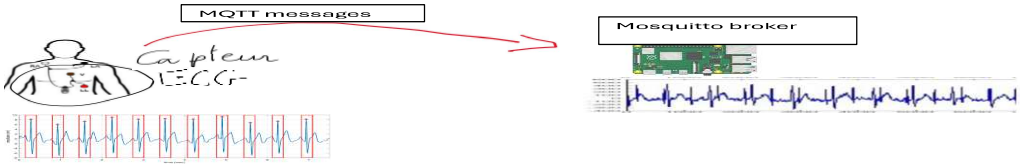
Technologies et Protocoles pour l'IoT

TP : capteurs ECG pour le rythme cardiaque et apprentissage pour la reconstruction du signale complet au niveau IoT Edge



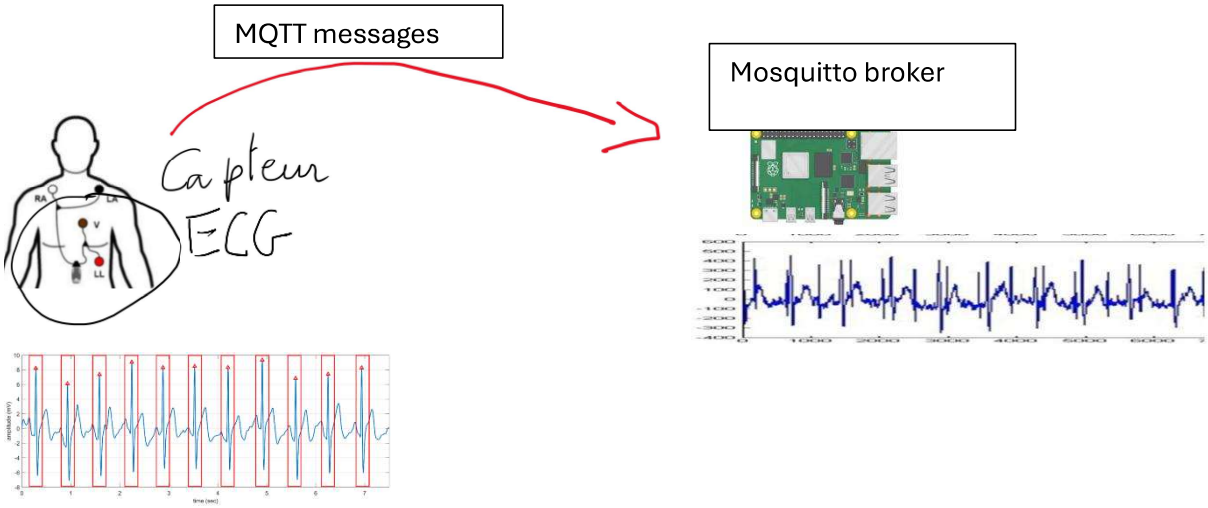
Toujours pour la même problématique de réduction des flux de données vers le Edge et vers le clouds.

Problématique
Les capteurs IoT (ex. ECG, température, vibration) génèrent des **données en continu** à haute fréquence.
Si on envoie **toutes les données brutes** vers le Edge ou le Cloud :

- **Consommation de bande passante élevée**
- **Latence accrue**
- **Coût de stockage et traitement important**
- **Impact énergétique** (batterie des capteurs)

Solutions courantes

1. **Réduction de la fréquence d'échantillonnage**
    - Moins de points → moins de données transmises.
    - Problème : perte de précision (signal dégradé).
2. **Compression des données**
    - Algorithmes comme **lossless** (ZIP) ou **lossy** (quantification).
    - Problème : complexité computationnelle sur capteurs limités.
3. **Filtrage et agrégation locale**
    - Calculer des **moyennes**, **min/max**, ou **features** (ex. fréquence cardiaque) avant envoi.
    - Réduit le volume mais peut perdre des détails.
4. **Edge Intelligence (IA au bord)**
    - Utiliser des modèles (ex. LSTM) pour **prédire ou reconstruire** les données manquantes.
    - Envoi uniquement des points critiques ou anomalies.



L'objectif de ce TP est de **traiter et reconstruire correctement les signaux ECG représentant le rythme cardiaque**, en tenant compte des contraintes liées à la transmission et à l'échantillonnage.

