Université de Bordeaux

Bordeaux INP

Enseirb-Matmeca

Année
3A

Option
Systèmes embarqués

Projet avancé : Développement de modèle IA optimisé avec TensorFlow Lite pour systèmes embarqués, implémenté sur Raspberry Pi

Encadrant : Patrice KADIONIK

by

MOHAMED EL HASSANI
&
NOUR LOUATI
&
ZAKARIA SOUHAIL
&
ALI JAWAD

Année universitaire 2024-2025

TABLE DES MATIÈRES

1 INTRODUCTION

L'intelligence artificielle (IA) embarquée est un domaine en plein essor, visant à intégrer des modèles de Machine Learning (ML) sur des appareils à ressources limitées tels que les microcontrôleurs ou les micro-ordinateurs, notamment la Raspberry Pi 4. Cette technologie permet d'exécuter des tâches complexes de manière autonome tout en respectant des contraintes strictes de performance et d'efficacité énergétique.

Dans ce projet, nous avons exploité les capacités de **TensorFlow Lite**, une version optimisée de TensorFlow conçue spécifiquement pour les environnements embarqués. TensorFlow Lite permet de transformer des modèles de machine learning en versions légères et performantes, parfaitement adaptées à des dispositifs comme la Raspberry Pi 4.

Deux cas d'utilisation principaux seront implémentés sur la Raspberry Pi 4 :

— **Détection et classification d'objets génériques** : une application exploitant le modèle YOLO (You Only Look Once) pour analyser des flux vidéo en temps réel, capturés par une caméra connectée.

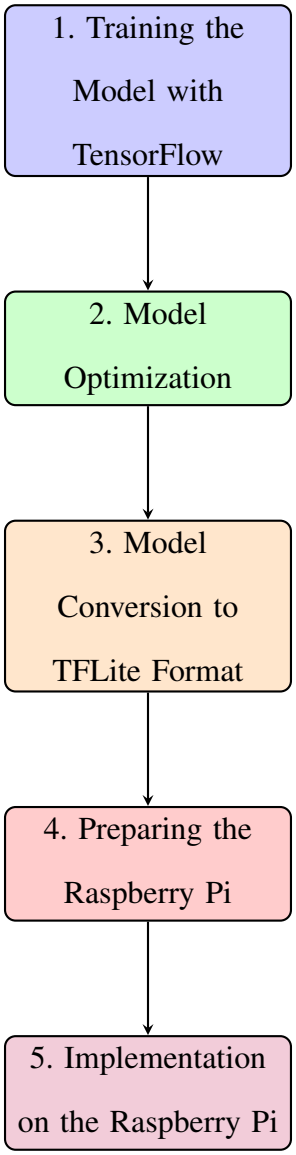
— **Reconnaissance de lettres de la langue des signes** : un système capable de détecter et de classifier des lettres à partir d'images, en utilisant un modèle optimisé pour l'embarqué.

Pour chaque cas d'utilisation, le déploiement suit un ensemble d'étapes structurées, comprenant :

1. **Entraînement du modèle avec TensorFlow** : Création d'un modèle adapté à la tâche spécifique.
2. **Optimisation du modèle** : Réduction de la taille et amélioration de l'efficacité via des techniques comme la quantification et le pruning.
3. **Conversion au format TFLite** : Transformation du modèle en une version légère adaptée aux environnements embarqués.

4. **Préparation de la Raspberry Pi** : Création d'un environnement virtuel Python et installation des bibliothèques nécessaires.
5. **Implémentation et tests sur la Raspberry Pi** : Développement d'algorithmes spécifiques pour tester le modèle sur des images et des flux vidéo.

Ces étapes garantissent une transition fluide du développement à l'implémentation, tout en mettant en lumière les défis et les opportunités des solutions d'IA embarquées.



2 Use Case : Détection d'objets

2.1 Introduction

2.1.1 Objectif du projet

L'objectif de cette partie du projet est d'implémenter une version du modèle de détection d'objets **YOLOv8** sur la **Raspberry Pi**, en suivant les mêmes formalismes que la partie précédente. Ce modèle sera ensuite testé en temps réel à l'aide de la webcam intégrée à la Raspberry Pi.

2.1.2 Présentation du matériel

Le matériel utilisé sera identique à celui de la partie précédente.

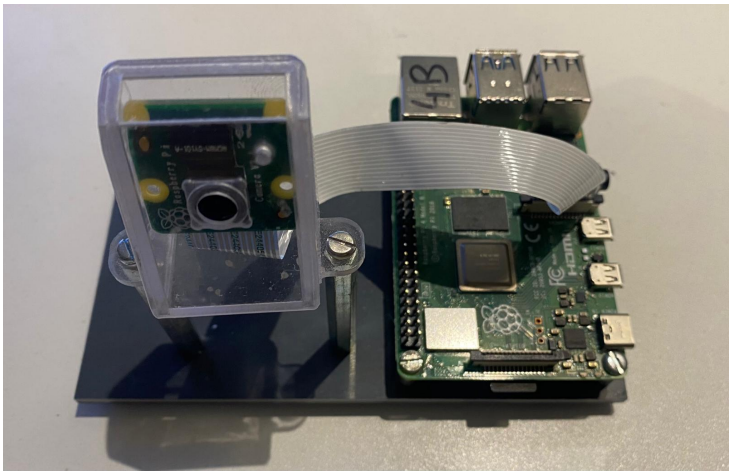


FIGURE 2.1 – Carte Raspberry Pi 4 avec caméra.

2.1.3 Présentation de l'environnement de développement logiciel

L'environnement de développement logiciel utilisé repose principalement sur **Anaconda** et **Jupyter Notebook**. Bien que cela ne soit pas strictement nécessaire (par exemple, **Spyder** aurait été plus simple à exécuter), nous avons initialement supposé que cet environnement serait indispensable. Finalement, cette approche nous a permis d'acquérir une nouvelle expertise sans pour autant compromettre notre efficacité.

Pour ce modèle, **nous n'appliquerons pas de pruning**, pour des raisons qui seront détaillées plus loin.

2.2 Préparation du modèle

Dans un premier temps, le travail est effectué sur un **PC** équivalent aux machines de l'école. Cette approche permet de vérifier l'ensemble du processus avant l'implémentation sur la Raspberry Pi, afin d'éviter les longs temps d'exécution et les difficultés de débogage inhérentes aux systèmes embarqués.

Les bibliothèques nécessaires sont installées sur le PC, notamment **Ultralytics, TensorFlow Lite, et Numpy**. Ensuite, un script (disponible en Annexe 5) est utilisé pour exporter le modèle au format `tflite`.

Explication du code :

— Importation des bibliothèques nécessaires.

— Chargement du modèle.

— Export du modèle au format `tflite`.

