IA Embarquée : Étude et implémentation de TensorFlow Lite Micro sur les Microcontrôleurs

Encadrant: Patrice Kadionik
Arouch Salma - Tber Zakaria - Thierry Elouan - Msallem Maysa

Introduction

L'intelligence artificielle (IA) occupe une place croissante dans notre quotidien, avec des applications allant de l'assistance vocale à la reconnaissance d'images.. L'avenir de l'IA réside en grande partie dans son intégration dans des systèmes embarqués, permettant ainsi d'exécuter des modèles

intelligents directement sur des microcontrôleurs et des objets connectés, sans dépendre d'une infrastructure cloud. Cette évolution ouvre de nouvelles perspectives pour des systèmes autonomes, plus réactifs et surtout plus économes en énergie.

Dans ce contexte, notre projet, intitulé *IA Embarquée : Étude et implémentation de* TensorFlow Lit*e dans les microcontrôleurs*, s'inscrit dans une démarche pédagogique visant à explorer les capacités de l'IA embarquée. Initié par notre professeur, M.Kadionik, ce projet a pour objectif de développer plusieurs cas d'usage concrets qui pourront être réutilisés dans les travaux pratiques de la filière systèmes embarqués de notre école.

Nous avons ainsi mis en place trois cas d'étude illustrant des applications variées de l'IA embarquée :

- Détection de chute, un système capable d'identifier une chute à partir de données de capteurs.
- Détection de mouvement, qui vise à repérer des déplacements verticaux ou horizontaux.
- Détection de neige, permettant de reconnaître des conditions météorologiques enneigées.

Ces expériences nous ont permis d'explorer l'implémentation de modèles TensorFlow Lite sur des microcontrôleurs, d'analyser leurs performances et d'identifier les défis liés à l'optimisation des ressources dans des environnements contraints.

1 - Présentation de TensorFlow Lite et de l'IA embarquée

Un système embarqué présente plusieurs défis techniques pour l'implémentation de l'IA. L'un des principaux est la **capacité mémoire limitée** : contrairement aux serveurs et aux PC, les microcontrôleurs disposent de très peu de mémoire RAM et de stockage, ce qui impose de compresser et d'optimiser les modèles. Un autre défi majeur réside dans la **puissance de calcul réduite**. En effet, les processeurs embarqués offrent des performances bien inférieures à celles des CPU ou GPU des ordinateurs, nécessitant ainsi l'utilisation de techniques d'optimisation comme la **quantification** et la **simplification des architectures neuronales**.

Pour répondre à ces contraintes, **TensorFlow Lite (TFLite)** a été développé comme une version optimisée de TensorFlow, permettant l'exécution de modèles de machine learning sur des appareils à ressources limitées, tels que les microcontrôleurs et les objets connectés. Contrairement à la version complète de TensorFlow, conçue pour fonctionner sur des GPU et des processeurs puissants, **TFLite réduit l'empreinte mémoire et la consommation énergétique**. L'exécution d'un modèle avec **TFLite** repose sur deux étapes principales :

1. **Conversion du modèle** : un modèle entraîné avec TensorFlow est converti en un format optimisé (*FlatBuffer*) pour être utilisé sur un microcontrôleur.

2. **Exécution du modèle** : le modèle est déployé sur l'appareil cible, où un interpréteur optimisé le fait fonctionner, avec la possibilité d'utiliser des accélérations matérielles si disponibles.

TensorFlow Lite intègre plusieurs optimisations, notamment la **quantification**, qui réduit la taille du modèle en diminuant la précision des poids (par exemple, conversion de *float32* en *int8*), améliorant ainsi les performances sur des architectures embarquées.

Dans le contexte de l'IA embarquée, il est essentiel de différencier **TensorFlow Lite (TFLite)** et **TensorFlow Lite Micro (TFLM)**, chacun étant destiné à des types d'appareils spécifiques :

- **TFLite** est conçu pour des systèmes embarqués plus puissants, dotés d'un système d'exploitation (ex. Raspberry Pi, smartphones, microprocesseurs).
- **TFLM** est adapté aux **microcontrôleurs ultra-contraints**, avec une gestion mémoire statique et une empreinte minimale, ne nécessitant ni OS ni allocation dynamique de mémoire.

Dans notre projet, **puisque nous travaillons avec une Arduino**, nous utilisons **TensorFlow Lite Micro**, qui est spécifiquement conçu pour exécuter des modèles avec des ressources extrêmement limitées.

2 - Matériel et environnement de développement

Dans ce projet, nous avons utilisé la carte Arduino Nano 33 BLE Sense Rev 2, un microcontrôleur conçu pour des applications embarquées. Cette carte est équipée d'un processeur nRF52840 basé sur un Cortex-M4F ainsi que de plusieurs capteurs intégrés, notamment un accéléromètre, un gyroscope, un capteur de température, un capteur de pression et un microphone, ce qui la rend particulièrement adaptée aux tâches d'IA embarquée.

