Projet IA/IoT : prédiction de la température extérieure

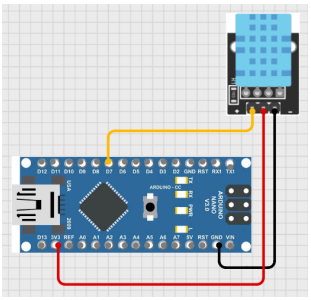
Dans le cadre du cours d'IoT et d'IA du Master Cybersécurité, nous avons voulu développer un système de prédiction de température à l'aide d'un capteur DHT11 connecté à une carte Arduino Nano 33 BLE Rev2. Cette carte dispose de 256 KB de SRAM, utile pour les calculs, ainsi que de 1 MB de mémoire Flash, intéressante pour stocker des variables (par exemple les poids du modèle).

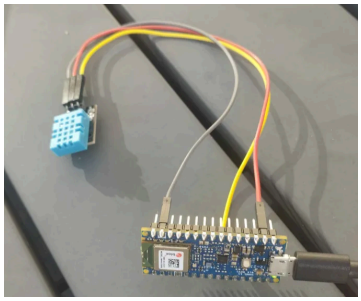
Ce rapport va vous détailler notre méthodologie, les difficultés rencontrées ainsi que les principales avancées réalisées au cours de ce projet.

1/ Récupération des données

➢ Prototype

Dans un premier temps, nous avons construit un prototype simple nous permettant de récupérer les données de notre capteur et de vérifier son bon fonctionnement. Pour cela, nous avons utilisé la *DTH sensor library* (1.4.6) d'Adafruit.





test_thermometer.ino

```
#include "DHT.h"

#define DHTPIN 7
#define DHTTYPE DHT11

DHT dht(DHTPIN, DHTTYPE);

void setup() {
  Serial.begin(9600);
  dht.begin();
}

void loop() {
  float h = dht.readHumidity();
  float t = dht.readTemperature();

  Serial.print("T = ");
  Serial.print(t);
  Serial.print(" °C, H = ");
  Serial.print(h);
  Serial.println(" %");

  delay(2000);
}
```

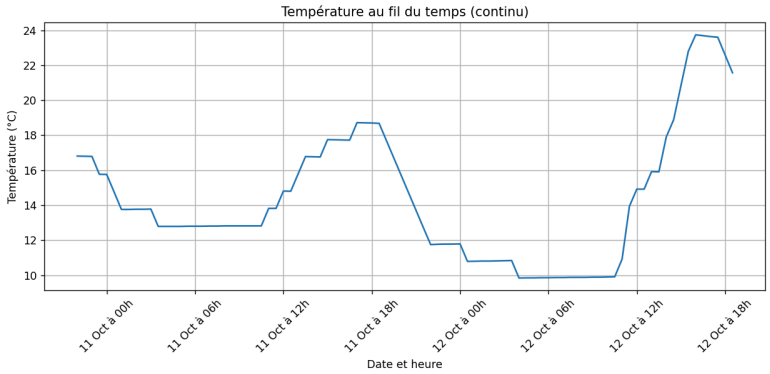
➢ Récupération des données sur le long terme

Pour entraîner notre modèle, il nous fallait récupérer un jeu de données conséquent, ce qui impliquait de laisser le capteur DHT11 à l'extérieur sur une période prolongée. Afin de récupérer ces mesures sans rester branché à un ordinateur, nous avons pu nous servir du module Bluetooth Low Energy (BLE) de la carte Arduino.

La communication directe par BLE entre la carte Arduino et nos ordinateurs s'est révélée trop instable, nous poussant rapidement à mettre de côté cette approche.

Nous avons donc testé une solution avec deux cartes Arduino : l'une placée à l'extérieur avec le capteur, et la deuxième servant de relais vers l'ordinateur[1]. Bien que cette méthode ait permis d'automatiser la collecte, des soucis de connectivité survenaient sur les captures de longues durées.

Pour pallier ces problèmes de fiabilité, nous avons décidé de garder en mémoire sur la carte un historique des dernières mesures, pour pouvoir les récupérer même en cas de connexion instable. Par souci de simplicité et compte tenu des contraintes temporelles du projet, nous avons finalement récupéré manuellement cet historique de données depuis un téléphone via la connexion BLE. Ci-dessous, l'historique des données récupérées.