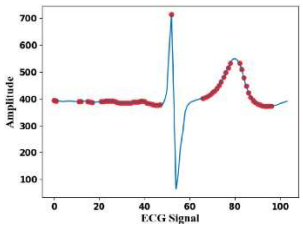
Le signal initial est capté par des capteurs, puis **échantillonné** et transmis vers une plateforme **Edge** via le protocole **MQTT**. Cependant, en raison de la réduction de la fréquence d'échantillonnage et des variations de délais de transmission, le signal reconstruit au niveau de l'Edge présente des **lacunes** et une **dégradation de qualité**.

Partie 1 : Mise en évidence du problème

- Implémenter la chaîne de transmission pour montrer l'impact de la **réduction de la fréquence d'échantillonnage** (nécessaire pour diminuer le volume de données transmises). Il faut implémenter deux processus/programmes python. Le premier qui permet de simuler le capteur EGC pour la génération de signal du rythme cardiaque. Il faut faire des petites variations dans les signaux pour simuler l'imprécision des capteurs. Puis ajouter un échantillonnage et envoi de messages MQTT sur les valeurs du signal pour les instants. Le deuxième programme est celui dans le Edge qui permet de lire les valeurs des échenillions et de les afficher.
- Analyser la qualité du signal reconstruit : démontrer que **moins de points d'échantillonnage** entraînent une **reconstruction imprécise** et une perte d'informations importantes. Pour analyser les résultats, il n'y a pas mieux que la visualisons des courbes des signaux ECG affiché au niveau du Edge. Vous pouvez tester les d'échantillonnage qui prend 5 points seulement d'une période ECG. Puis un autre exemple avec 20 points du ECG.

Partie 2 : Amélioration par apprentissage automatique

- Utiliser un modèle **LSTM (Long Short-Term Memory)** pour **prédire les valeurs manquantes** et améliorer la reconstruction du signal.
- Mettre en place une phase d'**apprentissage supervisé** pour que le modèle capture l'**allure typique des courbes ECG** et réalise un **forecasting** lorsque les points sont insuffisants.
- Évaluer la performance du modèle en comparant le signal reconstruit avec et sans prédiction.



