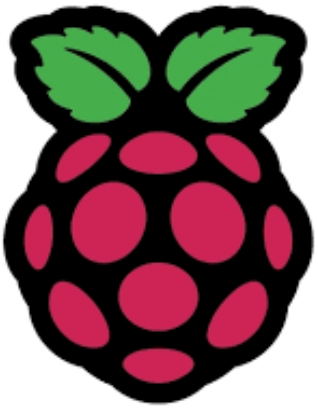
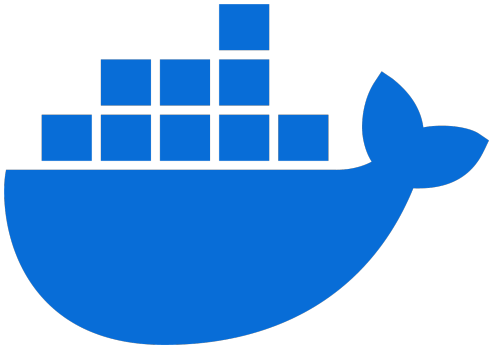
Rapport

IA embarquée





Auteurs :
M. El Mehdi Marzoug
M. Romain Losilla

Encadrant :
M. Frédéric Chatrie

Introduction

Les cartes embarquées sous Linux sont couramment utilisées dans les systèmes d'identification et de détection légers. Elles offrent une manœuvrabilité et une facilité d'utilisation qu'on ne retrouve pas avec les systèmes classiques. Leur faible consommation énergétique et leur coût réduit en font des solutions idéales pour des applications mobiles ou des systèmes embarqués. Cependant, elles exigent une implémentation extrêmement optimisée du système cible, ce qui peut rendre le développement plus complexe. Notre projet consiste à classifier des chiffres manuscrits en utilisant une carte Raspberry Pi. Nous commençons par entraîner notre modèle sur un conteneur Docker, ce qui permet d'assurer un environnement contrôlé pour l'entraînement. Ensuite, nous migrons le modèle pré-entraîné vers un code en langage C, afin d'optimiser les performances sur la carte Raspberry Pi. Cela a impliqué le développement des différentes couches du modèle en C ainsi que la mise en place d'un mécanisme permettant de lire les poids du modèle, garantissant une exécution rapide et efficace sur l'architecture embarquée.

Dans le cadre de notre projet, nous avons exploré différentes architectures de deep learning, telles que les réseaux de neurones convolutifs (CNN), ainsi que différents types de données d'entraînement afin d'obtenir le meilleur modèle possible. En ajustant les hyperparamètres et en analysant les performances de chaque modèle, nous avons cherché à maximiser la précision tout en minimisant l'utilisation des ressources de la carte Raspberry Pi. Cette approche nous a permis de trouver un équilibre entre la qualité du modèle et l'efficacité du système embarqué.

Données d'entraînement

Comme nous n'avions pas accès à la caméra du Raspberry Pi, nous avons été contraints de dessiner les chiffres de 0 à 9 sur Paint, en utilisant une résolution de 1000x1000 pixels (ensuite redimensionnées à 32x32 pixels) et un pinceau de largeur 28 px. Pour chaque chiffre, nous avons créé 10 variantes, ce qui nous a permis de couvrir différentes façons de dessiner chaque chiffre. Cette base de données a ensuite été rognée et délimitée à l'aide des fonctions fournies, afin d'améliorer la cohérence et la standardisation de la base. Ainsi, nous avons entraîné nos modèles sur une concaténation de ces différentes variantes, afin d'assurer une diversité suffisante au sein de la base de données et de garantir des performances robustes du modèle.

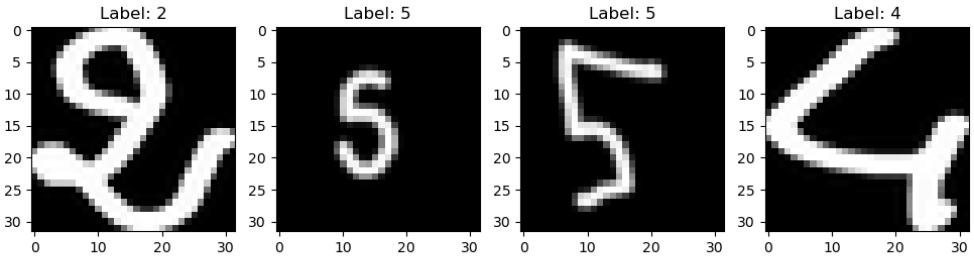


FIGURE 1 – Illustration de quelques échantillons de la base de données

Architectures

Notre approche pour définir les architectures était simple. Dans un premier temps, nous souhaitions explorer une architecture composée de calculs élémentaires afin d'établir une base solide de fonctions, ce qui nous permettrait ensuite d'implémenter l'inférence de réseaux plus complexes.