Pour compléter ce jeu de données, nous avons utilisé les données de l'api *Historical Weather,* fournie par open-meteo.com. Cette intégration nous a permis d'entraîner notre modèle sur des données plus représentatives, incluant des cycles journaliers avec de grands écarts de température ainsi que des données étalées sur différentes saisons. Celles-ci vont permettre à notre modèle de mieux comprendre les patterns journaliers et saisonniers.

2/ Implémentation du modèle

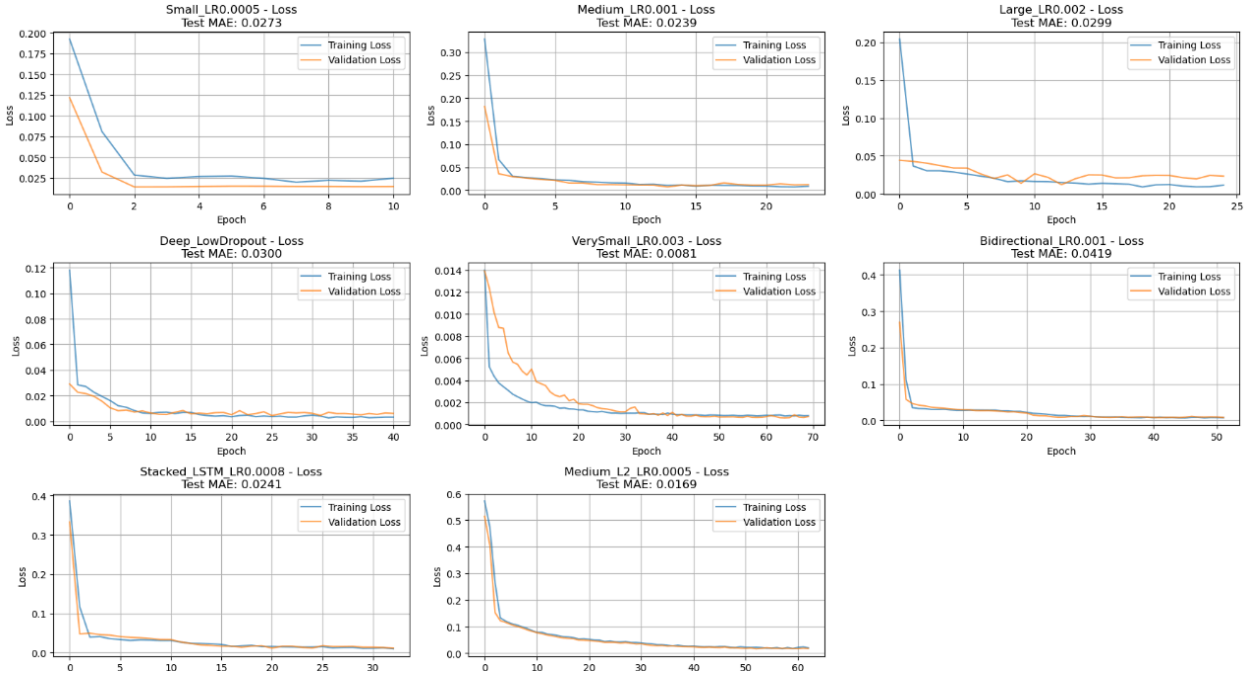
➢ Long Short-Term Memory (LSTM)

Dans un premier temps, nous avons trouvé pertinent de nous tourner vers un réseau de neurones LSTM (Long Short-Term Memory). Ce modèle est capable de mémoriser des informations sur le long terme et d'apprendre à reconnaître les liens entre des données qui se suivent dans le temps, ce qui le rend particulièrement adapté pour prédire l'évolution de données continues comme la température.