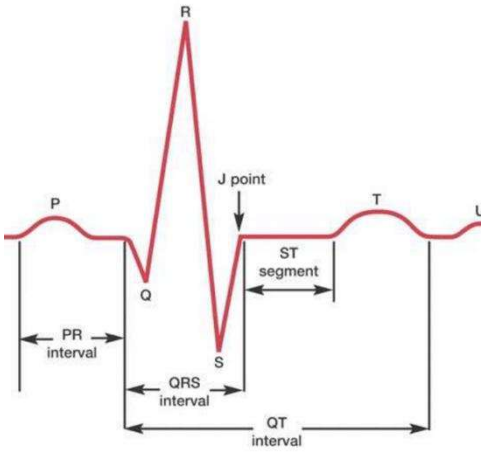
Exemple concret : ECG vers Edge

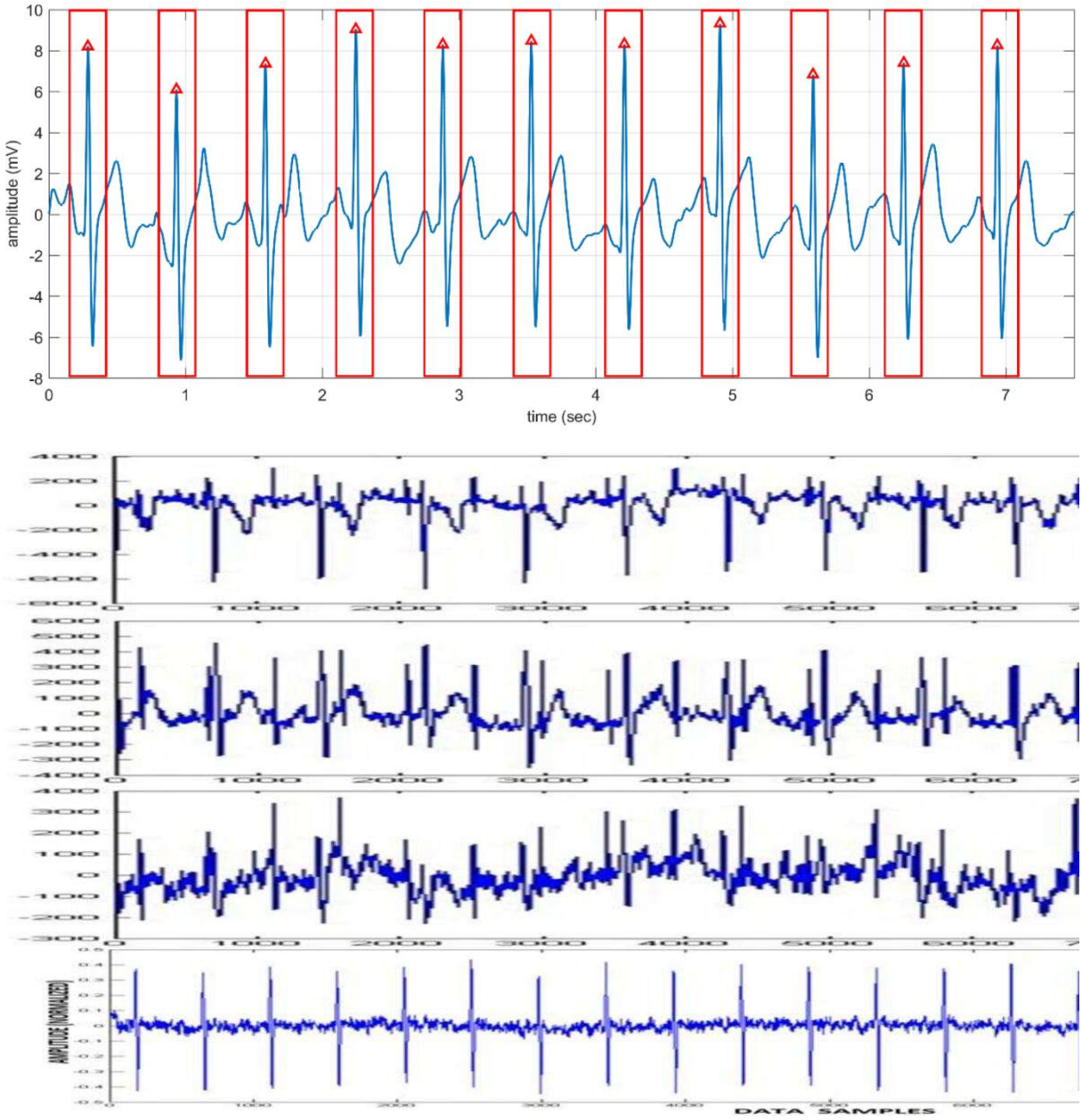
- **Situation** : Un capteur ECG envoie 1000 points/seconde.
- **Problème** : Trop de données pour le réseau MQTT → surcharge.
- **Solution** :
  - Réduire l'échantillonnage à 200 points/seconde.
  - Sur le Edge, utiliser un **modèle LSTM** pour **reconstruire la courbe complète**.
  - Envoi au Cloud uniquement des **anomalies détectées** (ex. arythmie).

Avantages

- Réduction de **80% du flux**.
- Maintien de la qualité grâce à **prédiction intelligente**.
- Moins de latence et coûts Cloud.