Bien que cela puisse sembler simple, cette étape peut rapidement devenir chronophage. En effet, tout au long du projet, l'installation des bibliothèques a posé problème, avec des conflits de versions et de dépendances omniprésents. Malheureusement, **il n'existe pas de recette universelle** pour résoudre ces incompatibilités.

À ce stade, notre modèle est disponible au format `tflite`. Il ne reste plus qu'à l'exploiter, d'abord sur PC, puis sur la Raspberry Pi.

2.3 Test du modèle sur PC

Le code utilisé pour tester le modèle est disponible en annexe 6 (les parties dédiées au débogage ont été supprimées).

Nous allons maintenant l'expliquer étape par étape :

- Importation des bibliothèques nécessaires.
- Définition des chemins du modèle et de l'image de test.
- Création de la liste `ClassNames` associant les `classID` aux classes correspondantes.
- Chargement du modèle et récupération des détails des entrées et sorties.
- Prétraitement de l'image : mise aux normes de YOLOv8 (par exemple, division par 255 pour normaliser les valeurs RGB).
- Inférence sur l'image.
- Récupération des données de sortie.
- Décodage de l'output : YOLOv8 génère 8400 détections par image. Chaque détection est représentée par un vecteur de 84 éléments :
  - Les deux premiers éléments indiquent les coordonnées du centre du rectangle à dessiner.
  - Les deux suivants correspondent à la hauteur et la largeur de la détection.
  - Les 80 restants représentent les scores de confiance associés aux classes détectées.

  Par exemple, si une personne est détectée, la confiance associée à la classe "person" (premier élément de `ClassNames`) sera élevée, tandis que les autres resteront faibles.
- Conversion des coordonnées du centre de détection en un format compatible avec **OpenCV** (utilisé pour afficher et dessiner les résultats sur l'image).
- Suppression des redondances avec la fonction `NMSBoxes`, qui filtre les multiples détections d'un même objet.

— Dessin des rectangles sur l'image avec **OpenCV**.

— Affichage du résultat final.

FIGURE 2.2 – Résultat de l'inférence sur PC.

2.4 Déploiement et Utilisation sur Raspberry Pi

Le déploiement sur la Raspberry Pi repose sur une approche similaire à celle utilisée sur PC, avec quelques ajustements :

— Capture d'images en temps réel à la place de l'utilisation d'images stockées.

— Affichage d'un flux vidéo en direct pour la détection d'objets.

— Ajustement des seuils de confiance et de suppression des redondances (`NMS`) pour améliorer la précision des détections (éviter que des objets proches soient fusionnés en une seule détection).

— Utilisation de **TensorFlow Lite Runtime** au lieu de **TensorFlow Lite**, car il est optimisé pour les systèmes embarqués comme la Raspberry Pi.

Cependant, nous avons rapidement atteint les limites matérielles de la Raspberry Pi. En effet, la vidéo en direct présente un **framerate de seulement 0.25 FPS**, ce qui est inacceptable pour un affichage en temps réel.

Analyse et solution : Une analyse des temps d'exécution a révélé que le problème ne provient pas de l'inférence du modèle, mais plutôt du décodage de la sortie. Ce processus est particulièrement coûteux en calcul et, malheureusement, **nous ne pouvons pas l'optimiser davantage**.

Nous avons initialement envisagé d'appliquer du **pruning** pour réduire la taille du modèle, mais cette approche n'aurait eu aucun impact sur la vitesse de traitement tout en dégradant potentiellement la qualité des prédictions.

Finalement, pour obtenir un résultat fonctionnel, nous avons remplacé le flux vidéo par un traitement image par image :

— Le programme capture une photo.

— Il applique le modèle YOLOv8 pour détecter les objets.

— Il affiche l'image annotée avec les détections.

Cette solution a été testée et validée en conditions réelles lors de la soutenance. Le modèle a correctement identifié plusieurs objets sur l'image, dont :

— Trois personnes.

— Deux tables et deux chaises.

— Un PC portable.

Toutes les détections étaient correctes. La photo utilisée est disponible sur la Raspberry Pi sous le chemin : `/Téléchargements/in.jpg`.

3 Use Case : Reconnaissance de la Langue des Signes avec TensorFlow Lite

3.1 Introduction

3.1.1 Objectif du projet

Ce projet vise à créer un modèle capable de reconnaître les lettres de l'alphabet en langue des signes à partir d'images, en utilisant TensorFlow. Le modèle est ensuite converti en format TensorFlow Lite pour un déploiement sur des systèmes embarqués, tels que la Raspberry Pi.

3.1.2 Présentation du matériel

Dans le cadre de ce projet, nous avons utilisé une carte Raspberry Pi 4, un mini-ordinateur puissant et polyvalent. Cette carte est équipée d'un processeur ARM Cortex-A72 quad-core, offrant une bonne capacité de calcul pour les tâches de traitement en temps réel. Sa petite taille et sa faible consommation énergétique en font un choix idéal pour des projets embarqués.

Nous avons également utilisé une caméra Pi v2.1, dotée d'un capteur de 8 mégapixels. Cette caméra est capable de capturer des images de haute qualité en résolution 1080p, ce qui la rend parfaitement adaptée aux applications de vision par ordinateur. Connectée à la carte Raspberry Pi via l'interface CSI, elle permet de réaliser des captures d'images et de vidéos pour les tâches de traitement.

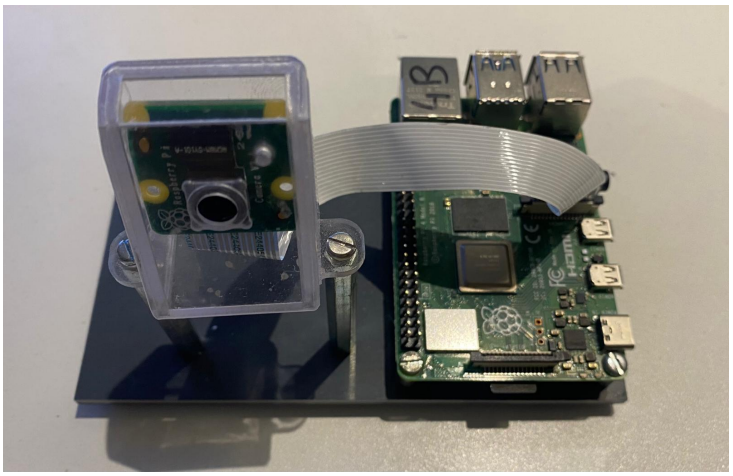


FIGURE 3.1 – Carte Raspberry pi 4 avec camera.