LSTM | Architecture :

Le LSTM est un type de *Recurrent Neural Network* (RNN). Il dispose, comme son nom l'indique, de deux mémoires, l'une à long terme et l'autre à court terme. Ainsi, il est capable de comprendre le cycle de températures journalier ainsi que les tendances en fonction de la période de l'année. Il répond à un problème important des RNN: la disparition du gradient. Chaque donnée prédite influe sur la mémoire d'une manière différente en fonction de sa valeur. Pour mettre à jour sa mémoire, le LSTM utilise des fonctions d'activation telles que *tanh* et *sigmoïde*.

Nous avons entraîné notre modèle en python en nous servant de *scikit-learn* et de *Tensorflow*. Nous avons comparé le modèle en faisant varier les paramètres et l'architecture dans le but d'avoir un modèle efficace. La figure ci-dessous montre la comparaison de valeurs de retour de la fonction de *loss*.

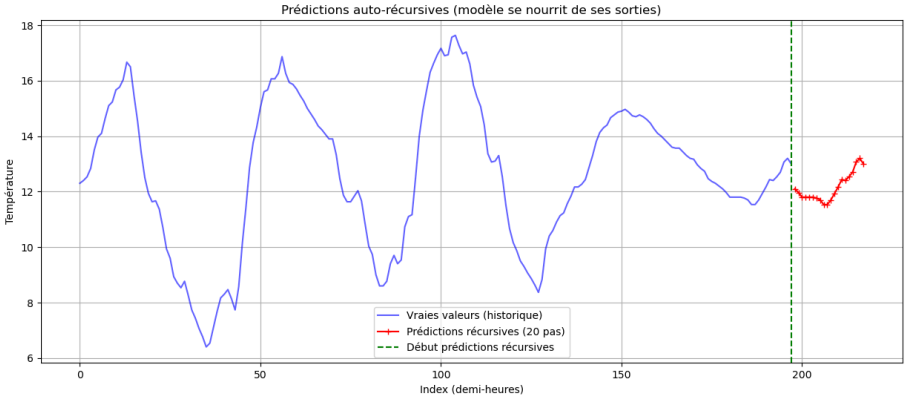


Lors de nos entraînements, nous avons remarqué qu'un paramètre impactait particulièrement le modèle : la taille des données en entrée. En effet, un input de moins de 24 heures empêche le modèle de comprendre le cycle journalier des températures. Par la suite, nous avons donc pris un input le plus petit possible (afin d'avoir un modèle léger) mais toujours plus grand que ce seuil de 24 heures.

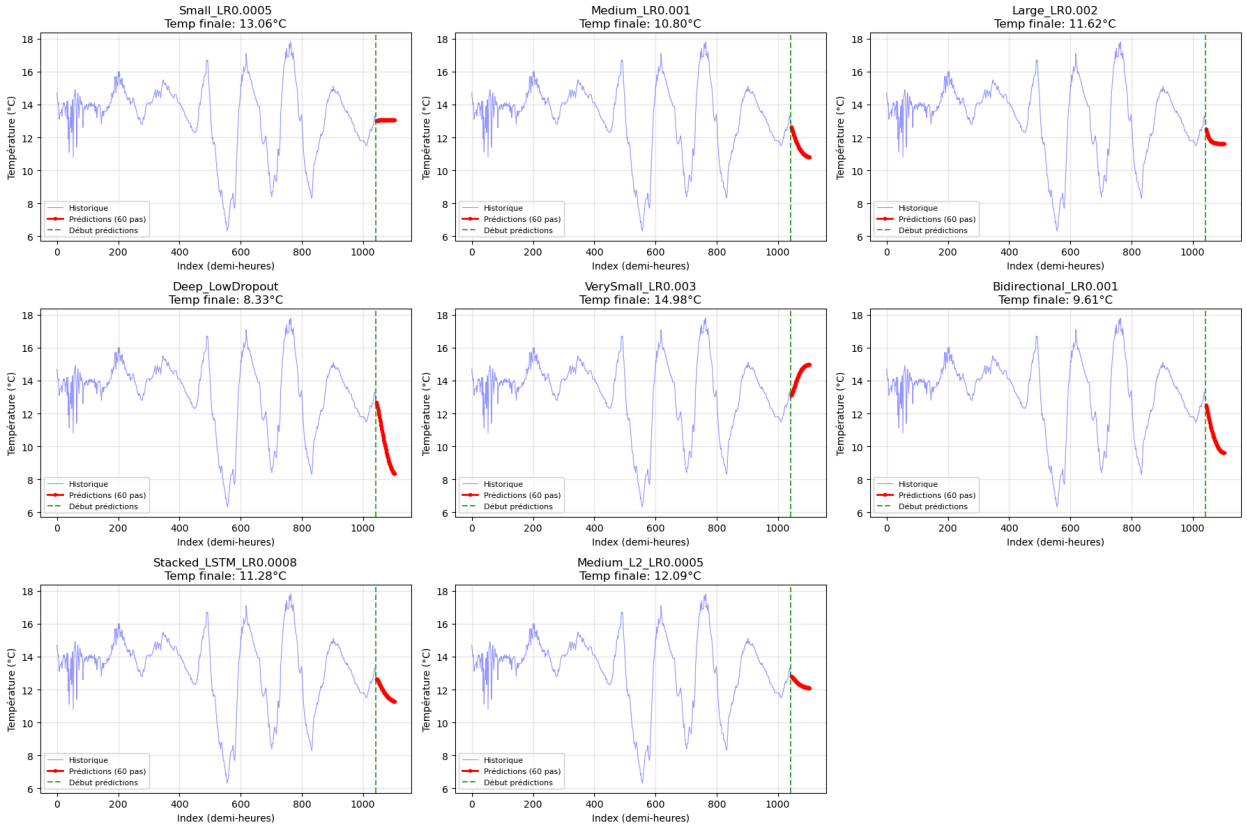
L'architecture finale est donc un modèle à 3 couches, dont une couche dense de 8 neurones munie d'une fonction d'activation *relu*. En tout, il y a environ 1300 paramètres. Une fois entraîné, le modèle fait 48 KB.