Ainsi, la première architecture sur laquelle nous avons réalisé l'apprentissage, suivie de l'étape d'inférence en C, est un MLP (Multi-Layer Perceptron). Son architecture est représentée dans la figure 2. Étant donné que les calculs effectués dans cette architecture sont relativement simples (une multiplication et une addition pour la couche Fully Connected, ainsi qu'une multiplication et une addition pour la couche BatchNorm1d), cela nous permet de nous concentrer sur la gestion de l'apprentissage et l'exportation des poids pour la phase d'inférence.

MLP

Cette architecture a été entraînée sur une base de données composée de 20 images de taille 32×32 en niveaux de gris (un seul canal) par label. La base de données a été divisée de manière à allouer 80 % des données à l'apprentissage et 20 % à la validation. À l'issue de l'entraînement, le modèle atteint un taux de précision de 70 % sur cette base.

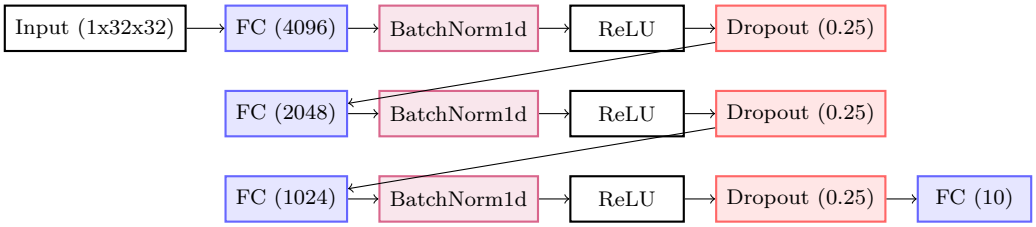


FIGURE 2 – Architecture du MLP utilisée

Le principal défaut de cette architecture est qu'elle est très gourmande en espace de stockage, avec plus de 165 millions de poids. De plus, ses performances sont inférieures à celles d'un CNN. Cependant, elle présente l'avantage de ne contenir que des opérations simples à implémenter (Multiplications matricielles).

CNN

La seconde architecture sur laquelle nous avons effectué l'apprentissage, ainsi que l'étape d'inférence implémentée en C par la suite, est un réseau de neurones convolutif (CNN). Cette architecture est largement répandue et permet d'obtenir de très bons résultats, même sur une base de données d'images de taille réduite. La structure détaillée de ce CNN est représentée dans la figure 3.

Nous avons obtenu une précision de 92.5% sur une base donnée composée des chiffres manuscrits rognés et non rognés (20 échantillons par chiffre).

Nous avons également testé notre modèle sur les bases des autres groupes, les résultats sont satisfaisants avec une précision d'inférence de 92 % sur la base de données du groupe AMOURA/LAHGAZI.