3.1.3 Présentation de l'environnement de développement logiciel

L'environnement de développement logiciel utilisé pour ce projet s'est principalement appuyé sur Google Colab et Spyder, deux outils performants et complémentaires. Le modèle de détection des signes a été développé et entraîné dans Google Colab, offrant un accès à des ressources matérielles puissantes, telles que les GPU, permettant un entraînement rapide et efficace. Parallèlement, Spyder a été utilisé pour le développement local et les tests approfondis du modèle, offrant une interface flexible et adaptée au développement en Python. L'optimisation du modèle, incluant des techniques telles que le pruning et la quantification, a également été réalisée dans ces deux environnements. Ce processus a conduit à la génération d'un fichier optimisé au format .tflite, spécialement conçu pour une exécution efficace lors de l'inférence sur la carte Raspberry Pi 4. Cette approche garantit des performances élevées tout en minimisant la consommation des ressources de l'appareil embarqué.

3.2 Préparation des Données

3.2.1 Chargement et Visualisation des Données

Les données utilisées sont issues de la base *Sign Language MNIST*, qui contient des images de lettres de l'alphabet (A-Z sans J ni Z). Les étapes principales incluent :

3.2.1.1 Chargement des fichiers CSV via Google Drive

Les fichiers CSV contenant les données d'entraînement et de test sont stockés dans Google Drive et chargés dans l'environnement Google Colab. Chaque fichier contient :

— Une colonne `label` représentant les classes des lettres (valeurs de 0 à 23).

— 784 colonnes correspondant aux pixels (28×28) des images.

Voici un aperçu des premières lignes des fichiers chargés (Figure 3.8).

label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9
3	107	118	127	134	139	143	146	150	153
6	155	157	156	156	156	157	156	158	158
2	187	188	188	187	187	186	187	188	187
2	211	211	212	212	211	210	211	210	210
13	164	167	170	172	176	179	180	184	185
16	161	168	172	173	178	184	189	193	196
8	134	134	135	135	136	137	137	138	138
22	114	42	74	99	104	109	117	127	142
3	169	174	176	180	183	185	187	188	190
3	189	189	189	190	190	191	190	190	190
18	133	135	141	146	150	155	158	159	163
10	0	25	38	40	41	46	50	56	69

FIGURE 3.2 – Aperçu des premières lignes des fichiers CSV des données d'entraînement et de test.

3.2.1.2 Séparation des labels et des données d'image

Les labels, correspondant à la colonne `label`, sont extraits séparément des données d'image. Cela permet de structurer les entrées et sorties du modèle.

3.2.1.3 Normalisation des pixels

Nous avons normalisé les valeurs des pixels des images afin qu'elles soient comprises entre 0 et 1. Cette étape a été réalisée en divisant chaque valeur par 255. La normalisation est une étape importante qui améliore la stabilité de l'entraînement et accélère la convergence du modèle.

3.2.1.4 Reshape des images

Nous avons reformé les images, initialement représentées sous forme de vecteurs unidimensionnels (784 pixels), en matrices de taille $28 \times 28 \times 1$. Ce format est adapté pour les réseaux de neurones convolutifs (CNN), car il conserve la structure spatiale des pixels.

Après cette transformation, les dimensions des ensembles d'entraînement et de test sont les suivantes :

— **Données d'entraînement :** $(27\,455, 28, 28, 1)$

— **Données de test :** $(7\,172, 28, 28, 1)$

3.2.1.5 Conversion des labels en encodage *one-hot*

Nous avons également converti les labels, initialement représentés sous forme d'entiers, en vecteurs *one-hot*. Par exemple, un label `3` est transformé en $[0, 0, 0, 1, 0, \ldots, 0]$, où la taille du vecteur correspond au nombre total de classes (24). Cette conversion est nécessaire pour que la sortie du modèle soit comparable aux labels cibles pendant l'entraînement.

3.3 Entraînement du Modèle

Dans cette section, nous décrivons l'architecture de notre modèle, le processus d'entraînement, et les résultats obtenus après optimisation.

3.3.1 Architecture du Modèle CNN

Pour ce projet, nous avons conçu un réseau de neurones convolutif (CNN) adapté à la reconnaissance des lettres de l'alphabet en langue des signes. Dans notre modèle CNN, le choix du nombre de couches et de filtres repose sur un compromis entre expressivité et efficacité computationnelle, guidé à la fois par des recherches sur des architectures existantes et par nos expériences pratiques acquises lors de stages d'été sur les réseaux de neurones convolutifs.

Le modèle est composé des couches suivantes :

— Deux couches convolutionnelles avec 32 et 64 filtres respectivement, suivies d'une fonction d'activation *ReLU* [1].

1. *ReLU* (*Rectified Linear Unit*) est une fonction d'activation définie par $f(x) = \max(0, x)$, qui met à zéro les valeurs négatives et conserve les positives. Elle est simple, rapide à calculer et favorise une convergence efficace en évitant les problèmes de saturation des fonctions .

— Deux couches de *MaxPooling* pour réduire la dimension des images tout en conservant les caractéristiques essentielles.

— Une couche de *Flattening* pour transformer les sorties convolutives en un vecteur unidimensionnel.

— Une couche dense intermédiaire avec 128 neurones et activation *ReLU*.

— Une couche de sortie avec 24 neurones et activation *Softmax*, correspondant aux 24 classes (lettres de l'alphabet sans J ni Z).

Nous avons choisi deux couches convolutives au lieu de trois pour éviter un coût computationnel excessif et limiter le risque de surapprentissage, en particulier pour des images de petite taille comme celles de MNIST. De plus, chaque couche de convolution est suivie d'un max-pooling `(2×2)` afin de réduire la dimensionnalité et d'éliminer les informations redondantes tout en conservant les caractéristiques les plus importantes. Cette approche a été validée par nos expériences en stage, où nous avons observé que l'ajout de couches de pooling améliore à la fois les performances et l'efficacité du modèle. Enfin, une couche `Flatten` transforme les matrices de caractéristiques en vecteurs, qui sont ensuite traités par une couche `Dense(128, activation='relu')`, jouant le rôle de classificateur.

La couche finale `Dense(24, activation='softmax')` est adaptée à notre tâche de classification multi-classes avec 24 catégories. Cette architecture, bien qu'optimisée pour des images de petite taille, assure un bon équilibre entre capacité d'apprentissage, généralisation et efficacité computationnelle, tout en restant compatible avec les contraintes matérielles d'un Raspberry Pi. L'architecture du modèle est résumée dans le Tableau 3.1.

Type de Couche	**Paramètres**	**Description**
Convolution 2D	32 filtres, 3×3	Activation *ReLU*
MaxPooling 2D	2×2	Réduction de la taille
Convolution 2D	64 filtres, 3×3	Activation *ReLU*
MaxPooling 2D	2×2	Réduction de la taille
Flatten	-	Transformation en vecteur
Dense	128 neurones	Activation *ReLU*
Dense	24 neurones	Activation *Softmax*

TABLE 3.1 – Architecture du modèle CNN.