LSTM | Évaluation :

En cohérence avec l'usage que nous souhaitions faire de ce projet, nous avons mis en place des prédictions auto-récursives, de manière à ce que le modèle se nourrisse de ses propres sorties. C'est une manière de tester qui est propre aux modèles temporels : en se basant sur ses précédentes prédictions, le modèle accumule un biais qui est plus ou moins important. Cela nous permet de facilement classer les modèles entre eux. Ci-dessous, les prédictions auto-récursives du modèle.



Ce modèle est issu des neuf modèles implémentés. Ci-dessous, les prédictions auto-récursives des neuf modèles, avec un pas différent et donc différentes données prédites. Nous avons choisi le modèle central, car d'après nous, c'est le seul qui a été capable de comprendre le cycle journalier et d'afficher une température ascendante après le minimum, atteint à l'abscisse x=1010.



Note: les performances des modèles LSTM que nous avons développés sont assez limitées. Nous avons préféré nous assurer que le modèle fonctionnait bien sur la carte avant de chercher à en créer un plus efficace. Le modèle affiche un score $R^2 = 0.125$.

LSTM | Utilisation :

Une fois notre modèle entraîné et ses performances évaluées sur un autre ensemble de données, nous l'avons converti en fichier *.tflite*, que l'on peut ensuite importer sur la carte Arduino via le module *Tensorflow Lite Micro*.

Toutefois, nous nous sommes rendu compte que pour obtenir des résultats satisfaisants, notre réseau LSTM allait être trop lourd, et que certaines opérations n'étaient pas supportées par la carte. C'est pourquoi nous avons envisagé une alternative plus légère : le GRU.

➢ Gated Recurrent Unit (GRU)