Pour développer et déployer nos modèles sur l'Arduino, nous avons utilisé plusieurs outils :

- Google Colab : employé par certains membres du groupe pour entraîner et tester les modèles de machine learning en Python, grâce à son accès facile aux ressources GPU.
- Spyder : utilisé par d'autres pour le développement en Python, notamment pour le prétraitement des données et les expérimentations avec TensorFlow.
- Arduino IDE : utilisé pour programmer la carte et charger les modèles optimisés en C++.

Ce choix d'outils nous a permis de concilier l'entraînement des modèles sur des machines performantes avec leur exécution sur un microcontrôleur aux ressources limitées.

3 - Méthodologie pour entraîner et convertir les modèles en format TensorFlow Lite

L'entraînement et la conversion des modèles en **TensorFlow Lite (TFLite)** nécessitent plusieurs étapes, allant de la préparation des données à l'optimisation du modèle pour son exécution sur un microcontrôleur.

A - Préparation des données

Avant d'entraîner un modèle, il est essentiel de collecter et de prétraiter les données pour qu'elles soient exploitables par un réseau de neurones.

- **Collecte des données** : Former sa base de données
- **Nettoyage et normalisation** : Les données brutes ont été filtrées et mises à l'échelle pour améliorer l'apprentissage.
- **Segmentation et étiquetage** : Nous avons divisé les données en séquences temporelles et attribué des étiquettes correspondant aux événements à détecter.

B - Entraînement du modèle

L'entraînement s'est déroulé sur **Google Colab** ou **Spyder**, en utilisant **TensorFlow/Keras** :

1. **Définition du modèle** : Nous avons conçu des architectures légères adaptées aux contraintes du microcontrôleur, par exemple, des réseaux de neurones convolutifs (CNN)..
2. **Compilation et entraînement** : Le modèle a été entraîné avec la fonction de coût **CrossEntropyLoss** et l'optimiseur **Adam**, en utilisant des ensembles de données collectées ou disponibles en open source.
3. **Validation** : Une évaluation a été réalisée pour mesurer la précision et éviter le surapprentissage (*overfitting*).

C - Conversion du modèle en format TensorFlow Lite

Une fois le modèle entraîné et validé, il a été converti en format **TensorFlow Lite Micro (TFLM)**, adapté aux microcontrôleurs :

Conversion en format FlatBuffer : Le modèle est compressé et enregistré en un fichier *.tflite*.

Lors de la conversion, nous avons appliqué une technique de **quantification post-entraînement**. Cette approche consiste à réduire la précision des poids du modèle, en passant de **float32** à **int8**, ce qui réduit considérablement la taille du modèle. La quantification améliore également les performances en optimisant les calculs, ce qui est essentiel pour les architectures à faible puissance de calcul, comme celles des microcontrôleurs utilisés dans ce projet.

Ainsi, cette conversion permet d'adapter le modèle à des contraintes de mémoire et de puissance tout en maintenant une performance acceptable pour l'inférence sur l'Arduino. L'inférence est l'application d'un modèle pour faire des prédictions.

D - Déploiement sur l'Arduino

Le fichier **model.tflite** a été intégré dans un programme **C++** sous **Arduino IDE**, en utilisant la bibliothèque **TensorFlow Lite for Microcontrollers**. L'inférence a été réalisée en exploitant les capteurs embarqués et en effectuant des prédictions en temps réel.

Cette méthodologie nous a permis d'**entraîner des modèles en Python**, puis de les **convertir et les exécuter efficacement sur un microcontrôleur**, malgré ses ressources limitées.

4 - Détection de chute

L'objectif principal de ce projet est de développer un système de détection de chutes pour les personnes âgées, en utilisant l'intelligence artificielle (IA) embarquée sur un microcontrôleur Arduino

Nano 33 BLE Sense. Ce dispositif est conçu pour détecter automatiquement les chutes en temps réel, afin d'alerter les proches ou les services d'urgence, améliorant ainsi la sécurité des individus, en particulier des personnes âgées qui sont plus vulnérables aux risques de chutes.

Le projet repose sur l'utilisation des capteurs de mouvement intégrés dans le microcontrôleur, comme l'accéléromètre et le gyroscope, pour analyser les données relatives aux mouvements du corps. Ces données sont ensuite traitées localement par un modèle d'IA optimisé pour le microcontrôleur, permettant ainsi une détection rapide et efficace des chutes, tout en garantissant une faible consommation d'énergie et une réponse en temps réel.

Le développement de ce système représente une avancée importante dans l'application des technologies embarquées pour améliorer le quotidien et la sécurité des personnes âgées.