3.3.2 Compilation et Entraînement

Nous avons compilé le modèle en utilisant l'optimiseur *Adam* et la fonction de perte *categorical crossentropy*, adaptée aux problèmes de classification multi-classes. L'entraînement a été effectué sur 10 époques avec une taille de lot (*batch size*) de 32, en utilisant l'ensemble d'entraînement. L'ensemble de test a été utilisé pour valider les performances à chaque époque. Les performances obtenues après 10 époques sont :

— **Précision sur les données d'entraînement :** 100%.

— **Précision sur les données de test :** 79.34%.

3.3.3 Évaluation du Modèle

Après l'entraînement, nous avons évalué le modèle sur l'ensemble de test pour obtenir les performances finales. Le tableau 3.2 résume les résultats de l'évaluation.

Métrique	Valeur
Précision	79.34%
Perte	0.74

TABLE 3.2 – Performance du modèle sur l'ensemble de test.

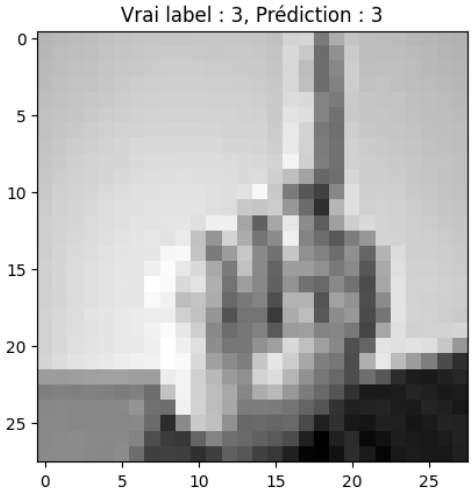
3.3.4 Visualisation des Prédictions

Pour mieux comprendre les performances du modèle, nous avons visualisé des exemples d'images issues de l'ensemble de test, accompagnées de leurs labels réels et des prédictions générées par le modèle. Ces exemples permettent d'évaluer la capacité du modèle à reconnaître les lettres de la langue des signes.

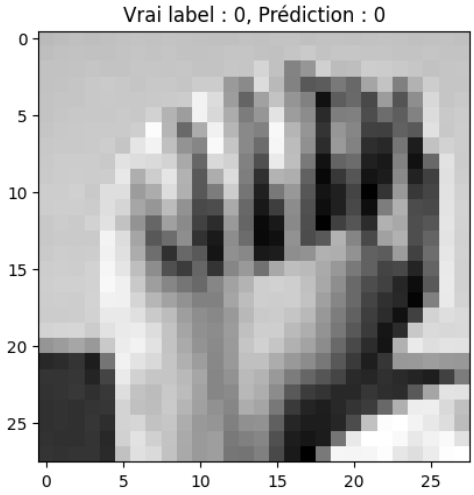
Les Figures 3.3a et 3.3b illustrent deux prédictions correctes effectuées par le modèle :

— **Figure 3.3a** : L'image représente la lettre associée au label 0. Le modèle a prédit correctement 0.

— **Figure 3.3b** : L'image représente la lettre associée au label 3. Le modèle a prédit correctement 3.



(a) Vrai label 3**, prédiction correcte** 3**.**



(b) Vrai label 0**, prédiction correcte** 0**.**

FIGURE 3.3 – Exemples d'images de l'ensemble de test avec leurs prédictions.

Ces visualisations démontrent que le modèle est capable d'identifier correctement certaines lettres, mais des exemples supplémentaires seraient nécessaires pour analyser ses erreurs et comprendre ses limitations.

3.4 Conversion en TensorFlow Lite

Afin de déployer le modèle sur des systèmes embarqués comme la Raspberry Pi, nous avons converti le modèle entraîné en format *TensorFlow Lite* (*TFLite*). Ce format est optimisé pour des environnements à ressources limitées, permettant une exécution rapide et efficace. La conversion du modèle a été effectuée à l'aide de l'API *TFLiteConverter* fournie par TensorFlow.

3.4.1 optimisation du modele

L'optimisation du modèle a constitué une étape essentielle dans ce projet, visant à adapter le modèle pour une exécution fluide sur des dispositifs embarqués tels que la Raspberry Pi 4. Deux techniques clés ont été utilisées : le pruning et la quantification. Le pruning a permis de réduire la complexité du modèle en supprimant les poids les moins significatifs tout en maintenant sa précision globale. Quant à la quantification, elle a transformé les poids du modèle en une représentation plus compacte, notamment en passant à des formats comme le float16 ou l'int8,

réduisant ainsi considérablement la taille du modèle. Ces optimisations ont eu pour objectif de minimiser les besoins en ressources computationnelles et énergétiques lors de l'inférence, tout en veillant à préserver des performances acceptables en termes de précision. Cette approche permet ainsi de garantir une exécution rapide et efficace, adaptée aux contraintes des environnements embarqués.

3.4.1.1 Prunning

Pour optimiser le modèle, une technique de pruning a été mise en œuvre afin de réduire sa taille et d'accélérer l'inférence sur une plateforme embarquée comme la Raspberry Pi 4. Le pruning, ou élagage, consiste à supprimer progressivement les poids les moins significatifs du réseau neuronal, en maintenant ceux qui contribuent le plus à la précision du modèle. Cela permet de rendre le modèle plus léger sans compromettre notablement ses performances.

Une stratégie de pruning par magnitude faible a été adoptée, où les poids les plus proches de zéro ont été éliminés. La sparsité a été introduite progressivement à l'aide d'une planification polynomiale (polynomial decay schedule). Cette méthode commence avec une sparsité initiale de 0%, pour éviter une perturbation soudaine au début de l'entraînement, et augmente graduellement jusqu'à une sparsité finale de 50%. Cette approche progressive permet au modèle de s'adapter au changement structurel tout en minimisant l'impact sur la précision.

Après l'entraînement du modèle pruné, une étape de stripping a été réalisée pour retirer les métadonnées liées au pruning, générant ainsi une version compacte et prête pour l'inférence. Cette réduction structurée des paramètres s'est traduite par une diminution significative de la taille du modèle tout en conservant des performances presque identiques à celles du modèle original.

3.4.1.2 Quantification

Une quantification en FLOAT16 a été réalisée pour réduire davantage la taille du modèle tout en maintenant des performances acceptables. La quantification consiste à réduire la précision des paramètres du modèle, initialement en 32 bits (FLOAT32), à 16 bits (FLOAT16). Cela diminue la consommation de mémoire et améliore la vitesse d'inférence, particulièrement utile pour des environnements embarqués à ressources limitées. Une tentative initiale de quantification en INT8 avait été réalisée, mais elle a significativement dégradé les performances du modèle, rendant cette approche moins adaptée. En revanche, la quantification FLOAT16 a permis d'obtenir un bon compromis entre la réduction de la taille du modèle et le maintien de la précision, avec une dégradation minimale des performances. Cette optimisation est particulièrement avantageuse sur des architectures ARM modernes comme celles utilisées dans la Raspberry Pi 4, qui prennent en charge les calculs en FLOAT16.