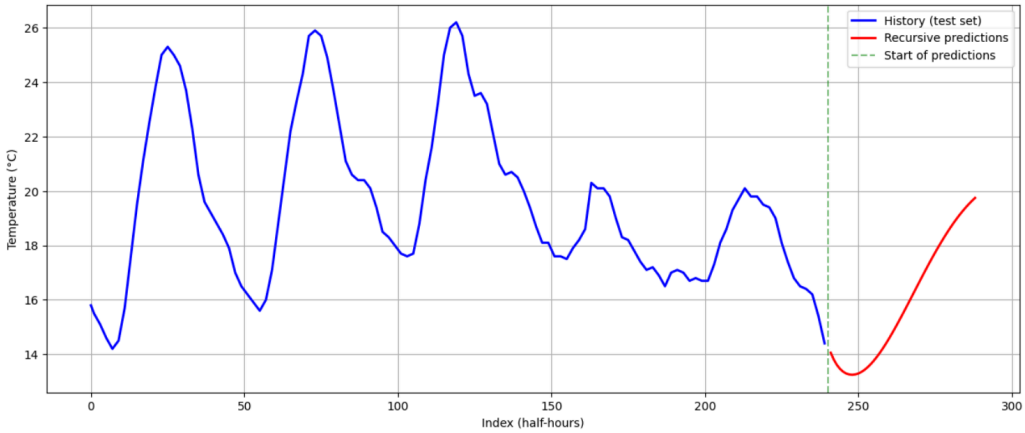
Le modèle GRU est très proche du LSTM, c'est d'ailleurs aussi un RNN. Il s'agit d'une version simplifiée : il est plus rapide à entraîner et plus léger en calcul, tout en conservant une performance proche. Cela le rend particulièrement adapté pour les microcontrôleurs.

GRU | Architecture :

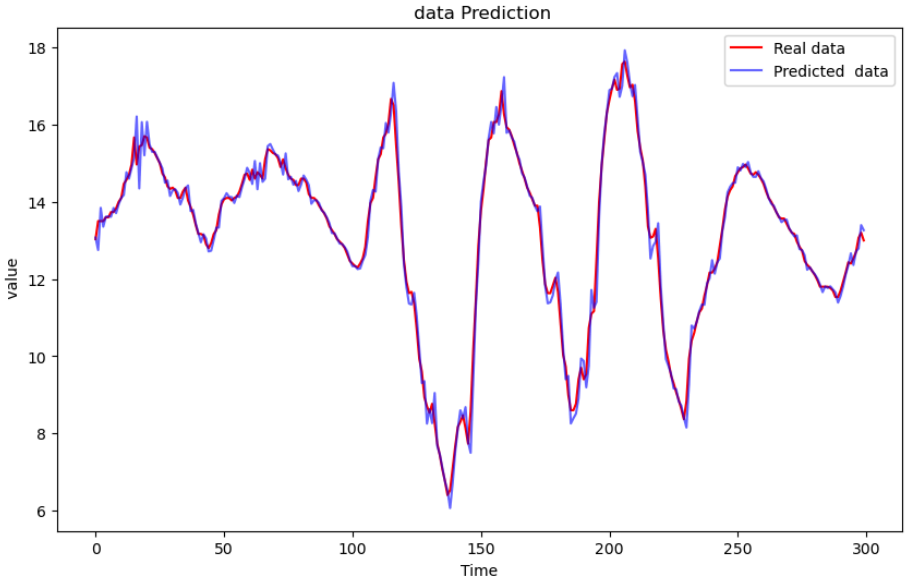
De la même manière que le LSTM, nous avons essayé plusieurs architectures avec différents paramètres. L'architecture finale est très similaire à celle du LSTM. Cependant, nous avons réduit par 2 le nombre de paramètres de la première couche, ce qui aura des conséquences visibles dans la partie suivante. Notre modèle comporte donc 400 paramètres, pour un poids total une fois entraîné de 31 KB.

GRU | Évaluation :

Après avoir entraîné le modèle, nous obtenions des résultats assez satisfaisants lors des tests sous Python. Ci-dessous, le graphique de prédictions auto-récursives du modèle GRU allégé. On remarque que le modèle réussit facilement à prédire des valeurs cohérentes avec la situation. Il a compris le pattern typique d'une journée.



La figure ci-dessous représente la différence entre les données réelles et les données prédites. Ce graphique est plus convaincant que le dernier car il se contente de prédire une seule température. On ne demande pas au modèle de se nourrir de ses précédentes prédictions. Cela nous a permis de calculer un score de corrélation entre les deux courbes $R^2 = 0.984$. Ce deuxième graphique est parfois trompeur quand on travaille avec des modèles de type RNN. En effet, si on crée un *"modèle"* qui prédit la température suivante comme étant égale à la température précédente, le résultat sera visuellement très satisfaisant, car les deux courbes auront exactement la même allure et le score de corrélation sera facilement meilleur qu'un modèle de prédiction moyen.