A - Base de données

Dans cette première étape du projet, le but était de collecter des données sur les chutes et les mouvements normaux à l'aide de l'Arduino Nano 33 BLE Sense.Le code utilisé pour collecter les données de chutes et de mouvements normaux se trouve dans le fichier base_de_donnees. Ce code a été développé et exécuté dans l'Arduino IDE, un environnement de développement intégré qui permet de programmer et de communiquer avec l'Arduino Nano 33 BLE Sense.

L'Arduino Nano 33 BLE Sense, équipé d'un accéléromètre et d'un gyroscope, a été utilisé pour enregistrer les données de mouvement. Lors des tests de chute, des mouvements brusques vers le bas ont été réalisés pour simuler les chutes. Les données d'accélération (mesurées en G) et de gyroscope (mesurées en degrés par seconde) ont été capturées à chaque mouvement et stockées dans un fichier appelé fall.txt.

Parallèlement, des mouvements normaux ont également été enregistrés, comme la marche, les flexions ou des rotations, en reproduisant des actions courantes. Les données correspondantes ont été sauvegardées dans un fichier normal.txt. Ces deux ensembles de données, chutes et mouvements normaux, constituent la base de données qui sera utilisée pour entraîner et tester le modèle de détection de chutes.

B - Prétraitement de la base de données

Une fois les données collectées à partir des mouvements simulés de chutes et des mouvements normaux, l'étape suivante consiste à préparer ces données pour l'entraînement du modèle de détection de chutes.

Cette préparation inclut le chargement des fichiers de données, l'ajout des étiquettes appropriées pour chaque type de mouvement, ainsi que la division des données en échantillons de taille fixe. Ces échantillons sont ensuite traités et normalisés afin d'être prêts pour l'entraînement et le test du modèle de machine learning. La gestion des classes déséquilibrées est également prise en compte, et si nécessaire, une technique de rééchantillonnage est appliquée pour garantir des résultats fiables. Enfin, les données sont divisées en ensembles d'entraînement et de test, permettant de valider le modèle de manière objective et efficace.

Pour mieux comprendre le processus de prétraitement, voici une explication détaillée du code qui a permis de charger, traiter et préparer les données pour l'entraînement du modèle.

- Chargement des fichiers de données : On commence par charger les fichiers fall.txt et normal.txt contenant respectivement les données de chutes et de mouvements normaux à l'aide de la fonction

pd.read_csv de la bibliothèque Pandas. Ces fichiers sont lus sans en-tête (header=None), car ils contiennent uniquement les données numériques.

- Ajout des labels : Chaque ligne des données est étiquetée pour indiquer s'il s'agit d'une chute ou d'un mouvement normal. Pour cela, une nouvelle colonne label est ajoutée, avec 'fall' pour les chutes et 'normal' pour les mouvements normaux.

- Combinaison des données : Les deux jeux de données (chutes et mouvements normaux) sont ensuite combinés dans un seul DataFrame à l'aide de pd.concat, ce qui permet d'avoir toutes les données dans une seule structure pour faciliter le traitement ultérieur.

- Division des données en échantillons : Une fonction appelée split_data_by_length est définie pour diviser les données en échantillons de taille fixe (120 lignes par échantillon. La taille d'échantillon de 120 a été choisie car c'est le nombre de données collectées à chaque expérience avec l'Arduino IDE, où chaque séquence de mouvement (chute ou mouvement normal) était enregistrée sur 120 points de données.

- Vérification du nombre d'échantillons : Après la division, le code vérifie qu'il y a suffisamment d'échantillons pour pouvoir procéder à l'entraînement et au test. Si le nombre d'échantillons est insuffisant, il soulève une erreur.

- Préparation des données pour l'entraînement : La fonction prepare_data est ensuite utilisée pour préparer les données pour l'entraînement du modèle :

- Les données sont extraites et transformées en matrices numpy, et les labels sont convertis en valeurs binaires (1 pour les chutes et 0 pour les mouvements normaux).
- La répartition des labels est vérifiée pour s'assurer que les classes sont équilibrées.
- Si les classes sont déséquilibrées, la technique SMOTE (Synthetic Minority Over-sampling Technique) est utilisée pour rééquilibrer les données.
- Les données sont ensuite divisées en ensembles d'entraînement et de test, avec 20 % des données utilisées pour le test.
- Une normalisation est appliquée sur les données d'entraînement et de test à l'aide du StandardScaler.
- Enfin, les données sont reformattées pour être compatibles avec un réseau de neurones convolutif (Conv1D).

C - Création du modèle CNN

Après avoir préparé et prétraité les données, le modèle CNN (Convolutional Neural Network) est créé pour détecter les chutes. Ce modèle est composé de plusieurs couches, chacune ayant un rôle spécifique pour extraire et analyser les caractéristiques des données de capteurs.

