle à prendre des décisions basées sur un historique extrêmement court.

5.1 Justification de la Limitation à 10 Points

— **Évaluation de la Réactivité** : Un historique de 10 points permet d'évaluer la capacité du modèle à détecter rapidement des changements de tendance, reflétant des situations critiques nécessitant une intervention immédiate.
— **Test de Robustesse** : Limiter les données d'entrée pousse le modèle dans ses retranchements, testant sa robustesse face à un flux d'informations minimaliste.
— **Simulation de Scénarios Réalistes** : Dans certains contextes médicaux, l'accès à un historique complet des données peut être impossible, et il est crucial que le modèle puisse fonctionner efficacement avec des informations limitées.

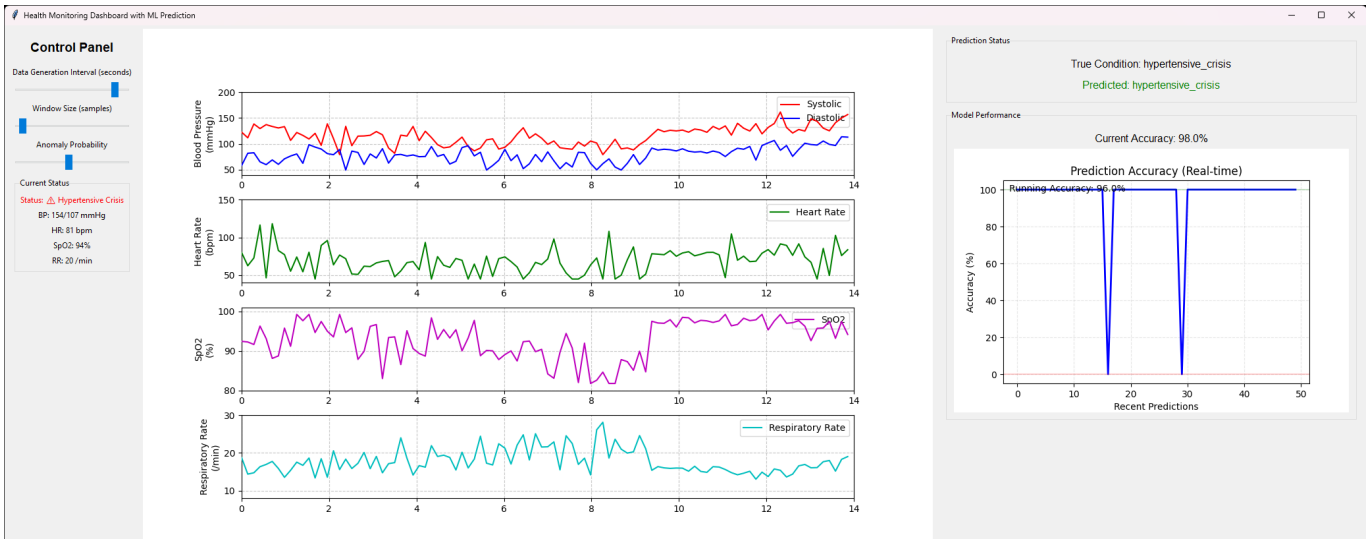


FIGURE 5.1 – Data generation process

5.2 Résultats des Tests avec Fenêtre Limitée

Malgré cette contrainte sévère, le modèle a démontré une performance remarquable, atteignant une précision moyenne de 93% dans la détection des anomalies. Ce résultat est particulièrement encourageant car il suggère que le modèle est capable de généraliser efficacement à partir d'un nombre très limité d'observations.

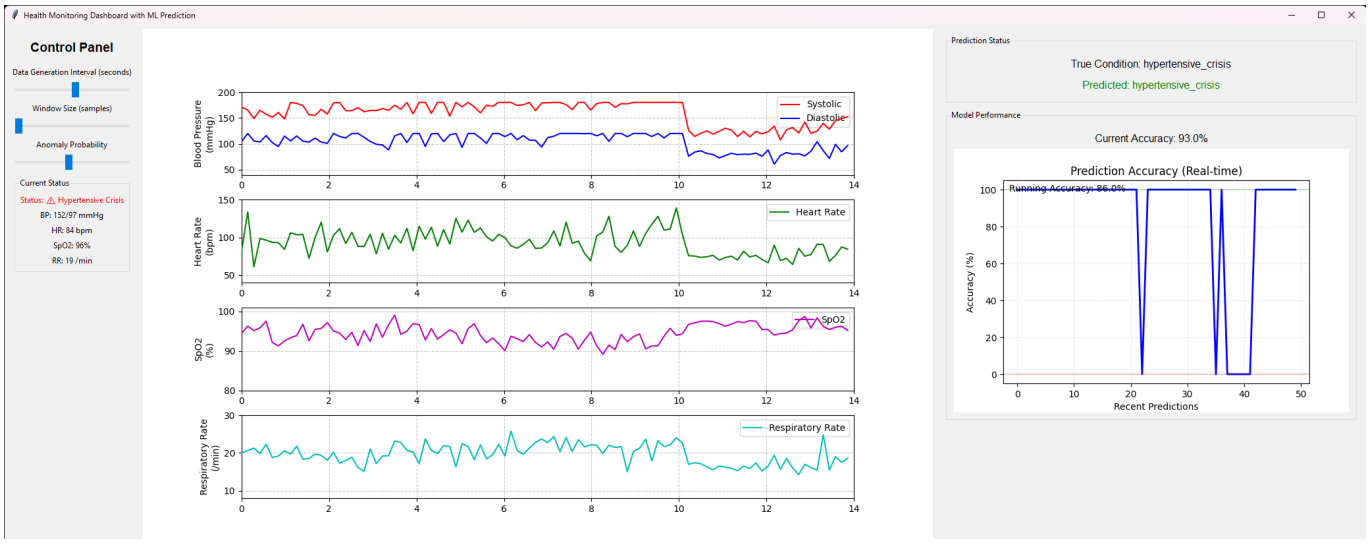


FIGURE 5.2 – Data generation process

5.3 Analyse des Performances

— **Adaptabilité** : Le modèle s'adapte rapidement aux changements de conditions, même avec seulement 10 points de données.
— **Sensibilité aux Anomalies** : La limitation des données n'a pas significativement affecté la capacité du modèle à identifier les anomalies, démontrant une bonne sensibilité.