D'une manière générale, nous pouvons dire que le modèle n'affiche pas, comme le LSTM, un grand écart entre la dernière température et sa prédiction (écart entre la courbe bleue et la courbe rouge). De plus, il a su apprendre les motifs journaliers.

GRU | Utilisation :

Lors du téléversement du modèle sur la carte Arduino, nous avons constaté des différences importantes entre les prédictions obtenues en Python et celles générées en embarqué.

Après avoir comparé les paramètres et la quantification côté Python et Arduino, nous avons estimé que le problème venait probablement des différences d'implémentation des calculs internes par Tensorflow Lite sur Python et Arduino. En effet, lors d'une prédiction effectuée en python, les données sont normalisées entre 0 et 1. Nous avons affaire à des données réelles, les opérations effectuées sont donc naturellement adaptées. Cependant, la manière dont Tensorflow Lite manipule les données sur la carte Arduino diffère. En effet, on parle de *quantisation* entre -127 et 128. Nous devons donc travailler avec des entiers sur 8 bits. Par exemple, au lieu d'envoyer au modèle 0.4 normalisé dans l'intervalle [0, 1], nous envoyons un -25 normalisé dans l'intervalle [-127, 128]. Pour tenter d'expliquer ce problème, nous avons créé un modèle qui prend des données normalisées entre -127 et 128. La sortie n'était pas la même en local avec Tensorflow et sur la carte avec Tensorflow Lite. Nous en avons donc conclu que les opérations n'étaient pas réalisées de la même manière sur les deux environnements.

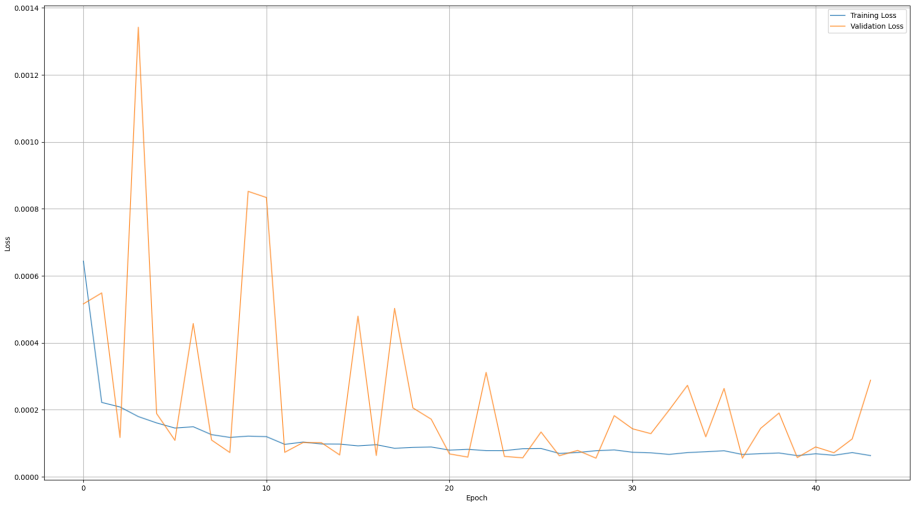
Face à ce problème de compatibilité, nous avons pensé à implémenter nous-même en Arduino les opérations nécessaires au fonctionnement du modèle GRU. Nous avons alors répertorié les opérations nécessaires au bon fonctionnement du modèle : il y en a 15 (ex : *sigmoïd*, *tanh*...). Faute de temps, nous avons préféré nous pencher vers une solution plus simple : le MLP.

➢ Multi-Layer Perceptron (MLP)

MLP | Architecture :

Un réel avantage lié à l'utilisation d'un modèle MLP est sa légèreté. En effet, après avoir travaillé avec des modèles sophistiqués, nous avons pu nous permettre d'ajouter une quatrième couche ainsi que d'augmenter le nombre de neurones. C'est ainsi que nous avons créé un modèle contenant 2250 paramètres, répartis en quatre couches et utilisant comme fonction d'activation *relu*. L'idée derrière ce modèle était de pouvoir l'importer sur la carte Arduino, nous avons donc privilégié des fonctions simples afin de simplifier l'implémentation. Le poids de ce modèle est de 6.4 KB, ce qui en fait le modèle le plus léger des trois.