- **Conv1D** : La première couche est une couche convolutive 1D qui applique 4 filtres avec une taille de noyau de 3 et une activation ReLU. Elle permet d'extraire des motifs importants des séquences temporelles d'accélération et de gyroscope.
- **MaxPooling1D** : Cette couche de pooling réduit la taille des données tout en conservant les informations essentielles, permettant au modèle de se concentrer sur les caractéristiques les plus pertinentes.
- **Flatten** : La couche d'aplatissement transforme les données en un vecteur unidimensionnel, rendant les caractéristiques prêtes à être utilisées par les couches denses.
- **Dense** : Une couche dense avec 4 neurones, activée par ReLU, est ajoutée pour renforcer l'apprentissage des relations complexes entre les caractéristiques extraites.
- **Dense finale** : La couche de sortie, composée d'un seul neurone avec une activation sigmoïde, permet de classer les données en deux catégories : "chute" ou "normal".

Ce modèle a été conçu en choisissant des fonctions compatibles avec TensorFlow Lite, pour garantir une bonne précision tout en gardant une taille de modèle réduite. L'objectif était de ne pas alourdir le modèle, car il sera exécuté sur un Arduino, qui a des ressources très limitées. Ainsi, l'architecture permet d'obtenir des résultats performants sans augmenter excessivement la taille du modèle, afin de garantir une exécution fluide sur un microcontrôleur avec peu de mémoire et de puissance de traitement.

D - Compilation et Entraînement du Modèle

Une fois le modèle défini, il doit être compilé et entraîné pour pouvoir effectuer la détection de chutes.

Compilation du modèle :

Le modèle est compilé avec l'optimiseur Adam, qui est un choix populaire pour les réseaux neuronaux grâce à son efficacité et sa capacité à ajuster les taux d'apprentissage pendant l'entraînement. Le taux d'apprentissage est fixé à 0.001. La fonction de perte choisie est binary_crossentropy, adaptée aux problèmes de classification binaire (chute ou normal). La précision (accuracy) est utilisée comme métrique pour évaluer les performances du modèle pendant l'entraînement.

Entraînement du modèle :

Le modèle est ensuite entraîné pendant 100 époques avec un batch size de 64. L'ensemble d'entraînement (X_train, y_train) est utilisé pour ajuster les poids du modèle, tandis que l'ensemble de test (X_test, y_test) est utilisé pour valider les performances du modèle à chaque époque.

E - Prédictions et Matrice de Confusion

Une fois l'entraînement du modèle terminé, il est important d'évaluer ses performances en utilisant les prédictions sur l'ensemble de test et en générant une matrice de confusion.

Prédictions : Le modèle effectue des prédictions sur l'ensemble de test (X_test) avec la méthode predict. Cela génère des probabilités de chute pour chaque échantillon. Ces probabilités sont ensuite converties en classes binaires en utilisant un seuil de 0.5, où les valeurs supérieures à 0.5 sont classées comme des "chutes" (1), et les autres comme "normales" (0). Cela permet de déterminer si le modèle a correctement classé chaque échantillon comme une chute ou un mouvement normal.

Matrice de confusion : La matrice de confusion est générée à l'aide de la fonction confusion_matrix, qui compare les étiquettes réelles (y_test) et les classes prédites (y_pred_classes). La matrice de confusion permet d'observer la performance du modèle, en montrant combien de prédictions sont correctes et incorrectes, ainsi que les types d'erreurs spécifiques (faux positifs et faux négatifs).

Le modèle atteint une précision de 92 %, ce qui indique une bonne performance pour la détection des chutes, avec un taux élevé de classifications correctes sur l'ensemble de test.