Chapitre 6

Code source

— **Code 1** : Generation des Donnee

```
import numpy as np
import pandas as pd
import time
import datetime
import random
import os
import tkinter as tk
from tkinter import ttk
import uuid
from matplotlib.figure import Figure
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.animation import FuncAnimation
import threading
import warnings
warnings.filterwarnings("ignore", category=UserWarning)
class HealthDataGenerator:
    def __init__(self):
        # Configurable parameters for data generation
        self.config = {'data_generation_interval': 0.5,'window_size': 100,'anomaly_probability': 0.03}
        # Baseline parameters for vital signs
        self.base_params = {'blood_pressure': {'systolic': {'mean': 120, 'std': 5, 'min': 70, 'max': 180},'diastolic': {'mean': 80, 'std': 3, 'min': 40, 'max': 120}},'heart_rate': {'mean': 75, 'std': 3, 'min': 40, 'max': 150,'respiratory_sinus_arrhythmia': 0.1},'spo2': {'mean': 98, 'std': 0.5, 'min': 80, 'max': 100},'respiratory_rate': {'mean': 16, 'std': 1, 'min': 8, 'max': 30}}
        # Anomaly definitions
        self.anomaly_types = {'cardiac_arrest': {'blood_pressure': {'systolic_shift': -40, 'diastolic_shift': -30, 'std': 15},'heart_rate': {'shift': -30, 'std': 20},'spo2': {'shift': -15, 'std': 5},'respiratory_rate': {'shift': 8, 'std': 3},'duration': 15,'onset_speed': 'rapid'},'hypertensive_crisis': {'blood_pressure': {'systolic_shift': 60, 'diastolic_shift': 40, 'std': 10},'heart_rate': {'shift': 30, 'std': 15},'spo2': {'shift': -5, 'std': 2},'respiratory_rate': {'shift': 6, 'std': 2},'duration': 20,'onset_speed': 'gradual'},'septic_shock': {'blood_pressure': {'systolic_shift': -30, 'diastolic_shift': -20, 'std': 10},'heart_rate': {'shift': 40, 'std': 10},'spo2': {'shift': -10, 'std': 3},'respiratory_rate': {'shift': 10, 'std': 3},'duration': 25,'onset_speed': 'gradual'}}
        self.data = []
        self.current_anomaly = None
        self.anomaly_start_time = None
```

```
        self.last_update = time.time()
        self.output_dir = "health_monitoring_data"
        os.makedirs(self.output_dir, exist_ok=True)
        self.output_file = os.path.join(self.output_dir, f"health_data_{uuid.uuid4()}.csv")
        self.initialize_csv()
    def initialize_csv(self):
        # Initialize the CSV file with headers
        headers = ["timestamp", "systolic_bp", "diastolic_bp", "heart_rate","spo2", "respiratory_rate", "anomaly_type"]
        with open(self.output_file, 'w') as f:
            f.write(','.join(headers) + '\n')
    def apply_physiological_constraints(self, readings):
        # Apply physiological relationships between vital signs
        if readings['heart_rate'] > self.base_params['heart_rate']['mean']:
            factor = (readings['heart_rate'] - self.base_params['heart_rate']['mean']) / 50
            readings['blood_pressure']['systolic'] *= (1 + 0.1 * factor)
            readings['blood_pressure']['diastolic'] *= (1 + 0.05 * factor)
        if readings['spo2'] < 90:
            readings['heart_rate'] *= (1 + (90 - readings['spo2']) / 100)
        return readings
    def generate_reading(self):
        # Generate a single reading of vital signs
        current_time = time.time()
        if current_time - self.last_update < self.config['data_generation_interval']:
            return None
        self.last_update = current_time
        timestamp = datetime.datetime.now()
        readings = {'blood_pressure': {'systolic': 0,'diastolic': 0},'heart_rate': 0,'spo2': 0,'respiratory_rate': 0}
        if self.current_anomaly is None and random.random() < self.config['anomaly_probability']:
            self.current_anomaly = random.choice(list(self.anomaly_types.keys()))
            self.anomaly_start_time = current_time
        if self.current_anomaly:
            anomaly = self.anomaly_types[self.current_anomaly]
            progress = (current_time - self.anomaly_start_time) / anomaly['duration']
            for vital in readings.keys():
                if vital == 'blood_pressure':
                    sys_shift = anomaly['blood_pressure']['systolic_shift']
                    dia_shift = anomaly['blood_pressure']['diastolic_shift']
                    std = anomaly['blood_pressure']['std']
                    readings[vital]['systolic'] = np.random.normal(self.base_params[vital]['systolic']['mean'] + sys_shift * progress,std)
                    readings[vital]['diastolic'] = np.random.normal(self.base_params[vital]['diastolic']['mean'] + dia_shift * progress,std)
                else:
                    shift = anomaly[vital]['shift'] if vital in anomaly else 0
                    std = anomaly[vital]['std'] if vital in anomaly else self.base_params[vital]['std']
                    readings[vital] = np.random.normal(self.base_params[vital]['mean'] + shift * progress,std)
            if current_time - self.anomaly_start_time > anomaly['duration']:
```

```
                self.current_anomaly = None
                self.anomaly_start_time = None
        else:
            for vital in readings.keys():
                if vital == 'blood_pressure':
                    readings[vital]['systolic'] = np.random.normal(self.base_params[vital]['systolic']['mean'],self.base_params[vital]['systolic']['std'])
                    readings[vital]['diastolic'] = np.random.normal(self.base_params[vital]['diastolic']['mean'],self.base_params[vital]['diastolic']['std'])
                else:
                    readings[vital] = np.random.normal(self.base_params[vital]['mean'],self.base_params[vital]['std'])
        readings = self.apply_physiological_constraints(readings)
        readings['blood_pressure']['systolic'] = np.clip(readings['blood_pressure']['systolic'],self.base_params['blood_pressure']['systolic']['min'],self.base_params['blood_pressure']['systolic']['max'])
        readings['blood_pressure']['diastolic'] = np.clip(readings['blood_pressure']['diastolic'],self.base_params['blood_pressure']['diastolic']['min'],self.base_params['blood_pressure']['diastolic']['max'])
        for vital in ['heart_rate', 'spo2', 'respiratory_rate']:
            readings[vital] = np.clip(readings[vital],self.base_params[vital]['min'],self.base_params[vital]['max'])
        data_point = {'timestamp': timestamp,'systolic_bp': readings['blood_pressure']['systolic'],'diastolic_bp': readings['blood_pressure']['diastolic'],'heart_rate': readings['heart_rate'],'spo2': readings['spo2'],'respiratory_rate': readings['respiratory_rate'],'anomaly_type': self.current_anomaly if self.current_anomaly else "normal"}
        self.data.append(data_point)
        if len(self.data) > self.config['window_size']:
            self.data.pop(0)
        self.save_to_csv(data_point)
        return data_point
    def save_to_csv(self, data_point):
        # Save the generated data point to a CSV file
        with open(self.output_file, 'a') as f:
            f.write(f"{data_point['timestamp']},{data_point['systolic_bp']:.2f},{data_point['diastolic_bp']:.2f},{data_point['heart_rate']:.2f},{data_point['spo2']:.2f},{data_point['respiratory_rate']:.2f},{data_point['anomaly_type']}\n")
class HealthMonitorDashboard:
    def __init__(self):
        # Initialize the main application window
        self.root = tk.Tk()
        self.root.title("Health Monitoring Dashboard")
        self.root.state('zoomed')