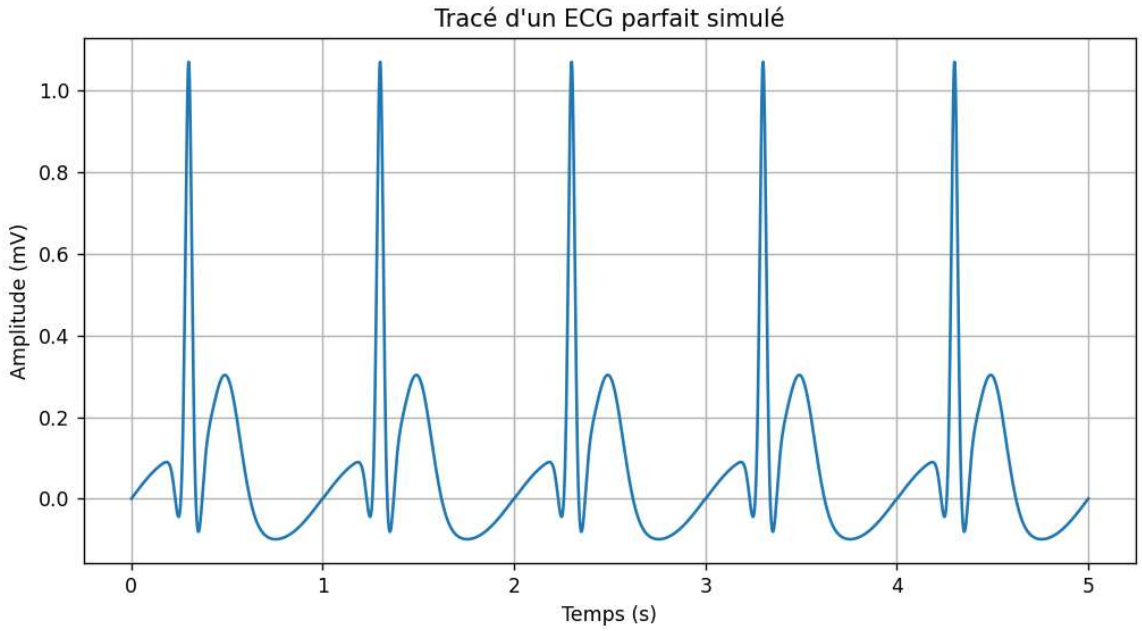
Background et code Types





1. ECG parfait

Traçage d'une courbe idéale d'un ECG (électrocardiogramme) simulé. Ce n'est pas basé sur des signaux physiologiques réels, mais ça imite les pics caractéristiques du cycle cardiaque (onde P, complexe QRS, onde T).

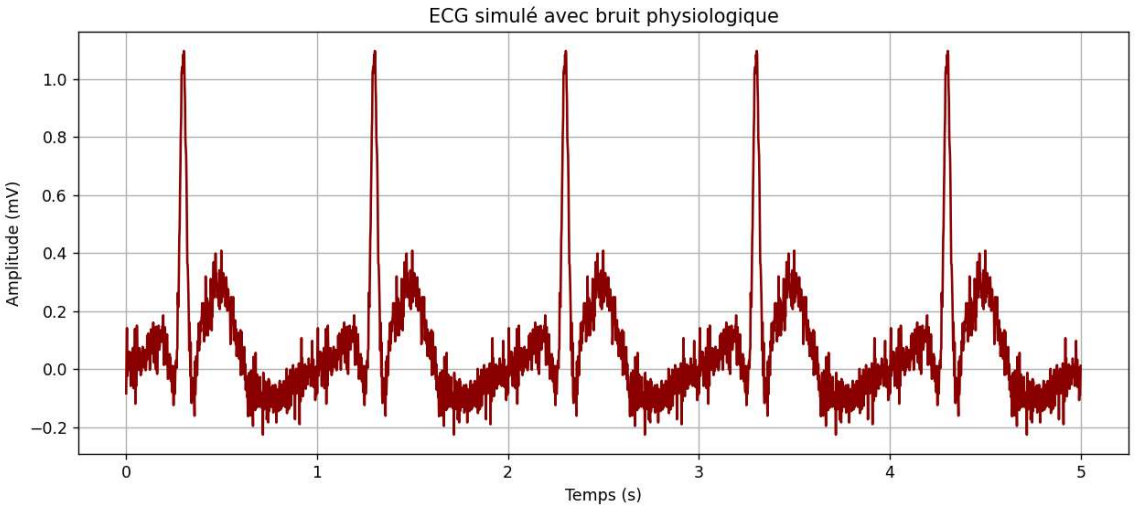



```
import numpy as np
import matplotlib.pyplot as plt

# Fonction pour générer une onde ECG simulée
def ecg_synthetique(t):
  # Création d'une onde idéale avec des composantes de type onde P, QRS, et T
  return (
    0.1 * np.sin(2 * np.pi * t * 1) +  # petite onde P
    -0.15 * np.exp(-((t - 0.25) ** 2) / 0.001) +  # creux Q
    1.0 * np.exp(-((t - 0.3) ** 2) / 0.0005) +  # pic R
    -0.2 * np.exp(-((t - 0.35) ** 2) / 0.001) +  # creux S
    0.3 * np.exp(-((t - 0.5) ** 2) / 0.01)  # onde T
  )

# Paramètres du signal
t = np.linspace(0, 1, 500)  # 1 seconde, 500 points
ecg_signal = ecg_synthetique(t)

# Répétition pour simuler plusieurs battements
cycle = ecg_synthetique(t)
ecg_complet = np.tile(cycle, 5)  # 5 cycles cardiaques
t_total = np.linspace(0, 5, len(ecg_complet))

# Affichage du tracé
plt.plot(t_total, ecg_complet)
plt.title("Tracé d'un ECG parfait simulé")
plt.xlabel("Temps (s)")
plt.ylabel("Amplitude (mV)")
plt.grid(True)
plt.show()
```