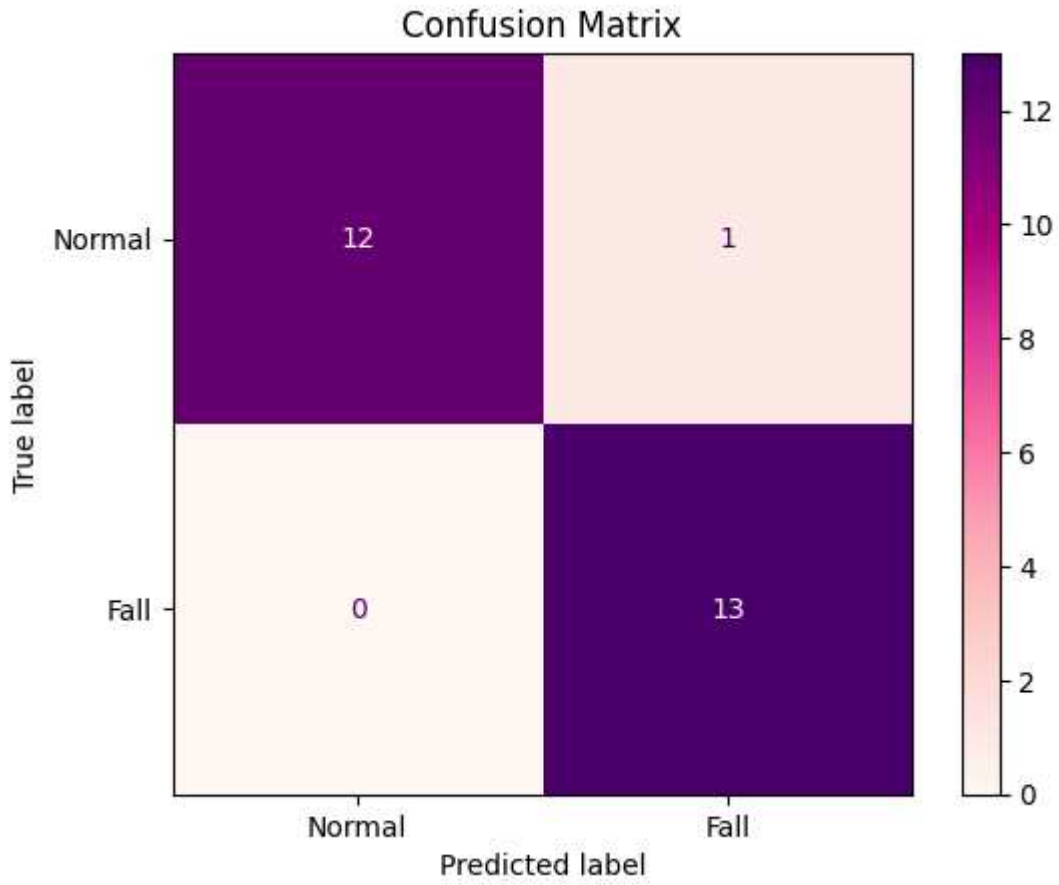


Figure 1 : matrice de confusion

Les résultats obtenus montrent que le modèle atteint une précision de 92 % sur l'ensemble de test, ce qui indique une excellente capacité à détecter les chutes, avec un faible taux d'erreurs. La matrice de confusion a permis d'évaluer la performance du modèle en mettant en évidence le nombre de prédictions correctes et incorrectes, ainsi que les erreurs spécifiques. Avec ces résultats, nous pouvons désormais passer à l'implémentation du modèle sur la carte embarquée, où il sera utilisé pour effectuer des prédictions en temps réel sur des données réelles.

F - Génération du fichier .tflite

Dans notre projet, nous avons utilisé **TensorFlow Lite (TFLite)** pour déployer un modèle d'apprentissage profond sur un système embarqué. Après avoir entraîné le modèle avec TensorFlow, nous l'avons converti au format TFLite afin de l'optimiser pour une exécution en temps réel sur du matériel à ressources limitées. Pour améliorer encore l'efficacité, nous avons appliqué **la quantification**, qui réduit la taille du modèle et les besoins en calculs en convertissant les poids de 32 bits en entiers de 8 bits. Cette optimisation permet au modèle de fonctionner plus rapidement tout en consommant moins de mémoire, le rendant ainsi plus adapté aux applications embarquées. Enfin, nous avons sauvegardé le modèle quantifié sous le nom "fall_model_quantized.tflite" et vérifié sa structure en le chargeant.

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
```

```
converter.optimizations = [tf.lite.Optimize.DEFAULT]

converter.target_spec.supported_ops =
[tf.lite.OpsSet.TFLITE_BUILTINS]

tflite_quantized_model = converter.convert()

# Save the quantized model to disk

with open("fall_model_quantized.tflite", "wb") as f:

    f.write(tflite_quantized_model)
```

G - Conversion d'un fichier .tflite en fichier .h à l'aide de la commande xxd

Dans cette étape, nous avons converti le fichier **.tflite** contenant notre modèle en un fichier **.h** pour l'intégrer directement dans notre projet embarqué. Cette conversion a été réalisée en utilisant la commande xxd, qui permet de transformer les données binaires du modèle en un format hexadécimal compatible avec le langage C. Le fichier **model.h** contient désormais le modèle sous forme d'un tableau de type unsigned char, ce qui permet de l'inclure directement dans le code source du projet sans nécessiter de fichiers externes. Cette méthode est particulièrement utile pour des applications embarquées où il est nécessaire de charger rapidement des modèles tout en réduisant la dépendance à des fichiers externes.