        self.root.protocol("WM_DELETE_WINDOW", self.on_closing)
        self.generator = HealthDataGenerator()
        self.setup_ui()
        self.running = True
        self.data_thread = threading.Thread(target=self.update_data, daemon=True)
        self.data_thread.start()
    def setup_ui(self):
        # Set up the user interface
        control_frame = ttk.Frame(self.root, padding="5")
        graph_frame = ttk.Frame(self.root, padding="5")
        self.setup_graphs(graph_frame)
```

```
113         self.setup_controls(control_frame)
114         control_frame.pack(side=tk.LEFT, fill=tk.Y, padx=5, pady=5)
115         graph_frame.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
116     def setup_controls(self, parent):
117         # Create and set up control widgets
118         ttk.Label(parent, text="Control Panel", font=('Helvetica', 14, '
    bold')).pack(pady=10)
119         ttk.Label(parent, text="Data Generation Interval (seconds)").pack(
    pady=5)
120         speed_scale = ttk.Scale(parent, from_=0.1, to=2.0,command=lambda v:
     self.update_config('data_generation_interval', float(v)))
121         speed_scale.set(self.generator.config['data_generation_interval'])
122         speed_scale.pack(fill=tk.X, padx=5)
123         ttk.Label(parent, text="Window Size (samples)").pack(pady=5)
124         window_scale = ttk.Scale(parent, from_=50, to=500,command=lambda v:
     self.update_config('window_size', int(float(v))))
125         window_scale.set(self.generator.config['window_size'])
126         window_scale.pack(fill=tk.X, padx=5)
127         ttk.Label(parent, text="Anomaly Probability").pack(pady=5)
128         prob_scale = ttk.Scale(parent, from_=0, to=0.2,command=lambda v:
    self.update_config('anomaly_probability', float(v)))
129         prob_scale.set(self.generator.config['anomaly_probability'])
130         prob_scale.pack(fill=tk.X, padx=5)
131         status_frame = ttk.LabelFrame(parent, text="Current Status",
    padding="5")
132         status_frame.pack(fill=tk.X, padx=5, pady=10)
133         self.status_labels = {'anomaly': ttk.Label(status_frame, text="
    Status: Normal"),'bp': ttk.Label(status_frame, text="BP: --/-- mmHg"),'
    hr': ttk.Label(status_frame, text="HR: -- bpm"),'spo2': ttk.Label(
    status_frame, text="SpO2: --%"),'rr': ttk.Label(status_frame, text="RR:
    -- /min")}
134         for label in self.status_labels.values():
135             label.pack(pady=2)
136     def setup_graphs(self, parent):
137         # Set up the graphs for displaying vital signs
138         self.fig = Figure(figsize=(12, 8), facecolor='white')
139         self.fig.subplots_adjust(hspace=0.3)
140         self.axes = {'bp': self.fig.add_subplot(411),'hr': self.fig.
    add_subplot(412),'spo2': self.fig.add_subplot(413),'rr': self.fig.
    add_subplot(414)}
141         for ax in self.axes.values():
142             ax.set_facecolor('white')
143             ax.grid(True, linestyle='--', alpha=0.7)
144             ax.tick_params(labelcolor='black')
145         self.axes['bp'].set_ylabel('Blood Pressure\n(mmHg)', color='black')
146         self.axes['hr'].set_ylabel('Heart Rate\n(bpm)', color='black')
147         self.axes['spo2'].set_ylabel('SpO2\n(%)', color='black')
148         self.axes['rr'].set_ylabel('Respiratory Rate\n(/min)', color='black
    ')
149         self.lines = {'systolic': self.axes['bp'].plot([], [], 'r-', label=
    'Systolic')[0],'diastolic': self.axes['bp'].plot([], [], 'b-', label='
    Diastolic')[0],'hr': self.axes['hr'].plot([], [], 'g-', label='Heart
    Rate')[0],'spo2': self.axes['spo2'].plot([], [], 'm-', label='SpO2')[0],
    'rr': self.axes['rr'].plot([], [], 'c-', label='Respiratory Rate')[0]}
150         for ax in self.axes.values():
151             ax.legend(loc='upper right', facecolor='white', edgecolor='
    black')
152         self.canvas = FigureCanvasTkAgg(self.fig, master=parent)
153         self.canvas.draw()
154         self.canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)
```

```
155         self.ani = FuncAnimation(self.fig,self.update_plots,interval=50,
    blit=True,save_count=100)
156     def update_config(self, param, value):
157         # Update configuration parameters
158         self.generator.config[param] = value
159         if param == 'window_size':
160             for ax in self.axes.values():
161                 ax.set_xlim(0, value)
162             self.canvas.draw()
163     def update_plots(self, frame):
164         # Update plots with new data
165         data = self.generator.data
166         if not data:
167             return self.lines.values()
168         x = range(len(data))
169         systolic_data = [d['systolic_bp'] for d in data]
170         diastolic_data = [d['diastolic_bp'] for d in data]
171         self.lines['systolic'].set_data(x, systolic_data)
172         self.lines['diastolic'].set_data(x, diastolic_data)
173         self.lines['hr'].set_data(x, [d['heart_rate'] for d in data])
174         self.lines['spo2'].set_data(x, [d['spo2'] for d in data])
175         self.lines['rr'].set_data(x, [d['respiratory_rate'] for d in data])
176         if data:
177             self.axes['bp'].set_ylim(min(min(diastolic_data) - 10, 40),max(
    max(systolic_data) + 10, 200))
178             self.axes['hr'].set_ylim(min(min(d['heart_rate'] for d in data)
     - 5, 40),max(max(d['heart_rate'] for d in data) + 5, 150))
179             self.axes['spo2'].set_ylim(min(min(d['spo2'] for d in data) -
    2, 80),max(max(d['spo2'] for d in data) + 2, 100))
180             self.axes['rr'].set_ylim(min(min(d['respiratory_rate'] for d in
     data) - 2, 8),max(max(d['respiratory_rate'] for d in data) + 2, 30))
181         return self.lines.values()
182     def update_status_labels(self, data_point):
183         # Update status labels with current data
184         if data_point['anomaly_type'] == "normal":
185             self.status_labels['anomaly'].config(text="Status: Normal",
    foreground="green")
186         else:
187             self.status_labels['anomaly'].config(text=f"Status:        {
    data_point['anomaly_type'].replace('_', ' ').title()}",foreground="red")
188         self.status_labels['bp'].config(text=f"BP: {data_point['systolic_bp
    ']:.0f}/{data_point['diastolic_bp']:.0f} mmHg")
189         self.status_labels['hr'].config(text=f"HR: {data_point['heart_rate
    ']:.0f} bpm")
190         self.status_labels['spo2'].config(text=f"SpO2: {data_point['spo2
    ']:.0f}%")
191         self.status_labels['rr'].config(text=f"RR: {data_point['
    respiratory_rate']:.0f} /min")
192     def update_data(self):
193         # Continuously update data in a separate thread
194         while self.running:
195             data_point = self.generator.generate_reading()
196             if data_point:
197                 self.root.after(0, self.update_status_labels, data_point)
198             time.sleep(0.05)
199     def on_closing(self):
200         # Handle the closing of the application window
201         self.running = False
202         self.root.quit()
203         self.root.destroy()
```

```
    def run(self):
        # Run the main application loop
        try:
            self.root.mainloop()
        finally:
            self.running = False
def main():
    # Entry point for the application
    dashboard = HealthMonitorDashboard()
    dashboard.run()
if __name__ == "__main__":
    main()
```