3.4.2 Test du Modèle TFlite sur des Images Réelles

Après avoir converti le modèle en format *TensorFlow Lite*, nous avons décidé de le tester sur des images capturées par nous-mêmes. Ces tests permettent d'évaluer les performances du modèle dans des situations réelles et de vérifier sa capacité à généraliser au-delà de la base de données *Sign Language MNIST*.

3.4.2.1 Capture et Prétraitement des Images

Les images des lettres en langue des signes ont été capturées à l'aide d'une caméra. Ces images ont ensuite été adaptées pour correspondre au format attendu par le modèle *TFLite*. Les étapes de prétraitement sont les suivantes :

— Chargement de l'image en niveaux de gris.

— Redimensionnement de l'image à une taille 28×28.

— Normalisation des pixels pour que leurs valeurs soient comprises entre 0 et 1.

— Ajout d'une dimension pour correspondre au format `(1, 28, 28, 1)` attendu par le modèle.

3.4.2.2 Prédiction avec le Modèle TFLite

Une fois les images adaptées, elles ont été passées dans le modèle *TensorFlow Lite*. Les étapes sont les suivantes :

— Chargement du modèle TFLite sauvegardé (`.tflite`).

— Passage de l'image prétraitée dans le modèle.

— Extraction de la classe prédite (lettre correspondante).

3.4.2.3 Résultats des Tests

Nous avons testé le modèle sur plusieurs images capturées par nous-mêmes. Par exemple, l'image de la lettre **C** (Figure 3.4) a été correctement prédite comme étant la lettre **C**. Ces résultats montrent que le modèle peut bien généraliser sur des données réelles après un prétraitement adapté.

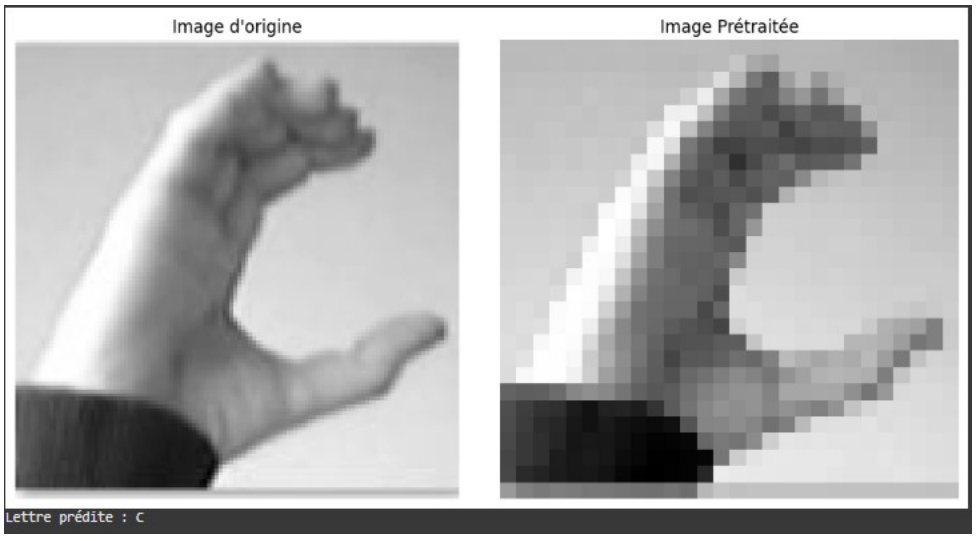


FIGURE 3.4 – Image d'origine capturée pour la lettre C (à gauche) et son image prétraitée (à droite).

La Table 3.3 résume les résultats obtenus pour plusieurs lettres.

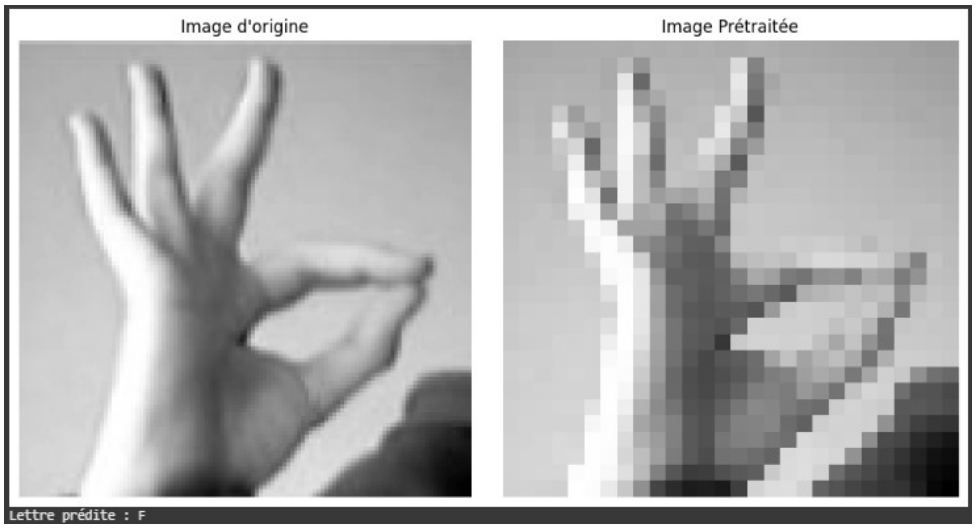


FIGURE 3.5 – Image d'origine capturée pour la lettre F (à gauche) et son image prétraitée (à droite).

Lettre Capturée	Lettre Prédite	Statut
I	I	Correct
U	U	Correct
G	G	Correct
F	F	Correct
C	C	Correct

TABLE 3.3 – Résultats des prédictions du modèle sur des images capturées.

3.5 Déploiement et Utilisation sur Raspberry Pi

3.5.1 Installation et Configuration

Pour exécuter le fichier .tflite généré lors de la phase de développement, un environnement virtuel Python a été configuré sur la Raspberry Pi 4. Cet environnement inclut des bibliothèques essentielles telles que tensorflow-aarch64 pour la compatibilité avec l'architecture ARM, et tflite-runtime pour l'inférence avec TensorFlow Lite. Les bibliothèques numpy et pandas sont utilisées pour le traitement des données, tandis que matplotlib permet de visualiser les résultats et d'analyser les performances. Pour gérer la caméra de la Raspberry Pi, la bibliothèque picamera a été installée. De plus, opencv-python est présent pour le traitement des images. Cette configuration assure une exécution fluide du modèle optimisé sur la Raspberry Pi en exploitant les capacités de l'architecture ARM64.