```
echo "const unsigned char model[] = {" > model.h
cat gesture_model.tflite | xxd -i      >> model.h
echo "};"                              >> model.h
```

H - Resultats obtenu

Les résultats obtenus lors de l'implémentation du modèle sur la carte embarquée montrent que celle-ci réussit efficacement à prédire les chutes de manière précise en temps réel. Grâce à l'intégration du modèle quantifié, la carte est capable de détecter rapidement les chutes, avec un faible taux de latence, tout en maintenant une grande précision. Les prédictions effectuées sur des données réelles ont confirmé que le système réagit correctement aux mouvements, distinguant avec succès les chutes des mouvements normaux. Cette performance souligne la capacité de la carte embarquée à traiter les informations et à fournir des résultats fiables, même avec des ressources matérielles limitées.

5 - Détection de mouvement

A - Explication du cas d'usage

Dans ce cas d'usage, nous cherchons à détecter et classifier des gestes en temps réel à partir de données collectées par un capteur analogique. L'objectif est d'entraîner un modèle de réseau de neurones capable de reconnaître différents types de mouvements en fonction des variations de la valeur analogique capturée.

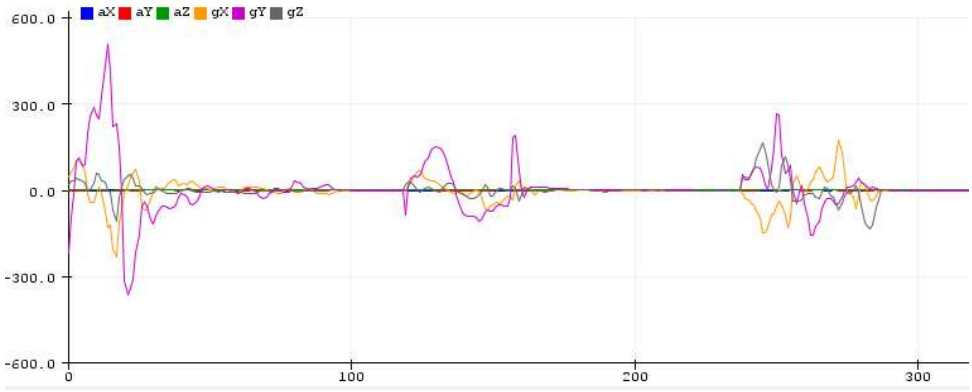
Les gestes pris en compte dans cette étude sont les suivants :

- **Punch** : mouvement rapide et brusque simulant un coup de poing.
- **Flex** : mouvement plus lent simulant une flexion du poignet.

Une fois le modèle d'intelligence artificielle entraîné, il pourra être exporté et intégré dans un système embarqué capable de classifier ces gestes en temps réel, ouvrant ainsi la voie à de potentielles applications interactives.

B - Préparation du jeu de données

1. Collecte des données



Les données ont été collectées à l'aide d'un capteur analogique connecté à une carte Arduino. Pour chaque type de geste, un ensemble de mesures a été enregistré, chaque enregistrement contenant une séquence de valeurs analogiques capturées à une fréquence constante.

Deux fichiers CSV ont été générés, un pour chaque type de geste :

- **punch.csv**
- **flex.csv**

Chaque fichier contient 50 enregistrements correspondant à 50 exécutions distinctes du même geste, chacune constituée de 119 valeurs analogiques.

2. Normalisation et transformation des données

Les données brutes sont ensuite prétraitées afin d'assurer une meilleure convergence du modèle d'apprentissage. Les étapes de prétraitement sont les suivantes :

- **Normalisation** : les valeurs analogiques sont mises à l'échelle entre 0 et 1 afin d'assurer une cohérence dans les entrées du modèle.
- **Construction des entrées et sorties** : chaque enregistrement est converti en un vecteur de 119 valeurs, et chaque geste est associé à une sortie encodée en one-hot.
- **Mélange et partitionnement** : les données sont mélangées aléatoirement puis divisées en trois ensembles distincts :
    - 60% pour l'entraînement
    - 20% pour les tests
    - 20% pour la validation

C - Entraînement du modèle

Le modèle de réseau de neurones a été construit en utilisant TensorFlow et se compose de plusieurs couches :

- Une couche dense de 50 neurones avec activation **ReLU**.
- Une couche dense de 15 neurones avec activation **ReLU**.
- Une couche de sortie comportant autant de neurones que de classes de gestes, avec activation **softmax**.