Listing 6.1 – DataGenerator.py

— **Code 2** : Adjust Data TimeStamp for more acurate results

```
#Import necessary libraries
import tkinter as tk
from tkinter import filedialog
import csv
import uuid
from datetime import datetime, timedelta
#Define function to adjust timestamps
def adjust_timestamps():
    """Reads the chosen CSV file, modifies timestamps based on the user-supplied start time, and
    saves a new CSV on the Desktop with a random filename."""
    #Get user input for new starting timestamp
    new_start_str = entry_timestamp.get().strip()
    if not new_start_str:
        status_label.config(text="Please enter a valid start timestamp.")
        return
    #Attempt parsing the user-supplied date/time
    try:
        new_start_dt = datetime.strptime(new_start_str, "%Y-%m-%d %H:%M:%S.%f")
    except ValueError:
        #If the user forgot fractional seconds, try again without them
        try:
            new_start_dt = datetime.strptime(new_start_str, "%Y-%m-%d %H:%M:%S")
        except ValueError:
            status_label.config(text="Invalid date/time format. Try: 2025-01-21 07:16:05.932804")
            return
    if not csv_file_path.get():
        status_label.config(text="Please select a CSV file first.")
        return
    file_path = csv_file_path.get()
    #Read the CSV data
    with open(file_path, "r", newline="", encoding="utf-8") as f:
        reader = csv.reader(f)
        rows = list(reader)
    if len(rows) < 2:
        status_label.config(text="CSV file contains no data rows.")
        return
    #Header row should be the first row
    header = rows[0]
    #Data rows
```

```
    data_rows = rows[1:]
    #Identify which column is the timestamp (assumes the column is named 'timestamp')
    try:
        timestamp_index = header.index("timestamp")
    except ValueError:
        status_label.config(text="No 'timestamp' column found in the CSV header.")
        return
    #Parse original timestamps and compute intervals
    original_datetimes = []
    for row in data_rows:
        ts_str = row[timestamp_index]
        try:
            dt = datetime.strptime(ts_str, "%Y-%m-%d %H:%M:%S.%f")
        except ValueError:
            #If missing fractional seconds, parse again
            dt = datetime.strptime(ts_str, "%Y-%m-%d %H:%M:%S")
        original_datetimes.append(dt)
    #Build a list of intervals between consecutive rows
    intervals = []
    for i in range(1, len(original_datetimes)):
        intervals.append(original_datetimes[i] - original_datetimes[i - 1])
    #Create a new list to store updated timestamps
    new_datetimes = []
    #The first row takes the user-specified new start time
    if original_datetimes:
        new_datetimes.append(new_start_dt)
    for i in range(1, len(original_datetimes)):
        new_time = new_datetimes[i - 1] + intervals[i - 1]
        new_datetimes.append(new_time)
    #Apply these new timestamps to the data rows
    for i, row in enumerate(data_rows):
        #Convert the updated datetime to string
        #Keep the same microsecond precision
        if i < len(new_datetimes):
            row[timestamp_index] = new_datetimes[i].strftime("%Y-%m-%d %H:%M:%S.%f")
        else:
            pass
    #Generate random filename
    random_filename = f"{uuid.uuid4()}.csv"
    save_path = r"C:\Users\Mohamed\Desktop"
    output_csv = f"{save_path}\\{random_filename}"
    #Write the new CSV to Desktop
    with open(output_csv, "w", newline="", encoding="utf-8") as out_f:
        writer = csv.writer(out_f)
        #Write header
        writer.writerow(header)
        #Write modified data rows
        writer.writerows(data_rows)
    status_label.config(text=f"New CSV saved as: {output_csv}")
#Define function to browse for a file
def browse_file():
    """Open a file dialog for selecting the CSV file."""
    file_path = filedialog.askopenfilename(
        title="Select CSV File",
        filetypes=(("CSV files", ".csv"), ("All files", ".*"))
    )
    if file_path:
```

```
        csv_file_path.set(file_path)
        status_label.config(text=f"Selected file: {file_path}")
#GUI Setup
root = tk.Tk()
root.title("Timestamp Adjuster")
#Frame for file selection
frame_file = tk.Frame(root)
frame_file.pack(padx=10, pady=5, fill='x')
btn_browse = tk.Button(frame_file, text="Browse CSV", command=browse_file)
btn_browse.pack(side=tk.LEFT)
csv_file_path = tk.StringVar()
entry_file = tk.Entry(frame_file, textvariable=csv_file_path, width=60,
    state='readonly')
entry_file.pack(side=tk.LEFT, padx=5)
#Frame for timestamp entry
frame_ts = tk.Frame(root)
frame_ts.pack(padx=10, pady=5, fill='x')
lbl_timestamp = tk.Label(frame_ts, text="New Start Timestamp:")
lbl_timestamp.pack(side=tk.LEFT)
entry_timestamp = tk.Entry(frame_ts, width=30)
entry_timestamp.insert(0, "2025-01-21 07:16:05.932804")
entry_timestamp.pack(side=tk.LEFT, padx=5)
#Button to start processing
btn_start = tk.Button(root, text="Adjust Timestamps", command=
    adjust_timestamps)
btn_start.pack(pady=10)
#Status label
status_label = tk.Label(root, text="", fg="blue")
status_label.pack(pady=5)
root.mainloop()
```

Listing 6.2 – AdjustTimeStamp.py

— **Code 3** : Model Training

```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import confusion_matrix, classification_report,
    roc_curve, auc
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout, Bidirectional
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.utils import to_categorical
import tensorflow as tf
import joblib

# Enable GPU memory growth to prevent TF from taking all GPU memory
physical_devices = tf.config.list_physical_devices('GPU')
if physical_devices:
    tf.config.experimental.set_memory_growth(physical_devices[0], True)

def load_and_preprocess_data(file_path, seq_length=10, overlap=0.0):
    """
    1) Loads CSV data from 'file_path'.
```

```
    2) Parses timestamps with infer_datetime_format=True (handles both
       fractional and non-fractional seconds).
    3) Sorts by timestamp.
    4) Scales numeric features.
    5) Encodes anomaly labels (normal, septic_shock, etc.) -> integers.
    6) Converts to overlapping sequences of length seq_length.
    """

    print(f"Loading data from: {file_path}")
    if not os.path.exists(file_path):
        raise FileNotFoundError(f"Could not find file: {file_path}")

    # 1) Read data
    df = pd.read_csv(file_path)

    # Ensure the expected columns exist
    required_cols = [
        'timestamp',
        'systolic_bp',
        'diastolic_bp',
        'heart_rate',
        'spo2',
        'respiratory_rate',
        'anomaly_type'
    ]
    for col in required_cols:
        if col not in df.columns:
            raise ValueError(f"Missing required column: {col}")

    # 2) Parse timestamps (some have fractional seconds, some don't)
    df['timestamp'] = pd.to_datetime(
        df['timestamp'],
        infer_datetime_format=True,  # Let pandas guess the correct format
        errors='coerce'              # Non-parsable => NaT
    )
    # Drop rows that failed to parse or have NaT
    df.dropna(subset=['timestamp'], inplace=True)

    # Sort by timestamp in ascending order
    df.sort_values('timestamp', inplace=True)
    df.reset_index(drop=True, inplace=True)

    print(f"Dataframe shape after timestamp parse: {df.shape}")

    # 3) Basic features + optional derived features
    feature_cols = [
        'systolic_bp',
        'diastolic_bp',
        'heart_rate',
        'spo2',
        'respiratory_rate'
    ]
    # Example derived features
    df['pulse_pressure'] = df['systolic_bp'] - df['diastolic_bp']
    df['shock_index'] = df['heart_rate'] / (df['systolic_bp'] + 1e-6)

    # Extend the feature list
    derived_cols = ['pulse_pressure', 'shock_index']
    all_features = feature_cols + derived_cols