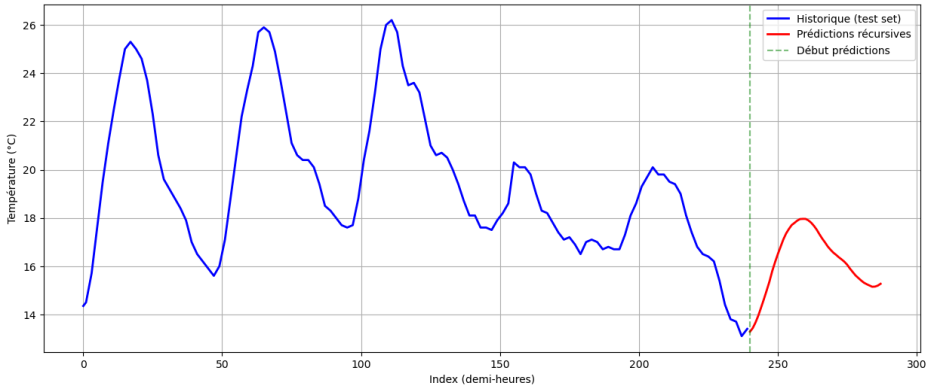
Ci-dessous le graphique d'entraînement du modèle MLP. On remarque que l'output de la fonction de loss pour les données de validations est plutôt hétérogène. Par exemple, la version du modèle aux époques supérieures à 42 est moins bonne que celle aux époques 26 à 28. Pour palier à ce problème, nous avons spécifié à la fonction *fit* d'arrêter l'entraînement si la valeur de *validation loss* remonte depuis un temps donné. De plus, le modèle en sortie est ramené à celui de l'époque pour laquelle le *validation loss* est le meilleur. En d'autres termes, on évite de l'entraîner trop longtemps de manière inutile et on restaure les meilleurs poids calculés pendant l'entraînement[4].



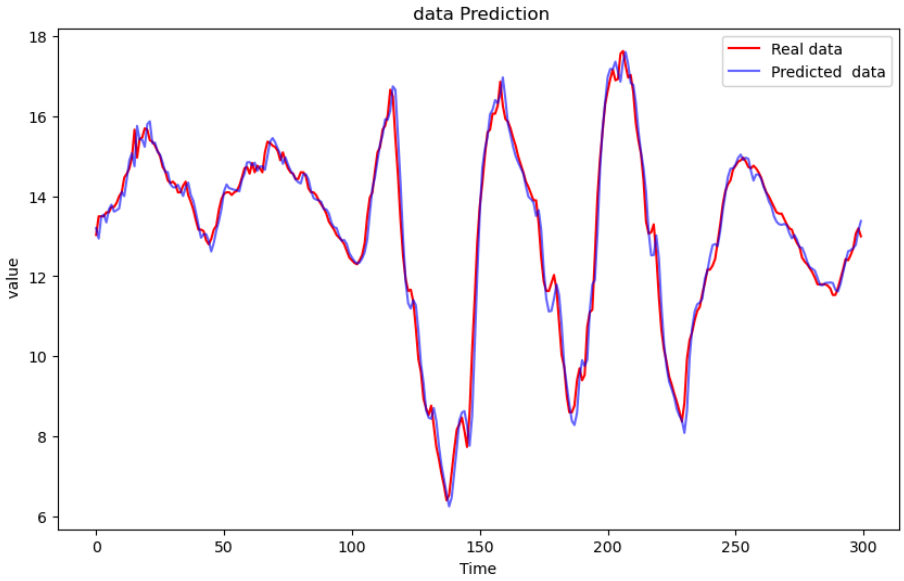
MLP | Évaluation :