Quelques notes utiles :

- Ce code modélise un ECG idéal avec des fonctions mathématiques.
- Pour des ECGs réalistes ou cliniques, il faut utiliser des signaux enregistrés ou des bibliothèques comme wfdb.
- Tu peux aussi jouer sur la fréquence ou ajouter du bruit pour imiter des variations cardiaques.

2. Bruit sur ECG



Ajouter du **bruit** permet de rendre le tracé ECG plus réaliste, en simulant les petites fluctuations dues à des artefacts ou à des variations physiologiques naturelles. Voici une version modifiée du code où l'on ajoute du bruit aléatoire :

```
import numpy as np
import matplotlib.pyplot as plt

# Fonction pour générer une onde ECG simulée
def ecg_synthetique(t):
  return (
    0.1 * np.sin(2 * np.pi * t * 1) +              # Onde P
    -0.15 * np.exp(-((t - 0.25) ** 2) / 0.001) +      # Onde Q
    1.0 * np.exp(-((t - 0.3) ** 2) / 0.0005) +        # Pic R
    -0.2 * np.exp(-((t - 0.35) ** 2) / 0.001) +       # Onde S
    0.3 * np.exp(-((t - 0.5) ** 2) / 0.01)            # Onde T
  )

# Paramètres du signal
t = np.linspace(0, 1, 500)  # 1 seconde, 500 points
cycle = ecg_synthetique(t)

# Génération de bruit aléatoire
bruit = 0.05 * np.random.normal(size=cycle.shape)
```

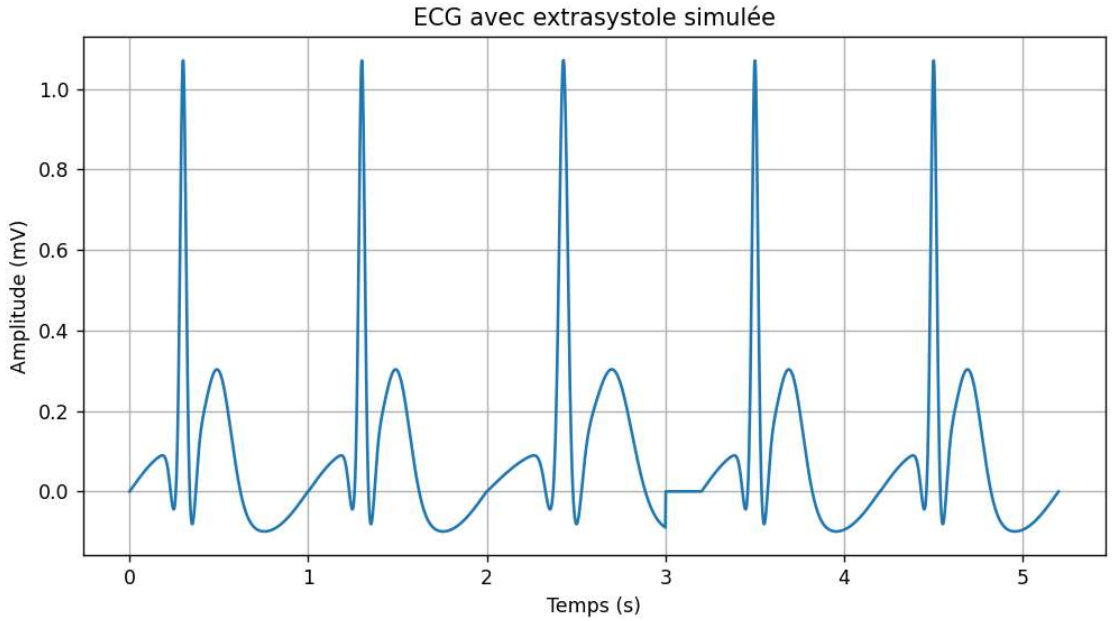
```
# Ajout du bruit à chaque cycle
cycle_bruite = cycle + bruit

# Répétition pour plusieurs battements
ecg_complet = np.tile(cycle_bruite, 5)
t_total = np.linspace(0, 5, len(ecg_complet))

# Tracé de la courbe ECG bruitée
plt.plot(t_total, ecg_complet, color='darkred')
plt.title("ECG simulé avec bruit physiologique")
plt.xlabel("Temps (s)")
plt.ylabel("Amplitude (mV)")
plt.grid(True)
plt.show()
```