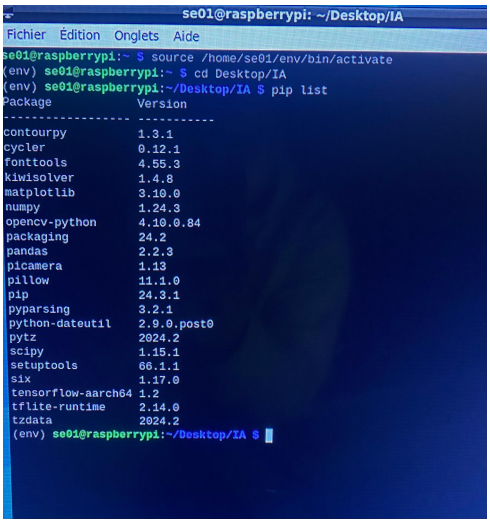


FIGURE 3.6 – Bibliothèques installées.

3.5.2 Évaluation du modèle optimisé TFlite sur la carte Raspberry Pi

3.5.2.1 Base de Données de Test MNIST

Afin d'évaluer la performance du modèle optimisé, un premier test a été réalisé à l'aide de la base de données de test Sign Language MNIST. Cette base contient des images représentant des lettres en langue des signes, préalablement utilisées pour l'entraînement et la validation. Le modèle TFLite a été chargé dans un environnement d'inférence spécifique, où chaque image a été prétraitée pour correspondre au format d'entrée attendu : les images en niveaux de gris ont été normalisées, redimensionnées à une taille de 28x28 et converties en tenseurs appropriés. Ensuite, cinq exemples aléatoires ont été sélectionnés et soumis au modèle pour générer des prédictions. Les résultats incluent les lettres réelles (celles étiquetées dans la base de données) et les lettres prédites par le modèle. Ce test permet de visualiser les performances initiales du modèle sur des données non vues tout en identifiant des cas de réussite ou d'erreur, comme illustré dans l'image des prédictions.

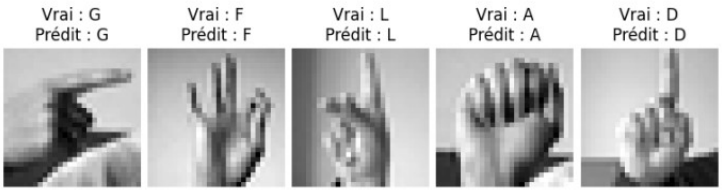


FIGURE 3.7 – Résultats des prédictions du modèle sur des exemples de la base de données de test.

3.5.2.2 Images Externes Personnalisées

Dans ce test, nous avons effectuer des prédictions de signes à partir d'images externes fournies. Chaque image a subi un prétraitement spécifique afin d'assurer sa compatibilité avec le modèle optimisé au format .tflite. Ce prétraitement comprend la conversion de l'image en niveaux de gris, son redimensionnement à une taille standard de 28x28 pixels, ainsi que la normalisation de ses pixels pour s'assurer que leurs valeurs sont comprises entre 0 et 1. Une fois ces étapes effectuées, le modèle applique le même algorithme que celui utilisé précédemment pour analyser les images et prédire le signe correspondant. Cette approche garantit la cohérence entre les tests réalisés sur la carte et les résultats attendus.

Nous avons réalisé ce test sur plusieurs images, en prenant comme exemple l'image représentant le signe de la lettre "I". Après avoir appliqué le prétraitement mentionné précédemment, le modèle a correctement prédit le signe correspondant. Le résultat de cette prédiction a ensuite été affiché directement sur l'image d'origine.

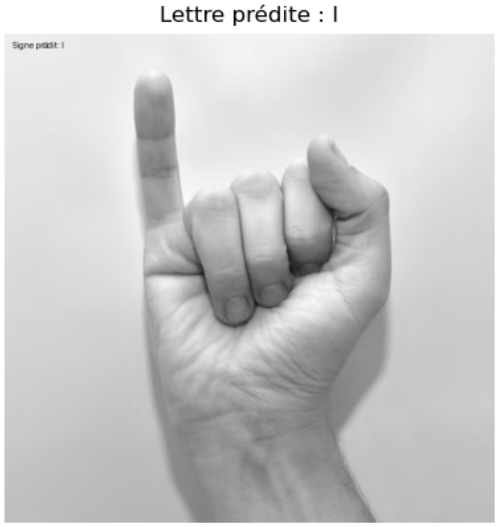


FIGURE 3.8 – Résultat de prédiction.

3.5.3 Prédictions en Temps Réel avec une Caméra

Le modèle est utilisé pour prédire les lettres capturées par une caméra connectée à la Raspberry Pi. Voici le script utilisé :

```
import tflite_runtime.interpreter as tflite
import cv2
import numpy as np

# Charger le mod le
interpreter = tflite.Interpreter(model_path="sign_language_model.
tflite")
interpreter.allocate_tensors()

# Obtenir les d tails des tenseurs
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# Initialiser la cam ra
cap = cv2.VideoCapture(0)

while True:
    ret, frame = cap.read()
    if not ret:
        break

    # Pr traitement
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    resized = cv2.resize(gray, (28, 28)).
    reshape(1, 28, 28, 1).astype('float32') / 255.0

    # Pr diction
```

```
    interpreter.set_tensor(input_details[0]['index'], resized)
    interpreter.invoke()
    output_data = interpreter.get_tensor(output_details[0]['index'])
    predicted_label = np.argmax(output_data)

    # Affichage
    cv2.putText(frame, f"Pr diction : {predicted_label}", (10, 30)
    , cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0), 2)
    cv2.imshow("Temps R el", frame)

    # Quitter avec 'q'
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

Lors de la phase d'implémentation de l'algorithme de détection des signes en temps réel sur la Raspberry Pi, nous avons rencontré un problème majeur lié aux dépendances de la bibliothèque OpenCV.Ce problème a empêché l'exécution de l'algorithme en temps réel, mettant en évidence les défis liés au déploiement de bibliothèques lourdes sur des systèmes embarqués aux ressources limitées.

3.6 Conclusion

Ce projet démontre la faisabilité de la reconnaissance des lettres de la langue des signes en utilisant un modèle de deep learning déployé sur une Raspberry Pi. Les résultats obtenus sont encourageants et pourraient être étendus pour inclure des mots ou phrases complets.

ANNEXES

A.1 ANNEXE 1

A.1.1 Code utilisé pour la préparation des Données

```
import pandas as pd
import numpy as np

# Chargement des donn es
train_data = pd.read_csv("sign_mnist_train.csv")
test_data = pd.read_csv("sign_mnist_test.csv")

# S paration des labels et des images
train_labels = train_data['label'].values
train_images = train_data.drop('label', axis=1).values
test_labels = test_data['label'].values
test_images = test_data.drop('label', axis=1).values

# Normalisation
train_images = train_images.astype('float32') / 255.0
test_images = test_images.astype('float32') / 255.0

# Reshape
train_images = train_images.reshape(-1, 28, 28, 1)
test_images = test_images.reshape(-1, 28, 28, 1)
```

A.2 ANNEXE 2

A.2.1 Code utilisé pour l'entraînement

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
Dense

# Construction du modele
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(24, activation='softmax')  # 24 classes
])

# Compilation
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Entrainement
model.fit(train_images, train_labels, epochs=10,
validation_data=(test_images, test_labels))
```

.