Le modèle a été compilé avec l'optimiseur **RMSprop** et la fonction de perte **MSE (Mean Squared Error)**. L'entraînement a été réalisé sur **600 époques** avec une taille de batch de **1**.

Après entraînement, le modèle a atteint un niveau de précision satisfaisant, permettant de classifier les gestes avec un faible taux d'erreur.

D - Conversion et exportation du modèle

1. Conversion du modèle

Une fois l'entraînement terminé, le modèle est converti au format TensorFlow Lite (.tflite) afin d'être exécuté sur un microcontrôleur. Cette conversion est réalisée à l'aide de TensorFlow Lite Converter, qui optimise le modèle pour une exécution sur des systèmes à ressources limitées.

Pour rendre le modèle directement utilisable dans un programme Arduino, il est ensuite converti en un fichier header C (model.h), où les poids et les paramètres du modèle sont stockés sous forme de tableau binaire. Ce fichier est inclus dans le projet Arduino afin que le microcontrôleur puisse effectuer des inférences en temps réel.

Le poids du fichier final du modèle est de X Ko (valeur à confirmer après test), ce qui le rend adapté aux systèmes embarqués disposant de peu de mémoire.

2. Implémentation sur Arduino

a) Matériel et bibliothèques utilisées

L'implémentation est réalisée sur une Arduino Nano 33 BLE Sense Rev2, équipée du capteur BMI270+BMM150 qui permet de mesurer l'accélération et la rotation. Plusieurs bibliothèques sont nécessaires :

- Arduino_BMI270_BMM150.h : pour récupérer les données du capteur IMU.
- TensorFlow Lite for Microcontrollers : pour exécuter le modèle de reconnaissance des gestes.

- model.h : fichier contenant le modèle converti.

b) Chargement du modèle et initialisation

Lors du démarrage de l'Arduino (setup()), plusieurs étapes sont réalisées :

1. Initialisation des LEDs :
   - Une LED rouge et une LED verte permettent d'afficher le résultat de la classification.
   - Elles sont éteintes au démarrage.
2. Vérification de l'IMU :
   - Si l'IMU ne s'initialise pas, les deux LEDs s'allument en guise d'alerte et le programme s'arrête.
3. Chargement du modèle TensorFlow Lite :
   - model.h est chargé en mémoire.
   - L'interpréteur TensorFlow Lite est initialisé avec une mémoire tampon de 8 Ko (tensorArena).
   - Les tenseurs d'entrée et de sortie sont configurés.
   - Si le modèle est incompatible avec la version de TensorFlow Lite embarquée, la LED rouge s'allume et le programme s'arrête.

c) Détection des gestes en temps réel

Une fois l'Arduino opérationnel, la boucle principale (loop()) exécute le processus de classification en temps réel :

1. Détection d'un mouvement significatif
   - L'accélération est mesurée en continu.
   - Si la somme des valeurs absolues des axes X, Y et Z dépasse 2.5 g, la collecte des données commence.
2. Acquisition et normalisation des données
   - 119 échantillons successifs sont collectés.
   - Les données sont normalisées pour correspondre à l'échelle d'entraînement du modèle.
3. Exécution du modèle
   - Une fois la séquence complète, le modèle est exécuté via Invoke().
   - Les probabilités des gestes sont récupérées dans le tenseur de sortie.
4. Interprétation des résultats et affichage
   - Si "punch" est détecté, la LED verte s'allume.
   - Si "flex" est détecté, la LED rouge s'allume.
   - Les scores de classification sont affichés sur le moniteur série.