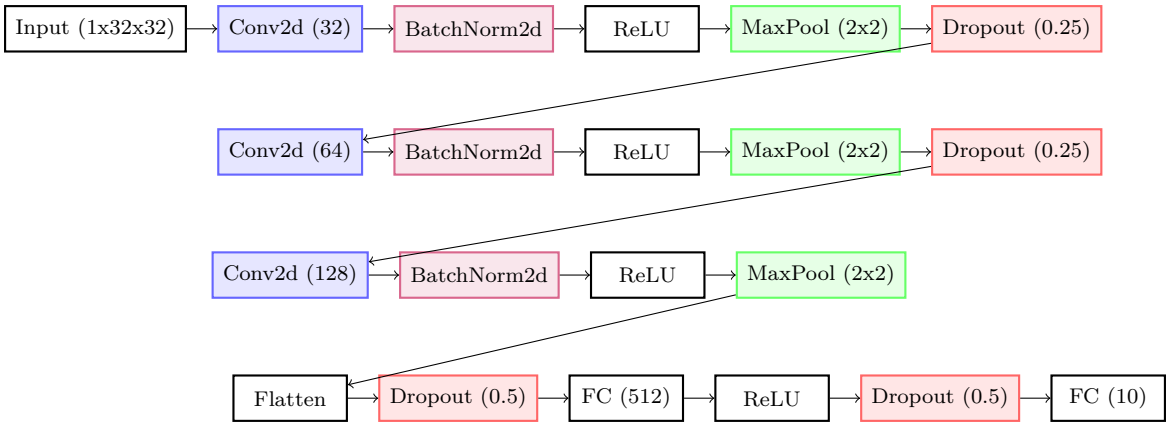


FIGURE 3 – Architecture du CNN utilisée

Implémentation sur Rasbperry

Maintenant que nous avons défini nos deux modèles et effectué les phases d'apprentissage, notre objectif est de porter la phase d'inférence de chaque modèle sur une carte Raspberry Pi et d'implémenter cette phase en C. Cette étape a pour but de répondre aux attentes actuelles de l'industrie, qui cherche à développer des IA sur des cartes embarquées, pour lesquelles le langage C est la meilleure solution.

L'implémentation de l'étape d'inférence en C se décompose en deux parties : l'enregistrement et la lecture des poids, ainsi que l'inférence des modèles.

Enregistrement et lecture des poids

Afin de pouvoir effectuer une étape d'inférence en C, il est nécessaire d'exporter les poids associés à chaque couche et de pouvoir les lire pour reconstruire chaque couche lors de l'inférence.

Pour ce faire, nous avons décidé d'enregistrer les poids de notre modèle dans un fichier texte, tout en suivant un formalisme précis. Ce formalisme est le suivant :

- **Nom de la couche**
- **Nombre total de paramètres de la couche**
- **Nombre de dimensions de la couche**
- **Taille de chaque dimension de la couche** (une dimension par ligne)
- **Poids de la couche** (un poids par ligne)
- **Saut d'une ligne**

Ainsi, en enregistrant nos poids selon ce formalisme, nous pouvons facilement lire le fichier texte et repérer chaque couche ainsi que les paramètres qui lui sont associés. Cette étape est effectué par le programme *aff_arci.py*.

Inférence des modèles

Maintenant que nous pouvons lire les poids de chaque couche, il faut savoir re-décrire l'architecture que ces poids définissent. Ainsi, l'implémentation de notre étape d'inférence se décompose en plusieurs étapes :

— **Chargement des poids** : Nous lisons dans le fichier texte les différentes informations que nous stockons dans un tableau 1D pour le nom de la couche, la taille des couches d'entrée et de sortie, et un tableau 2D pour les poids associés à chaque couche.
— **Chargement de l'image** : Nous lisons une image au format `.bmp` et la stockons dans un tableau 1D.
— **Enchaînement des couches** : Nous appelons une fonction qui enchaîne les différentes couches de notre architecture. Attention, dans nos modèles, après une couche de convolution, nous appliquons successivement une normalisation par lots (*batch normalization*), une activation ReLU et un sous-échantillonnage maximal (*max pooling*), dans cet ordre. Si nous appliquons une couche entièrement connectée (*Fully Connected*), alors nous ajoutons une activation ReLU après la normalisation par lots.

Dans la dernière étape, il faut être vigilant quant aux différents types d'arguments que chaque couche prennent. Ainsi, nous pouvons être amenés à transformer des tableaux 1D (les poids de chaque couche) en tableaux 2D pour les couches fully connected et en tableaux 4D pour les couches convolutionnelles. De plus, la gestion de la mémoire est très importante est cruciale pour bien enchaîner les différentes couche de nos modèles.Cette étape est effectué par le programme *weights_reader.c*.

Implémentation des couches en C

Dans cette section, nous détaillons l'implémentation des différentes couches du réseau de neurones en langage C. Chaque fonction a été optimisée pour l'exécution sur Raspberry Pi tout en maintenant la précision des calculs.