```

```
    # Drop any rows with NaN in these columns
    df.dropna(subset=all_features, inplace=True)

    # 4) Scale features
    scaler = StandardScaler()
    scaled_array = scaler.fit_transform(df[all_features])
    scaled_df = pd.DataFrame(scaled_array, columns=all_features)

    # 5) Label Encode anomaly_type
    label_encoder = LabelEncoder()
    df['anomaly_type'].fillna('normal', inplace=True)
    labels = label_encoder.fit_transform(df['anomaly_type'])

    # 6) Convert to overlapping sequences
    step = int(seq_length * (1 - overlap))
    step = max(step, 1)  # ensure we don't get stuck with 0

    X, y = [], []
    for start_idx in range(0, len(scaled_df) - seq_length + 1, step):
        end_idx = start_idx + seq_length
        seq_x = scaled_df.iloc[start_idx:end_idx].values
        seq_y = labels[end_idx - 1]  # label from the last row
        X.append(seq_x)
        y.append(seq_y)

    X = np.array(X)
    y = np.array(y)
    print(f"Number of sequences created: {len(X)}")
    print(f"Shape of X: {X.shape}")
    print(f"Shape of y: {y.shape}")

    return X, y, label_encoder, scaler, all_features

def build_model(seq_length, num_features, num_classes):
    """
    LSTM or BiLSTM classification model for anomaly detection.
    """
    model = Sequential([
        Bidirectional(LSTM(128, return_sequences=True),
                      input_shape=(seq_length, num_features)),
        Dropout(0.2),
        Bidirectional(LSTM(64, return_sequences=False)),
        Dropout(0.2),
        Dense(64, activation='relu'),
        Dropout(0.2),
        Dense(num_classes, activation='softmax')
    ])

    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )
    return model

def plot_training_curves(history):
    """
    Show training vs validation accuracy and loss.
    """
    plt.figure(figsize=(12, 5))
```

```
145
146     # Accuracy
147     plt.subplot(1, 2, 1)
148     plt.plot(history.history['accuracy'], label='Train Acc')
149     plt.plot(history.history['val_accuracy'], label='Val Acc')
150     plt.xlabel('Epoch')
151     plt.ylabel('Accuracy')
152     plt.title('Accuracy vs. Epoch')
153     plt.grid(True)
154     plt.legend()
155
156     # Loss
157     plt.subplot(1, 2, 2)
158     plt.plot(history.history['loss'], label='Train Loss')
159     plt.plot(history.history['val_loss'], label='Val Loss')
160     plt.xlabel('Epoch')
161     plt.ylabel('Loss')
162     plt.title('Loss vs. Epoch')
163     plt.grid(True)
164     plt.legend()
165
166     plt.tight_layout()
167     plt.show()
168
169 def plot_confusion_matrix(y_true, y_pred, label_encoder):
170     """
171     Plot confusion matrix with class labels.
172     """
173     cm = confusion_matrix(y_true, y_pred)
174     plt.figure(figsize=(8, 6))
175     sns.heatmap(
176         cm, annot=True, fmt='d', cmap='Blues',
177         xticklabels=label_encoder.classes_,
178         yticklabels=label_encoder.classes_
179     )
180     plt.title("Confusion Matrix")
181     plt.xlabel("Predicted")
182     plt.ylabel("True")
183     plt.show()
184
185 def plot_roc_curves(y_true, y_pred_proba, label_encoder):
186     """
187     Plot multi-class ROC curves for each label.
188     """
189     n_classes = len(label_encoder.classes_)
190     y_true_bin = to_categorical(y_true, n_classes)
191
192     plt.figure(figsize=(8, 6))
193     for i in range(n_classes):
194         fpr, tpr, _ = roc_curve(y_true_bin[:, i], y_pred_proba[:, i])
195         score = auc(fpr, tpr)
196         plt.plot(fpr, tpr, label=f"{label_encoder.classes_[i]} (AUC={score
    :.2f})")
197
198     plt.plot([0, 1], [0, 1], 'k--')
199     plt.xlabel("False Positive Rate")
200     plt.ylabel("True Positive Rate")
201     plt.title("ROC Curves")
202     plt.legend(loc='lower right')
203     plt.grid(True)
```

```
    plt.show()

def main():
    # Model Configuration
    CSV_FILE_PATH = r"C:\Users\Mohamed\Desktop\proj\health_monitoring_data\health_data.csv"
    SEQ_LENGTH = 10          # Sequence length of 10 time steps
    OVERLAP = 0.0           # Overlap ratio between sequences
    TEST_SIZE = 0.2         # Train/test split ratio
    VAL_SPLIT = 0.2         # Validation split ratio
    EPOCHS = 30             # Maximum number of training epochs
    BATCH_SIZE = 32         # Training batch size
    RANDOM_SEED = 42        # Random seed for reproducibility

    # 1) Load and Preprocess Data
    X, y, label_encoder, scaler, feature_names = load_and_preprocess_data(
        file_path=CSV_FILE_PATH,
        seq_length=SEQ_LENGTH,
        overlap=OVERLAP
    )

    # 2) Train/Test Split
    X_train, X_test, y_train, y_test = train_test_split(
        X,
        y,
        test_size=TEST_SIZE,
        shuffle=False,  # Keep chronological order
        random_state=RANDOM_SEED
    )
    print(f"Train set shape: {X_train.shape}, Test set shape: {X_test.shape}")

    # 3) Build Model
    num_features = X.shape[2]
    num_classes = len(np.unique(y))
    model = build_model(SEQ_LENGTH, num_features, num_classes)
    model.summary()

    # 4) Train Model
    callbacks = [
        EarlyStopping(
            monitor='val_loss',
            patience=5,
            restore_best_weights=True
        )
    ]

    history = model.fit(
        X_train, y_train,
        validation_split=VAL_SPLIT,
        epochs=EPOCHS,
        batch_size=BATCH_SIZE,
        callbacks=callbacks,
        verbose=1
    )

    # 5) Evaluate Model
    test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose=0)
    print(f"Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.4f}")
```

```
261
262     # 6) Visualize Results
263     # Training curves
264     plot_training_curves(history)
265
266     # Predictions and metrics
267     y_pred = np.argmax(model.predict(X_test), axis=1)
268     y_pred_proba = model.predict(X_test)
269
270     # Confusion matrix
271     plot_confusion_matrix(y_test, y_pred, label_encoder)
272
273     # Classification report
274     print("\nClassification Report:")
275     print(classification_report(y_test, y_pred, target_names=label_encoder.
    classes_))
276
277     # ROC curves
278     plot_roc_curves(y_test, y_pred_proba, label_encoder)
279
280     # 7) Save Model & Preprocessing Objects
281     print("\nSaving model and preprocessing objects...")
282     model.save("health_monitor_model.h5")
283     np.save("label_encoder_classes.npy", label_encoder.classes_)
284     joblib.dump(scaler, "scaler.pkl")
285     print("All done!")
286
287 if __name__ == "__main__":
288     main()
```