```
Le punch score est: 0.00
Le flex score est: 1.00
Le punch score est: 1.00
Le flex score est: 0.00
Le punch score est: 0.04
Le flex score est: 0.96
```

e) Résultats et perspectives

Grâce à cette implémentation, la classification des gestes est réalisée en temps réel sur l'Arduino. L'inférence est rapide et permet une reconnaissance efficace des mouvements "punch" et "flex".

Des améliorations possibles incluent :

- Ajouter d'autres gestes au modèle pour plus de polyvalence.
- Optimiser la normalisation des données pour améliorer la précision.
- Réduire la taille du modèle pour minimiser la consommation mémoire.

Ce projet a permis de développer un modèle de détection de gestes basé sur des données issues d'un capteur analogique. Les résultats obtenus sont encourageants et ouvrent des perspectives pour une intégration dans des dispositifs interactifs, notamment pour des applications dans la robotique, la réalité augmentée ou encore le domaine médical.

Des améliorations futures pourraient inclure :

- L'ajout de nouveaux gestes pour enrichir la classification.
- L'expérimentation avec d'autres architectures de réseaux de neurones pour optimiser les performances.
- La mise en place d'une quantification du modèle pour réduire sa taille et accélérer l'inférence sur microcontrôleur.

6 - Détection de neige

A - Explication du cas d'usage.

Dans ce cas d'usage, nous allons chercher détecter s' il neige ou pas en temps réel. Nous utiliserons le capteur de température et d'humidité de la carte Arduino.

B - Préparation du jeu de donnée

Pour collecter des données météorologiques horaires, nous avons utilisé WorldWeatherOnline. Ce site propose des données météorologiques historiques à l'échelle mondiale. Pour ce faire, nous avons importé la bibliothèque **wwo-hist**. Il faudra veiller à installer la version de pandas compatible avec cette bibliothèque (pandas==1.3.5). Ensuite,à l'aide de la fonction **retrieve_hist_data**, nous avons exporté les données dans l'objet **hist_df**, qui est une liste de DataFrames pandas. Cette fonction nous a permis de choisir le lieu et l'intervalle de temps des données météorologiques. Ainsi, au départ, nos données sont sous la forme d'un tableau où il y a la date, l'humidité, la température, la quantité de neige en centimètre.

À la différence des deux autres cas d'usage, le jeu de données n'est pas encore classé. Par conséquent, la première chose à faire est de mapper les chutes de neige en cm dans deux classes : "Oui, il neige", "Non, il ne neige pas". Dans ce projet, nous avons considéré la formation de neige uniquement lorsque la chute de neige en cm est supérieure à 5 cm.

L'étape suivante pour construire le jeu de données concerne la sélection des entrées du réseau de neurones pour prédire la neige. Nous avons choisi les valeurs de température et d'humidité des trois heures précédentes comme caractéristiques d'entrée. Par exemple, si t0 est l'heure actuelle, les valeurs stockées dans le jeu de données sont les suivantes :

- Temp0/Humi0 : Température/humidité à t = t0 - 2h
- Temp0/Humi0 : Température/humidité à t = t0 - 1h
- Temp0/Humi0 : Température/humidité à t = t0
- Snow : Étiquette indiquant si il va neiger à t = t0

Ensuite, nous avons équilibré le jeu de données car un jeu de données déséquilibré est un jeu de données où une des classes possède beaucoup plus d'exemples que les autres. L'entraînement avec un jeu de données déséquilibré pourrait produire un modèle avec une haute précision mais qui serait incapable de résoudre notre véritable problème. Pour équilibrer un jeu de données on met à peu près le même nombre d'étiquettes pour chaque classe.

Une fois que nous avons équilibré le jeu de données, il nous reste à mettre les caractéristiques d'entrée dans une plage numérique similaire. En effet, nos caractéristiques d'entrée, l'humidité et la température des trois dernières heures, se trouvent dans différentes plages numériques. Par exemple, l'humidité est comprise entre 0 et 100, tandis que la température sur l'échelle Celsius peut être négative et avoir une plage numérique plus petite que celle de l'humidité. Ainsi, le modèle ML peut être influencé davantage par celles ayant des valeurs plus grandes, ce qui peut affecter son efficacité. Par conséquent, les caractéristiques d'entrée doivent être mises à l'échelle pour garantir que chaque caractéristique contribue de manière égale lors de l'entraînement. Dans ce projet, nous utiliserons la technique de normalisation par Z-score.

C - Modèle utilisé

Le modèle conçu pour prédire la neige est un classifieur binaire simple basé sur un réseau de neurones. Le réseau comprend les couches suivantes :

1. Une couche entièrement connectée avec 12 neurones, suivie d'une fonction d'activation ReLU
2. Une couche de dropout avec un taux de 20 % (0.2) pour éviter le surapprentissage
3. Une couche entièrement connectée avec un seul neurone de sortie, suivie d'une fonction d'activation sigmoïde

Étant donné que la fonction sigmoïde produit la sortie, le résultat est compris entre **0 et 1**, et il est classé comme **Non** lorsque la valeur est inférieure à 0,5, et comme **Oui** sinon.

D - Déploiement du modèle sur microcontrôleur et Analyse des performances.

Pour le déploiement du modèle sur microcontrôleur et l'analyse des performances, nous avons suivi le même processus que pour les autres cas d'usage en important TensorFlow Lite et le fichier *model.h*. Ainsi, je ne vais pas développer cette partie.

E - Résultats

Afin d'obtenir une détection de la neige en temps réel, nous avons utilisé le capteur de température et d'humidité de la carte Arduino. Il était nécessaire de convertir les valeurs en int8 et de les ramener à l'échelle appropriée. Lors de nos tests en temps réel à Bordeaux, nous n'avons pas pu vérifier si la détection de la neige fonctionnait. Ainsi, nous avons codé en dur des valeurs de température et d'humidité correspondant à des conditions où il neige habituellement.

Conclusion

À travers ces différents cas d'usage, nous avons pu observer que la méthodologie d'implémentation d'un réseau de neurones sur un microcontrôleur suit un processus commun, depuis l'entraînement du modèle jusqu'à son déploiement en environnement embarqué. Cependant, la principale différence réside dans la collecte et le traitement des données. En effet, la préparation du jeu de données est une étape essentielle en apprentissage automatique, influençant directement la qualité et la précision du modèle final. Selon le cas d'usage, le prétraitement des données peut être plus ou moins complexe, nécessitant parfois des ajustements spécifiques pour garantir des performances optimales sur des systèmes embarqués à ressources limitées.

Ce projet nous a permis d’approfondir notre compréhension des défis liés à l’IA embarquée, notamment en termes d’optimisation des modèles et d’adaptation aux contraintes matérielles des microcontrôleurs. L'intégration de TensorFlow Lite Micro ouvre la voie à de nombreuses applications autonomes et économes en énergie, renforçant ainsi l’intérêt croissant pour l’IA embarquée dans divers domaines.