A.3 ANNEXE 3

A.3.1 Code utilisé pour effectuer ces étapes et afficher les images avant et après prétraitement

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def preprocess_and_display_image(image_path, target_size=(28, 28)):
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    plt.subplot(1, 2, 1)
    plt.imshow(image, cmap='gray')
    plt.title("Image d'origine")
    plt.axis('off')

    resized_image = cv2.resize(image, target_size)
    normalized_image = resized_image.astype('float32') / 255.0
    input_image = normalized_image.reshape(1,
    target_size[0], target_size[1], 1)

    plt.subplot(1, 2, 2)
    plt.imshow(resized_image, cmap='gray')
    plt.title("Image Pr trait e")
    plt.axis('off')

    plt.tight_layout()
    plt.show()

    return input_image
```

A.4 ANNEXE 4 : Pruning et quantification

```
pruned_model = tf.keras.models.clone_model(
    model,
    clone_function=lambda layer: tfmot.sparsity.keras.prune_low_magnitude(
        layer,
        pruning_schedule=tfmot.sparsity.keras.PolynomialDecay(
            initial_sparsity=0.0,  # Aucune sparsit  initiale
            final_sparsity=0.5,  # R duire la sparsit  finale

               50%
            begin_step=np.ceil(len(X_train) / 32).astype(np.int32)

            * 5,  # Retarder le d but
            end_step=np.ceil(len(X_train) / 32).astype(np.int32)

            * 20  #  tendre  la dur e
        )
    )
)

# Entra ner le mod le avec pruning
pruning_callbacks = [tfmot.sparsity.keras.UpdatePruningStep()]
pruned_model.compile(optimizer='adam', loss='categorical_crossentropy',

metrics=['accuracy'])
pruned_model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=20,
    batch_size=32,
```

```
    callbacks=pruning_callbacks
)

# Recompiler le mod le stripped avant l'  valuation
stripped_model = tfmot.sparsity.keras.strip_pruning(pruned_model)
stripped_model.compile(optimizer='adam', loss='categorical_crossentropy',

metrics=['accuracy'])
stripped_model.fit(X_train, y_train, validation_data=(X_val, y_val)

, epochs=10, batch_size=32)

# 4. Quantification FLOAT16
converter = tf.lite.TFLiteConverter.from_keras_model(stripped_model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]

# Appliquer les optimisations standards
converter.target_spec.supported_types = [tf.float16]

# Sp cifier la quantification FLOAT16
tflite_model = converter.convert()

# Sauvegarder le mod le TFLite
tflite_model_file = pathlib.Path("tflite_models/sign_language_

pruned_quantized_float16.tflite")
with open(tflite_model_file, "wb") as f:
    f.write(tflite_model)
print(f"TFLite Model saved to: {tflite_model_file}")
```

A.5 ANNEXE 5 : Code utilisé pour tester le modèle sur la carte

A.5.1 Test sur des images de la base de données MNIST

```
import numpy as np
import tflite_runtime.interpreter as tflite
import pandas as pd
import matplotlib.pyplot as plt

# Chemins des fichiers
MODEL_PATH = "/home/ubuntu/Documents/projet-

final/tflite_models/sign_language_model.tflite"
TEST_DATA_PATH = "/home/ubuntu/Documents/projet-final/
sign_mnist_test.csv"

# Charger le mod le TFLite
interpreter = tflite.Interpreter(model_path=MODEL_PATH)
interpreter.allocate_tensors()

# Obtenir les d tails des tenseurs
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# Liste des lettres d tect es
class_names = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "K",
"L", "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y"]

# Charger le fichier CSV
```

```
test_df = pd.read_csv(TEST_DATA_PATH)

# Pr traitement des donn es
def preprocess_data(df):
    """Pr traiter les donn es pour les tester avec le mod le."""
    labels = df['label'].values
    images = df.drop('label', axis=1).values
    images = images.reshape(-1, 28, 28, 1).astype('float32') / 255.0

    # Normalisation entre 0 et 1
    return images, labels

X_test, y_test = preprocess_data(test_df)

# Fonction pour pr dire une lettre
def predict_letter(image):
    """Pr dire la lettre dans une image fournie."""
    input_image = np.expand_dims(image, axis=0)  # Ajouter une

    dimension pour le batch

    # V rifier le type d'entr e attendu par le mod le
    input_type = input_details[0]['dtype']
    if input_type == np.float32:
        input_image = input_image.astype(np.float32)

    # Passer l'image dans le mod le
    interpreter.set_tensor(input_details[0]['index'], input_image)
    interpreter.invoke()
    output_data = interpreter.get_tensor(output_details[0]['index'])

```

```
    predicted_label = np.argmax(output_data)
    return class_names[predicted_label]

# Afficher 5 exemples d'images test avec pr dictions
def display_predictions(images, labels, predictions, class_names):
    """Afficher des exemples d'images avec leurs pr dictions."""
    plt.figure(figsize=(15, 10))
    for i in range(5):
        plt.subplot(1, 5, i + 1)
        plt.imshow(images[i].reshape(28, 28), cmap='gray')
        plt.title(f"Vrai : {class_names[labels[i]]}\nPr dit :

        {predictions[i]}")
        plt.axis('off')
    plt.tight_layout()
    plt.show()