Listing 6.3 – modelTraining.py

— **Code 4** : Model Testing

```
1  import numpy as np
2  import pandas as pd
3  import time
4  import datetime
5  import random
6  import os
7  import tkinter as tk
8  from tkinter import ttk
9  import uuid
10 from matplotlib.figure import Figure
11 from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
12 from matplotlib.animation import FuncAnimation
13 import threading
14 import warnings
15 import tensorflow as tf
16 from tensorflow.keras.models import load_model
17 import joblib
18 warnings.filterwarnings("ignore", category=UserWarning)
19 class HealthPredictorDashboard:
20     def __init__(self):
21         self.root = tk.Tk()
22         self.root.title("Health Monitoring Dashboard with ML Prediction")
23         self.root.state('zoomed')
24         self.model = load_model("C:/Users/Mohamed/Desktop/proj/model/
    health_monitor_model.h5")
25         self.label_encoder_classes = np.load(
```

```
            "C:/Users/Mohamed/Desktop/proj/model/label_encoder_classes.npy",
            allow_pickle=True
        )
        self.scaler = joblib.load("C:/Users/Mohamed/Desktop/proj/model/scaler.pkl")
        self.generator = HealthDataGenerator()
        self.prediction_buffer = []
        self.accuracy_history = []
        self.running = True
        self.setup_ui()
        self.data_thread = threading.Thread(target=self.update_data, daemon=True)
        self.data_thread.start()
        self.root.protocol("WM_DELETE_WINDOW", self.on_closing)
    def setup_ui(self):
        control_frame = ttk.Frame(self.root, padding="5")
        graph_frame = ttk.Frame(self.root, padding="5")
        prediction_frame = ttk.Frame(self.root, padding="5")
        self.setup_graphs(graph_frame)
        self.setup_controls(control_frame)
        self.setup_prediction_panel(prediction_frame)
        control_frame.pack(side=tk.LEFT, fill=tk.Y, padx=5, pady=5)
        graph_frame.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
        prediction_frame.pack(side=tk.RIGHT, fill=tk.Y, padx=5, pady=5)
    def setup_prediction_panel(self, parent):
        pred_status = ttk.LabelFrame(parent, text="Prediction Status", padding="10")
        pred_status.pack(fill=tk.X, padx=5, pady=5)
        self.true_label = ttk.Label(pred_status, text="True Condition: Normal", font=('Helvetica', 12))
        self.true_label.pack(pady=5)
        self.pred_label = ttk.Label(pred_status, text="Predicted: Normal", font=('Helvetica', 12))
        self.pred_label.pack(pady=5)
        accuracy_frame = ttk.LabelFrame(parent, text="Model Performance", padding="10")
        accuracy_frame.pack(fill=tk.X, padx=5, pady=5)
        self.current_accuracy = ttk.Label(accuracy_frame,
                                          text="Current Accuracy: 100%",
                                          font=('Helvetica', 12))
        self.current_accuracy.pack(pady=5)
        self.acc_fig = Figure(figsize=(6, 4), facecolor='white')
        self.acc_ax = self.acc_fig.add_subplot(111)
        self.acc_ax.set_ylim(0, 100)
        self.acc_ax.set_title("Accuracy Over Time")
        self.acc_ax.set_ylabel("Accuracy (%)")
        self.acc_ax.grid(True)
        self.acc_canvas = FigureCanvasTkAgg(self.acc_fig, master=accuracy_frame)
        self.acc_canvas.draw()
        self.acc_canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)
    def setup_controls(self, parent):
        ttk.Label(parent, text="Control Panel", font=('Helvetica', 14, 'bold')).pack(pady=10)
        ttk.Label(parent, text="Data Generation Interval (seconds)").pack(pady=5)
        speed_scale = ttk.Scale(parent, from_=0.1, to=2.0,
                                command=lambda v: self.update_config('data_generation_interval', float(v)))
```

```
75          speed_scale.set(self.generator.config['data_generation_interval'])
76          speed_scale.pack(fill=tk.X, padx=5)
77          ttk.Label(parent, text="Window Size (samples)").pack(pady=5)
78          window_scale = ttk.Scale(parent, from_=50, to=500,
79                                   command=lambda v: self.update_config('
    window_size', int(float(v))))
80          window_scale.set(self.generator.config['window_size'])
81          window_scale.pack(fill=tk.X, padx=5)
82          ttk.Label(parent, text="Anomaly Probability").pack(pady=5)
83          prob_scale = ttk.Scale(parent, from_=0, to=0.2,
84                                 command=lambda v: self.update_config('
    anomaly_probability', float(v)))
85          prob_scale.set(self.generator.config['anomaly_probability'])
86          prob_scale.pack(fill=tk.X, padx=5)
87          status_frame = ttk.LabelFrame(parent, text="Current Status",
    padding="5")
88          status_frame.pack(fill=tk.X, padx=5, pady=10)
89          self.status_labels = {
90              'anomaly': ttk.Label(status_frame, text="Status: Normal"),
91              'bp': ttk.Label(status_frame, text="BP: --/-- mmHg"),
92              'hr': ttk.Label(status_frame, text="HR: -- bpm"),
93              'spo2': ttk.Label(status_frame, text="SpO2: --%"),
94              'rr': ttk.Label(status_frame, text="RR: -- /min")
95          }
96          for label in self.status_labels.values():
97              label.pack(pady=2)
98      def setup_graphs(self, parent):
99          self.fig = Figure(figsize=(12, 8), facecolor='white')
100         self.fig.subplots_adjust(hspace=0.3)
101         self.axes = {
102             'bp': self.fig.add_subplot(411),
103             'hr': self.fig.add_subplot(412),
104             'spo2': self.fig.add_subplot(413),
105             'rr': self.fig.add_subplot(414)
106         }
107         for ax in self.axes.values():
108             ax.set_facecolor('white')
109             ax.grid(True, linestyle='--', alpha=0.7)
110             ax.tick_params(labelcolor='black')
111         self.axes['bp'].set_ylabel('Blood Pressure\n(mmHg)', color='black')
112         self.axes['hr'].set_ylabel('Heart Rate\n(bpm)', color='black')
113         self.axes['spo2'].set_ylabel('SpO2\n(%)', color='black')
114         self.axes['rr'].set_ylabel('Respiratory Rate\n(/min)', color='black
    ')
115         self.lines = {
116             'systolic': self.axes['bp'].plot([], [], 'r-', label='Systolic'
    )[0],
117             'diastolic': self.axes['bp'].plot([], [], 'b-', label='
    Diastolic')[0],
118             'hr': self.axes['hr'].plot([], [], 'g-', label='Heart Rate')
    [0],
119             'spo2': self.axes['spo2'].plot([], [], 'm-', label='SpO2')[0],
120             'rr': self.axes['rr'].plot([], [], 'c-', label='Respiratory
    Rate')[0]
121         }
122         for ax in self.axes.values():
123             ax.legend(loc='upper right')
124         self.canvas = FigureCanvasTkAgg(self.fig, master=parent)
125         self.canvas.draw()
126         self.canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)
```

```
        self.ani = FuncAnimation(self.fig, self.update_plots, interval=50, blit=True)
    def make_prediction(self, data_point):