Les résultats obtenus nous ont agréablement surpris. Comme nous pouvons le voir dans le graphique ci-dessous, le modèle a bien compris comment les températures évoluent au long de la journée. Cependant, le fait qu'il soit dépourvu d'une mémoire à long terme risquerait d'être un problème pour les saisons d'automne et de printemps, quand la température quotidienne est moins stable. Cela vient du fait que le modèle n'a qu'une faible connaissance de la mécanique des saisons. Il s'est entraîné sur un ensemble de données réparties sur 365 jours, les poids ont alors été calculés sur des journées très différentes, mais cela reste moindre par rapport à ce que propose la mémoire à long terme utilisée dans les deux autres modèles. Notons tout de même qu'en environnement tropical, là où

les saisons n'existent que très peu, la température quotidienne est plutôt stable. Cela laisse à penser que ce problème ne se posera pas dans ce type d'environnement.



Dans le graphique ci-dessous, nous remarquons que les données prédites sont à chaque moment de la journée proches de la réalité. La faiblesse principale de ce modèle concerne les extremums. On remarque bien que le modèle a parfois du mal à savoir quand la température va arrêter de monter ou, au contraire, quand elle arrête de diminuer. Nous pensons que cela vient du fait que le jour dure plus ou moins longtemps en fonction de la période de l'année, et donc les températures varient de manière différente. Ce problème est lui aussi lié à un manque de mémoire à long terme. Notons tout de même comme dans la dernière partie que dans un environnement tropical, où la différence de temps de jour entre l'été et l'hiver est minimale, ce problème ne se poserait pas.



Nous avons obtenu un score $R^2 = 0.977$, légèrement inférieur au GRU, mais tout de même très intéressant. C'est une belle manière de prouver qu'une solution simple est toujours un candidat à considérer.

MLP | Utilisation :

Notre dernier modèle était donc un perceptron multi-couche. Pour l'implémenter, nous avons recodé les fonctions de calculs du réseau neuronal directement en langage Arduino. Cette approche sans module complémentaire était risquée, mais nous offrait une capacité de compréhension accrue sur les calculs effectués par le modèle lors de la prédiction. Nous avons donc implémenté ces fonctions :

- *dense_layer*
- *scale_input*
- *inverse_scale_output*
- *model_feed_input*
- *Model_run*

Ce faisant, nous avons effectivement obtenu les mêmes prédictions que lorsque nous faisions tourner le modèle en local.

Conclusion

Pour conclure, ce projet nous a permis de mieux comprendre l'écosystème Arduino et les défis propres à l'informatique embarquée, en particulier ceux concernant les modèles d'IA. Les limitations de puissance et de poids nous ont invitées à réfléchir d'une autre manière aux divers modèles étudiés. Nous avons également pu approfondir notre connaissance du fonctionnement des modèles RNN et MLP, ainsi que les tâches auxquelles ils sont adaptés.

Ce projet nous a invités à collaborer de manière importante. En effet, nous partageons régulièrement nos avancées, points de vue et idées dans le but d'avoir un projet le plus abouti malgré les contraintes temporelles.

Nous aurions aimé approfondir plusieurs points : la collecte des données en la rendant complètement automatique, ajouter des intervalles de confiance sur la présentation des modèles, moins nous reposer sur des modules Arduino pour mieux comprendre comment la carte fonctionne. Ces trois points constituent les axes d'améliorations principaux à nos yeux.

D'une manière générale, ce projet aura été très enrichissant à la fois d'un point de vue technique et collaboratif. Nous sommes heureux du travail accompli.

Bibliographie

[1] Pili-Zhangqiu, *Wireless PC Communication with the Arduino Nano 33 Series*, GitHub.
Available at: https://github.com/pili-zhangqiu/Wireless-PC-Communication-with-the-Arduino-Nano-33-Series?tab=readme-ov-file

[2] Mistry, S., Pajak, D. and Siebeneicher, H. *Get Started With Machine Learning*, Docs.arduino.cc.
Available at : https://docs.arduino.cc/tutorials/nano-33-ble-rev2/get-started-with-machine-learning/

[3] Videvelop, *TensorFlow Lite Micro Library for Arduino*, GitHub.
Available at : https://github.com/tensorflow/tflite-micro-arduino-examples

[4] Keras, *Callbacks API*, Keras.io. Available at : https://keras.io/api/callbacks/