Personnalisation possible :

- Tu peux modifier la **quantité de bruit** (change la valeur 0.05) pour simuler des cas cliniques différents.

3. simulation Extrasystole

L'extrasystole est une contraction prématurée du cœur : le battement arrive trop tôt, suivi d'une pause compensatrice. On peut la simuler en insérant un battement décalé et plus petit dans le signal :




```
import numpy as np
import matplotlib.pyplot as plt

def ecg_cycle(t):
```

```
    return (
        0.1 * np.sin(2 * np.pi * t * 1) +
        -0.15 * np.exp(-((t - 0.25) ** 2) / 0.001) +
        1.0 * np.exp(-((t - 0.3) ** 2) / 0.0005) +
        -0.2 * np.exp(-((t - 0.35) ** 2) / 0.001) +
        0.3 * np.exp(-((t - 0.5) ** 2) / 0.01)
    )

# Génère plusieurs cycles normaux
t = np.linspace(0, 1, 500)
cycle_normal = ecg_cycle(t)
ecg = []

for i in range(5):
    if i == 2:
        # Extrasystole : insertion d'un cycle prématuré
        ecg.extend(ecg_cycle(t * 0.7))  # plus court et plus rapide
        ecg.extend(np.zeros(100))  # pause compensatrice
    else:
        ecg.extend(cycle_normal)

t_total = np.linspace(0, len(ecg)/500, len(ecg))
plt.plot(t_total, ecg)
plt.title("ECG avec extrasystole simulée")
plt.xlabel("Temps (s)")
plt.ylabel("Amplitude (mV)")
plt.grid(True)
plt.show()
```

4. simulation prédiction LSTM

Le **LSTM (Long Short-Term Memory)** est un type particulier de réseau de neurones récurrent (RNN) conçu pour traiter des **données séquentielles** (comme des signaux ECG, du texte ou des séries temporelles) tout en résolvant le problème du **gradient qui disparaît** rencontré dans les RNN classiques. Gère les dépendances à long terme dans les séquences. Très utilisé pour : **Prévision de séries temporelles** (finance, ECG, météo), Traitement du langage naturel, Reconnaissance vocale, ...