# Effectuer les pr dictions sur 5 exemples
predictions = [predict_letter(X_test[i]) for i in range(5)]

# Afficher les images avec les pr dictions
display_predictions(X_test[:5], y_test[:5], predictions, class_names)
```

A.5.2 Test sur une image externe

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Jan 20 11:08:38 2025

@author: ubuntu
```

```
"""

import numpy as np
import tflite_runtime.interpreter as tflite
from PIL import Image, ImageDraw, ImageFont
import matplotlib.pyplot as plt

# Fonction pour pr traiter une image
def preprocess_image(image_path, target_size=(28, 28)):
    try:
        # Charger l'image en niveaux de gris
        image = Image.open(image_path).convert('L')
    except Exception as e:
        raise ValueError(f"Impossible de charger l'image a

        l'emplacement {image_path}. Erreur : {e}")

    # Redimensionner l'image
    resized_image = image.resize(target_size)

    # Normaliser
    normalized_image = np.array(resized_image).astype('float32') / 255.0

    # Ajouter une dimension pour le batch et le canal
    input_image = normalized_image.reshape(1, target_size[0],

    target_size[1], 1)

    return input_image, image

# Fonction pour pr dire une lettre avec le mod le TFLite
```

```
38 def predict_letter(image, model_path, class_names):
39     # Charger le mod le TFLite
40     interpreter = tflite.Interpreter(model_path=model_path)
41     interpreter.allocate_tensors()
42 
43     # Obtenir les d tails des tenseurs
44     input_details = interpreter.get_input_details()
45     output_details = interpreter.get_output_details()
46 
47     # V rifier le type attendu par le mod le
48     if input_details[0]['dtype'] == np.float32:
49         image = image.astype(np.float32)
50 
51     # Passer l'image dans le mod le
52     interpreter.set_tensor(input_details[0]['index'], image)
53     interpreter.invoke()
54     output_data = interpreter.get_tensor(output_details[0]['index'])
55 
56     # Obtenir l'indice de la classe pr dite
57     predicted_index = np.argmax(output_data)
58     return class_names[predicted_index]
59 
60 # Configuration
61 MODEL_PATH = "/home/ubuntu/Documents/projet-final/tflite_models/
62 
63 sign_language_model.tflite"
64 IMAGE_PATH = "/home/ubuntu/Documents/projet-final/II.png"
65 CLASS_NAMES = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "K",
66 
67 "L", "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y"]
68 
```

```
# V rifier si le fichier existe avant de continuer
import os
if not os.path.exists(IMAGE_PATH):
    raise FileNotFoundError(f"L'image sp cifi e n'existe pas :

    {IMAGE_PATH}")

# Pr traiter l'image
input_image, display_image = preprocess_image(IMAGE_PATH)

# Faire une pr diction
predicted_letter = predict_letter(input_image, MODEL_PATH, CLASS_NAMES)

# Dessiner le texte sur l'image d'origine
draw = ImageDraw.Draw(display_image)
font = ImageFont.load_default()
text = f"Signe pr dit : {predicted_letter}"
draw.text((10, 10), text, fill="black", font=font)

# Enregistrer l'image avec le texte
output_image_path = "/home/ubuntu/Documents/projet-final/

output_image.png"
display_image.save(output_image_path, format="PNG")

# Afficher l'image avec le texte
plt.imshow(display_image, cmap='gray')
plt.axis('off')
plt.title(f"Lettre pr dite : {predicted_letter}")
plt.show()

```

```
# Afficher le r sultat dans le terminal
print(f"Lettre pr dite : {predicted_letter}")
print(f"L'image annot e a   t   enregistr e sous : {output_image_path}")
```

A.6 ANNEXE 6

A.6.1 Code utilisé pour exporter le YOLO

```
from ultralytics import YOLO
export_directory = "/Users/moham/Documents/tensorflow2"
model = YOLO("yolov8n.pt")
model.export
model.export(format="tflite")
```

A.7 ANNEXE 7

A.7.1 Code utilisé pour tester la détection d'objets sur PC

```
import tensorflow as tf
import numpy as np
import cv2

TFLITE_MODEL_PATH = "/Users/moham/Documents
/tensorflow2/yolov8n_saved_model/yolov8n_float32.tflite"
IMAGE_PATH = "/Users/moham/Documents/tensorflow2/input_image.jpg"
classNames =
["person", "bicycle", "car", "motorbike", "aeroplane",
"bus", "train", "truck", "boat",
"traffic light", "fire hydrant", "stop sign", "parking meter",
```

```
"bench", "bird", "cat",
"dog", "horse", "sheep", "cow", "elephant", "bear", "zebra",
"giraffe", "backpack", "umbrella",
"handbag", "tie", "suitcase", "frisbee", "skis", "snowboard",
"sports ball", "kite", "baseball bat",
"baseball glove", "skateboard", "surfboard", "tennis racket",
"bottle", "wine glass", "cup",
"fork", "knife", "spoon", "bowl", "banana", "apple", "sandwich",
"orange", "broccoli",
"carrot", "hot dog", "pizza", "donut", "cake", "chair", "sofa",
"pottedplant", "bed",
"diningtable", "toilet", "tvmonitor", "laptop", "mouse",
"remote", "keyboard", "cell phone",
"microwave", "oven", "toaster", "sink", "refrigerator",
"book", "clock", "vase", "scissors",
"teddy bear", "hair drier", "toothbrush"]

# On charge le mod le Tflite
interpreter = tf.lite.Interpreter(model_path=TFLITE_MODEL_PATH)

# Obtenir les d tails d'entr e et de sortie
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
interpreter.allocate_tensors()
"""
print()
print("Input details:")
print(input_details)
print()
print("Output details:")
print(output_details)
```

```
print()
"""
# Preprocess l'image
input_size = input_details[0]['shape'][1:3]  # Get input
size from model
image = cv2.imread(IMAGE_PATH)
image_resized = cv2.resize(image, (input_size[1], input_size[0]))
input_data = np.expand_dims(image_resized, axis=0).
astype(np.float32) / 255.0
# D clarer input tensor
interpreter.set_tensor(input_details[0]['index'], input_data)
# Inference
interpreter.invoke()

# R cuperation de output data
output_data = interpreter.get_tensor(output_details[0]['index'])
image_width=600
image_height=400
A = output_data.transpose() #Pour notre confort

# Detection apr s le process
boxes = []
confidences = []
for detection in A:
    for i in range(4,84,1):
        confidence = detection[i][0]  # Confiance
        if confidence > 0.5:  # CMinimum de confiance
            class_id = i-4  # Class ID, le -4 puisqu'on a
            commenc  notre boucle de 4
            #Calcul des coordonn es des rectangles    mettre,
            puisque le mod le
```

```
74                #donne des coordonn es
75                de centre
76                x1 = int(detection[0][0]*image_width-
77                detection[2][0]*image_width/2)
78                y1 = int(detection[1][0]*image_height-
79                detection[3][0]*image_height/2)
80                x2 = int(detection[0][0]*image_width+
81                detection[2][0]*image_width/2)
82                y2 = int(detection[1][0]*image_height+
83                detection[3][0]*image_height/2)
84                boxes.append([x1, y1, x2, y2])
85                confidences.append(confidence)
86
87 boxes = np.array(boxes)
88 confidences = np.array(confidences)
89 indices = cv2.dnn.NMSBoxes(boxes.tolist(), confidences.tolist(),
90 score_threshold=0.5, nms_threshold=0.4)
91 final_boxes = []
92 #Positionner les rectangles sur l'image
93 for sekizo in indices: #Nomenclatures  tranges , mais dur de
94 nous blamer, puisque
95     #il y a Vanilla, Raspberry,
96     et des recettes sur Yocto :( ...
97     cv2.rectangle(image, (boxes[sekizo][0], boxes[sekizo][1]),
98         (boxes[sekizo][2], boxes[sekizo][3]), (0, 255, 0), 2)
99     label = f"Class: {classNames[class_id]}, Conf:
100    {confidences[sekizo]:.2f}"
101    cv2.putText(image, label, (x1, y1 - 10), cv2
102    .FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)
103
104 # Affichage du r sultat
```

```
105 cv2.imshow("YOLOv8 Detection", image)
106 cv2.waitKey(0)
107 cv2.destroyAllWindows()
```