        self.prediction_buffer.append([
            data_point['systolic_bp'],
            data_point['diastolic_bp'],
            data_point['heart_rate'],
            data_point['spo2'],
            data_point['respiratory_rate'],
            data_point['systolic_bp'] - data_point['diastolic_bp'],
            data_point['heart_rate'] / (data_point['systolic_bp'] + 1e-6)
        ])
        if len(self.prediction_buffer) > 10:
            self.prediction_buffer.pop(0)
        if len(self.prediction_buffer) == 10:
            scaled_data = self.scaler.transform(self.prediction_buffer)
            model_input = scaled_data.reshape(1, 10, -1)
            pred_proba = self.model.predict(model_input, verbose=0)
            pred_class = np.argmax(pred_proba)
            predicted_label = self.label_encoder_classes[pred_class]
            correct = predicted_label == data_point['anomaly_type']
            self.accuracy_history.append(correct)
            if len(self.accuracy_history) > 100:
                self.accuracy_history.pop(0)
            current_accuracy = (sum(self.accuracy_history) / len(self.accuracy_history)) * 100
            self.root.after(0, self.update_prediction_display,
                            data_point['anomaly_type'],
                            predicted_label,
                            current_accuracy)
    def update_prediction_display(self, true_label, pred_label, accuracy):
        MAX_DISPLAY_POINTS = 50
        self.true_label.config(
            text=f"True Condition: {true_label}",
            foreground="black"
        )
        self.pred_label.config(
            text=f"Predicted: {pred_label}",
            foreground="green" if true_label == pred_label else "red"
        )
        self.current_accuracy.config(text=f"Current Accuracy: {accuracy:.1f}%")
        self.acc_ax.clear()
        display_history = self.accuracy_history[-MAX_DISPLAY_POINTS:]
        x_values = range(len(display_history))
        y_values = [1 if val else 0 for val in display_history]
        y_values = [y * 100 for y in y_values]
        self.acc_ax.plot(x_values, y_values, 'b-', linewidth=2)
        self.acc_ax.set_ylim(-5, 105)
        self.acc_ax.axhline(y=0, color='red', linestyle='-', alpha=0.2)
        self.acc_ax.axhline(y=100, color='green', linestyle='-', alpha=0.2)
        self.acc_ax.set_title("Prediction Accuracy (Real-time)", fontsize=12)
        self.acc_ax.set_ylabel("Accuracy (%)", fontsize=10)
        self.acc_ax.set_xlabel("Recent Predictions", fontsize=10)
        self.acc_ax.grid(True, linestyle='--', alpha=0.3)
        avg_accuracy = sum(display_history) / len(display_history) * 100
        self.acc_ax.text(0.02, 0.98, f'Running Accuracy: {avg_accuracy:.1f}%',
                         transform=self.acc_ax.transAxes,
```

```
                        verticalalignment='top',
                        fontsize=10)
        self.acc_canvas.draw()
    def update_config(self, param, value):
        self.generator.config[param] = value
    def update_plots(self, frame):
        data = self.generator.data
        if not data:
            return self.lines.values()
        x = range(len(data))
        systolic_data = [d['systolic_bp'] for d in data]
        diastolic_data = [d['diastolic_bp'] for d in data]
        self.lines['systolic'].set_data(x, systolic_data)
        self.lines['diastolic'].set_data(x, diastolic_data)
        self.lines['hr'].set_data(x, [d['heart_rate'] for d in data])
        self.lines['spo2'].set_data(x, [d['spo2'] for d in data])
        self.lines['rr'].set_data(x, [d['respiratory_rate'] for d in data])
        if data:
            self.axes['bp'].set_ylim(min(min(diastolic_data) - 10, 40),
                                     max(max(systolic_data) + 10, 200))
            self.axes['hr'].set_ylim(min(min(d['heart_rate'] for d in data)
 - 5, 40),
                                     max(max(d['heart_rate'] for d in data) +
 5, 150))
            self.axes['spo2'].set_ylim(min(min(d['spo2'] for d in data) -
2, 80),
                                       max(max(d['spo2'] for d in data) + 2,
100))
            self.axes['rr'].set_ylim(min(min(d['respiratory_rate'] for d in
 data) - 2, 8),
                                     max(max(d['respiratory_rate'] for d in
data) + 2, 30))
        for ax in self.axes.values():
            ax.set_xlim(0, len(data))
        return self.lines.values()
    def update_data(self):
        while self.running:
            data_point = self.generator.generate_reading()
            if data_point:
                self.root.after(0, self.update_status_labels, data_point)
                self.make_prediction(data_point)
            time.sleep(0.05)
    def update_status_labels(self, data_point):
        if data_point['anomaly_type'] == "normal":
            self.status_labels['anomaly'].config(text="Status: Normal",
foreground="green")
        else:
            self.status_labels['anomaly'].config(
                text=f"Status:        {data_point['anomaly_type'].replace('
_', ' ').title()}",
                foreground="red"
            )
        self.status_labels['bp'].config(
            text=f"BP: {data_point['systolic_bp']:.0f}/{data_point['
diastolic_bp']:.0f} mmHg"
        )
        self.status_labels['hr'].config(
            text=f"HR: {data_point['heart_rate']:.0f} bpm"
        )
        self.status_labels['spo2'].config(
```

```
            text=f"SpO2: {data_point['spo2']:.0f}%"
        )
        self.status_labels['rr'].config(
            text=f"RR: {data_point['respiratory_rate']:.0f} /min"
        )
    def on_closing(self):
        self.running = False
        self.root.quit()
        self.root.destroy()
    def run(self):
        try:
            self.root.mainloop()
        finally:
            self.running = False
class HealthDataGenerator:
    def __init__(self):
        self.config = {
            'data_generation_interval': 0.5,
            'window_size': 100,
            'anomaly_probability': 0.03
        }
        self.base_params = {
            'blood_pressure': {
                'systolic': {'mean': 120, 'std': 5, 'min': 70, 'max': 180},
                'diastolic': {'mean': 80, 'std': 3, 'min': 40, 'max': 120},
            },
            'heart_rate': {
                'mean': 75, 'std': 3, 'min': 40, 'max': 150,
                'respiratory_sinus_arrhythmia': 0.1
            },
            'spo2': {
                'mean': 98, 'std': 0.5, 'min': 80, 'max': 100
            },
            'respiratory_rate': {
                'mean': 16, 'std': 1, 'min': 8, 'max': 30
            }
        }
        self.anomaly_types = {
            'cardiac_arrest': {
                'blood_pressure': {'systolic_shift': -40, 'diastolic_shift'
    : -30, 'std': 15},
                'heart_rate': {'shift': -30, 'std': 20},
                'spo2': {'shift': -15, 'std': 5},
                'respiratory_rate': {'shift': 8, 'std': 3},
                'duration': 15,
                'onset_speed': 'rapid'
            },