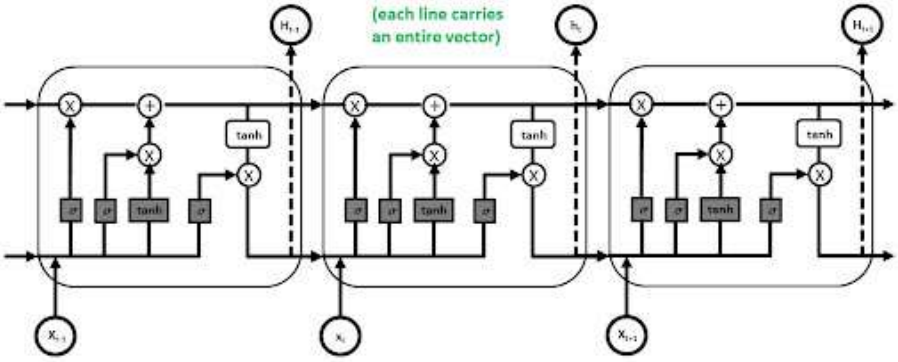
Short Term vs Long Term dans LSTM

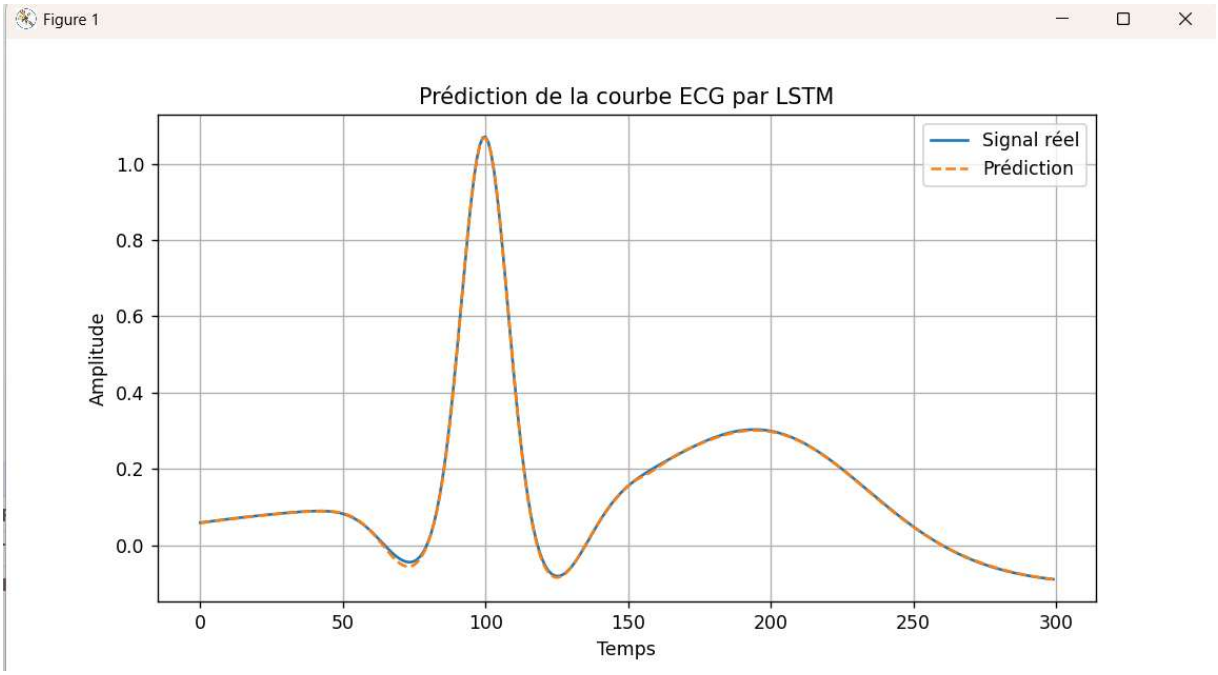
- **Mémoire à court terme (h_t)** :
    - Contient le **contexte immédiat** (informations récentes).
    - Sert à la prédiction instantanée.
- **Mémoire à long terme (c_t)** :
    - Stocke les **tendances globales** sur une longue durée.

- Permet de garder des informations importantes malgré des séquences longues.

Pourquoi deux mémoires ?