Convolution 2D (Conv2d)

```
void Conv2d(double* img_in, long in_channels, long out_channels,
            long width, long height,
            double**** weights, double* bias, long kernel_size,
            long padding, long stride, double* img_out);
```

La fonction `Conv2d` implémente une couche de convolution 2D avec support complet des fonctionnalités essentielles :
— Gestion du padding pour maintenir les dimensions spatiales
— Support des strides (pas de convolution)
— Traitement multi-canaux (entrée et sortie)

Max Pooling 2D (MaxPool2d)

```
void MaxPool2d(double* img_in, long in_channels, long width, long height,
               long pool, double* img_out);
```

La fonction `MaxPool2d` effectue l'opération de sous-échantillonnage maximal sur l'image d'entrée. Elle présente les caractéristiques suivantes :
— Support du traitement multi-canaux

— Calcul adaptatif des dimensions de sortie
— Gestion des bords de l'image
— Optimisation pour les tailles de pool standard (2x2)

Batch Normalization 2D (BatchNorm2d)

```
void BatchNorm2d(double* img_in, double* gamma, double* beta,
                 long in_channels, long width, long height,
                 double* img_out);
```

L'implémentation de la normalisation par lots comprend :
— Calcul des moyennes par canal
— Calcul des variances par canal
— Application des paramètres gamma et beta

Dropout

```
void Dropout(double* img_in, long in_channels, long width, long height,
             long p, double* img_out);
```

La fonction `Dropout` implémente la régularisation par abandon avec :
— Génération aléatoire optimisée avec `rand()`
— Mise à l'échelle automatique des sorties pendant l'inférence
— Support du traitement multi-canaux

Couche linéaire (FC)

```
void Linear(double* img_in, long size, long out_channels,
            double** weights, double* bias, double* img_out);
```

La fonction `Linear` implémente une couche entièrement connectée avec :
— Multiplication matricielle optimisée
— Addition des biais
— Vérification de validité des pointeurs d'entrée

Fonction d'activation ReLU

```
void Relu(double* img_in, long in_channels, long width, long height,
          double* img_out);
```

La fonction `Relu` implémente l'activation ReLU (Rectified Linear Unit) :
— Application élément par élément de la fonction max(0, x)
— Support du traitement multi-canaux

Optimisations générales

Toutes les fonctions développées partagent des caractéristiques communes d'optimisation :

— Utilisation de types `long` pour la gestion des indices
— Gestion efficace de la mémoire avec libération
— Vérifications de validité des pointeurs

Ces implémentations permettent d'exécuter efficacement l'inférence de nos modèles sur la carte Raspberry Pi, tout en maintenant un équilibre entre performances et utilisation des ressources.

Conclusion

Ce projet nous a permis d'explorer en profondeur l'implémentation d'algorithmes d'intelligence artificielle sur des systèmes embarqués, en particulier sur une carte Raspberry Pi. Nous avons suivi une approche progressive et méthodique, partant de l'entraînement des modèles jusqu'à leur déploiement sur la carte.

Les principaux résultats obtenus sont :

— Le développement réussi de deux architectures neuronales (MLP et CNN) avec des performances respectives de 70% et 92.5% sur notre jeu de données
— La validation de notre approche sur des données externes, avec notamment 92% de précision sur le jeu de données d'un autre groupe

Plusieurs défis techniques ont été surmontés durant ce projet :

— Trouver la meilleur architecture
— Le passage du framework PyTorch à une implémentation C pure

Perspectives d'amélioration

Plusieurs pistes d'amélioration peuvent être envisagées pour des développements futurs :

— L'implémentation d'autres types de couches pour supporter des architectures plus complexes
— L'ajout d'un système de quantification des poids pour réduire l'empreinte mémoire
— Le développement d'une interface utilisateur pour faciliter l'utilisation du système
— L'intégration de la capture d'image en temps réel via la caméra de la Raspberry Pi

Ce projet démontre la faisabilité du déploiement de modèles de deep learning sur des systèmes embarqués à ressources limitées, tout en maintenant des performances satisfaisantes. Il ouvre la voie à de nombreuses applications pratiques dans des domaines tels que la vision par ordinateur embarquée, les systèmes IoT intelligents.

Annexe

Vous trouverez notre base de données , l'image du docker [1](contenant les codes, les librairies et système d'exploitation) ainsi que les codes développés sur notre dépôt Git [2] .

1. `https://hub.docker.com/r/nartbeonline/ia_leg_marzoug_losilla`
2. `https://github.com/Nartbeonline/ia_leg.git`