            'hypertensive_crisis': {
                'blood_pressure': {'systolic_shift': 60, 'diastolic_shift':
     40, 'std': 10},
                'heart_rate': {'shift': 30, 'std': 15},
                'spo2': {'shift': -5, 'std': 2},
                'respiratory_rate': {'shift': 6, 'std': 2},
                'duration': 20,
                'onset_speed': 'gradual'
            },
            'septic_shock': {
                'blood_pressure': {'systolic_shift': -30, 'diastolic_shift'
    : -20, 'std': 10},
                'heart_rate': {'shift': 40, 'std': 10},
```

```
                'spo2': {'shift': -10, 'std': 3},
                'respiratory_rate': {'shift': 10, 'std': 3},
                'duration': 25,
                'onset_speed': 'gradual'
            }
        }
        self.data = []
        self.current_anomaly = None
        self.anomaly_start_time = None
        self.last_update = time.time()
    def apply_physiological_constraints(self, readings):
        if readings['heart_rate'] > self.base_params['heart_rate']['mean']:
            factor = (readings['heart_rate'] - self.base_params['heart_rate']['mean']) / 50
            readings['blood_pressure']['systolic'] *= (1 + 0.1 * factor)
            readings['blood_pressure']['diastolic'] *= (1 + 0.05 * factor)
        if readings['spo2'] < 90:
            readings['heart_rate'] *= (1 + (90 - readings['spo2']) / 100)
        return readings
    def generate_reading(self):
        current_time = time.time()
        if current_time - self.last_update < self.config['data_generation_interval']:
            return None
        self.last_update = current_time
        timestamp = datetime.datetime.now()
        readings = {
            'blood_pressure': {
                'systolic': 0,
                'diastolic': 0
            },
            'heart_rate': 0,
            'spo2': 0,
            'respiratory_rate': 0
        }
        if self.current_anomaly is None and random.random() < self.config['anomaly_probability']:
            self.current_anomaly = random.choice(list(self.anomaly_types.keys()))
            self.anomaly_start_time = current_time
        if self.current_anomaly:
            anomaly = self.anomaly_types[self.current_anomaly]
            progress = (current_time - self.anomaly_start_time) / anomaly['duration']
            for vital in readings.keys():
                if vital == 'blood_pressure':
                    sys_shift = anomaly['blood_pressure']['systolic_shift']
                    dia_shift = anomaly['blood_pressure']['diastolic_shift']
                    std = anomaly['blood_pressure']['std']
                    readings[vital]['systolic'] = np.random.normal(
                        self.base_params[vital]['systolic']['mean'] + sys_shift * progress,
                        std
                    )
                    readings[vital]['diastolic'] = np.random.normal(
                        self.base_params[vital]['diastolic']['mean'] + dia_shift * progress,
                        std
                    )
```

```
342                 else:
343                     shift = anomaly[vital]['shift'] if vital in anomaly
    else 0
344                     std = anomaly[vital]['std'] if vital in anomaly else
    self.base_params[vital]['std']
345                     readings[vital] = np.random.normal(
346                         self.base_params[vital]['mean'] + shift * progress,
347                         std
348                     )
349             if current_time - self.anomaly_start_time > anomaly['duration'
    ]:
350                 self.current_anomaly = None
351                 self.anomaly_start_time = None
352         else:
353             for vital in readings.keys():
354                 if vital == 'blood_pressure':
355                     readings[vital]['systolic'] = np.random.normal(
356                         self.base_params[vital]['systolic']['mean'],
357                         self.base_params[vital]['systolic']['std']
358                     )
359                     readings[vital]['diastolic'] = np.random.normal(
360                         self.base_params[vital]['diastolic']['mean'],
361                         self.base_params[vital]['diastolic']['std']
362                     )
363                 else:
364                     readings[vital] = np.random.normal(
365                         self.base_params[vital]['mean'],
366                         self.base_params[vital]['std']
367                     )
368         readings = self.apply_physiological_constraints(readings)
369         readings['blood_pressure']['systolic'] = np.clip(
370             readings['blood_pressure']['systolic'],
371             self.base_params['blood_pressure']['systolic']['min'],
372             self.base_params['blood_pressure']['systolic']['max']
373         )
374         readings['blood_pressure']['diastolic'] = np.clip(
375             readings['blood_pressure']['diastolic'],
376             self.base_params['blood_pressure']['diastolic']['min'],
377             self.base_params['blood_pressure']['diastolic']['max']
378         )
379         for vital in ['heart_rate', 'spo2', 'respiratory_rate']:
380             readings[vital] = np.clip(
381                 readings[vital],
382                 self.base_params[vital]['min'],
383                 self.base_params[vital]['max']
384             )
385         data_point = {
386             'timestamp': timestamp,
387             'systolic_bp': readings['blood_pressure']['systolic'],
388             'diastolic_bp': readings['blood_pressure']['diastolic'],
389             'heart_rate': readings['heart_rate'],
390             'spo2': readings['spo2'],
391             'respiratory_rate': readings['respiratory_rate'],
392             'anomaly_type': self.current_anomaly if self.current_anomaly
    else "normal"
393         }
394         self.data.append(data_point)
395         if len(self.data) > self.config['window_size']:
396             self.data.pop(0)
397         return data_point
```

```
398 def main():
399     dashboard = HealthPredictorDashboard()
400     dashboard.run()
401 if __name__ == "__main__":
402     main()
```

Listing 6.4 – RunTheModel.py

Chapitre 7

Conclusion Finale

Ce projet vise à développer un système IoT avancé pour surveiller les paramètres vitaux, tels que la pression sanguine et la fréquence cardiaque, en intégrant des techniques d'intelligence artificielle pour la gestion des ressources et la détection des anomalies.

Le projet s'articule autour de trois axes principaux :

1. **Prédiction des ressources** : Modèles de machine learning pour anticiper la consommation énergétique des capteurs IoT, évalués avec des métriques comme le RMSE.
2. **Détection des anomalies** : Réseaux neuronaux (autoencodeurs, RNN) pour identifier les comportements anormaux dans les données des capteurs.
3. **Ajustement des ressources** : Agent de reinforcement learning pour optimiser dynamiquement la gestion des ressources en temps réel.