- Les RNN classiques oublient vite les infos anciennes.
- LSTM sépare $\mathbf{h_t}$ (rapide, local) et $\mathbf{c_t}$ (lent, global) pour gérer dépendances **courtes et longues**.






```python
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

# 1. Génération d'un signal ECG simplifié
def ecg_cycle(t):
    return (
        0.1 * np.sin(2 * np.pi * t * 1) +
        -0.15 * np.exp(-((t - 0.25) ** 2) / 0.001) +
        1.0 * np.exp(-((t - 0.3) ** 2) / 0.0005) +
```

```
        -0.2 * np.exp(-((t - 0.35) ** 2) / 0.001) +
        0.3 * np.exp(-((t - 0.5) ** 2) / 0.01)
    )

t = np.linspace(0, 1, 500)
signal = np.tile(ecg_cycle(t), 10)  # 10 cycles

# 2. Préparation des séquences
sequence_length = 50
X, y = [], []
for i in range(len(signal) - sequence_length):
    X.append(signal[i:i+sequence_length])
    y.append(signal[i+sequence_length])

X = np.array(X)
y = np.array(y)

# Reshape en (samples, timesteps, features)
X = X.reshape((X.shape[0], X.shape[1], 1))

# 3. Modèle LSTM
model = Sequential([
    LSTM(64, input_shape=(sequence_length, 1)),
    Dense(1)
])
model.compile(optimizer='adam', loss='mse')

# 4. Entraînement
model.fit(X, y, epochs=10, batch_size=32)

# 5. Prédiction
y_pred = model.predict(X)

# 6. Affichage
plt.plot(y[:300], label='Signal réel')
plt.plot(y_pred[:300], label='Prédiction', linestyle='--')
plt.title("Prédiction de la courbe ECG par LSTM")
plt.xlabel("Temps")
plt.ylabel("Amplitude")
plt.legend()
plt.grid(True)
plt.show()
```

Explications :
LSTM(64, input_shape=(sequence_length, 1))

- **LSTM** : couche récurrente qui apprend les dépendances temporelles.
- **64** : nombre d'unités LSTM (neurones) → plus il y en a, plus la capacité d'apprentissage est grande.
- **input_shape=(sequence_length, 1)** :
    - sequence_length = longueur de la séquence (nombre de pas temporels).

- 1 = nombre de features par pas (ici, une seule valeur par point, typique pour un signal ECG).

Dense(1)

- Couche entièrement connectée avec **1 neurone en sortie**.
- Sert à prédire une seule valeur (par exemple, le prochain point du signal).

Fonctionnement

- LSTM reçoit une séquence (ex. 50 points ECG).
- Il extrait les **patterns temporels** grâce à ses mémoires (short-term et long-term).
- La couche Dense transforme la sortie LSTM en **valeur prédite** (forecasting).

Étapes améliorées pour plusieurs cycles ECG

1. Générer un signal avec plusieurs cycles

python
```
import numpy as np

def ecg_cycle(t):
    return (
        0.1 * np.sin(2 * np.pi * t * 1) +
        -0.15 * np.exp(-((t - 0.25) ** 2) / 0.001) +
        1.0 * np.exp(-((t - 0.3) ** 2) / 0.0005) +
        -0.2 * np.exp(-((t - 0.35) ** 2) / 0.001) +
        0.3 * np.exp(-((t - 0.5) ** 2) / 0.01)
    )

# Signal : 10 cycles ECG simulés
t_single = np.linspace(0, 1, 500)
cycle = ecg_cycle(t_single)
signal = np.tile(cycle, 10)  # 10 battements
```

2. Créer des séquences pour LSTM

python
```
sequence_length = 100  # Plus long pour capter la forme d'un battement
complet
X, y = [], []

for i in range(len(signal) - sequence_length):
    X.append(signal[i:i + sequence_length])
    y.append(signal[i + sequence_length])

X = np.array(X).reshape(-1, sequence_length, 1)
y = np.array(y)
```

3. Construire et entraîner le modèle

python
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

model = Sequential([
    LSTM(64, input_shape=(sequence_length, 1), return_sequences=False),
    Dense(1)
])
model.compile(optimizer='adam', loss='mse')
model.fit(X, y, epochs=20, batch_size=64)
```

4. Faire la prédiction et visualiser

python
```
y_pred = model.predict(X)

import matplotlib.pyplot as plt
plt.plot(y[:1000], label='ECG réel')
plt.plot(y_pred[:1000], label='ECG prédit', linestyle='--')
plt.title("Prédiction ECG sur plusieurs cycles")
plt.xlabel("Temps")
plt.ylabel("Amplitude")
plt.legend()
plt.grid(True)
plt.show()
```

~~**Tester d'autres architectures comme Bidirectional LSTM, GRU, ou même des CNN-LSTM**~~.