TSI: Développement d'une IA légère sur un système embarqué
Rapport de projet

Maxime Hurtubise
& Hector Taler-Fraisse

Table des matières

1 Introduction et analyse du problème **4**

2 Architecture MLP et CNN **4**
2.1 Architecture MLP . . . 4
2.2 Architecture CNN . . . 4

3 Résultats expérimentaux (entrainement et tests des modèles) **5**
3.1 Gestion du jeu de données et split stratifié . . . 6
3.2 Protocole d'entraînement . . . 6
3.3 Analyse des performances . . . 6

4 Export et chargement des poids **7**
4.1 Export des paramètres (Python) . . . 7
4.2 Chargement en mémoire (C) . . . 7
4.3 Correspondance des dimensions . . . 7

5 Description de la chaine de prétraîtement (inférence) **7**
5.1 Caractéristiques du jeu de données MNIST . . . 8
5.2 Réduction de la zone d'intérêt . . . 8
5.3 Binarisation et détection de contours . . . 9
5.4 Extraction et préparation du chiffre . . . 9
5.5 Redimensionnement et alignement du barycentre . . . 9
5.6 Padding et format final . . . 9
5.7 Normalisation des valeurs . . . 9
5.8 Outil de validation visuelle . . . 9
5.9 Illustration du pipeline de prétraitement . . . 10

6 Structure de l'implémentation C **10**
6.1 Architecture modulaire . . . 10
6.2 Module neural_network . . . 11
6.3 Module process_frame . . . 11
6.4 Module contour_detection . . . 12
6.5 Module contour_selector . . . 12
6.6 Module digit_extractor . . . 12
6.7 Module main . . . 13
6.8 Flux de données . . . 13
6.9 Compilation et dépendances . . . 13
6.10 Modularité et extensibilité . . . 14

7 Comparaison du MLP et du CNN avec une étude statistique **14**
7.1 Précision de classification . . . 14
7.2 Temps d'inférence . . . 15
7.3 Matrice de confusion . . . 15
7.4 Analyse . . . 15
7.5 Interprétation des métriques . . . 15
7.6 Conclusion . . . 15

8 Guide de déploiement sur Raspberry Pi **16**
8.1 Prérequis . . . 16
8.2 Installation . . . 16
8.3 Démarrage de l'application . . . 16
8.4 Autres commandes utiles . . . 17

9 Conclusion **17**

Annexes **18**
A Spécifications et performances attendues pour l'application 18
B Chargement du modèle MLP 18
C Chargement du modèle CNN 18

1 Introduction et analyse du problème

Ce projet a pour but d'implémenter une application de détection et reconnaissance de chiffres manuscrits. L'application doit être légère pour pouvoir être déployée sur une cible embarquée. Dans notre cas, le déploiement se fera sur une Raspberry Pi 5 via une application complètement conteneurisée avec Docker. Les performances attendues sont renseignées en annexe A. Ces dernières doivent être atteintes sur un dataset de test personnalisé de chiffres manuscrits imprimés puis filmés par la caméra embarquée sur la Raspberry Pi 5.
Parmis les défis principaux de ce projet nous pouvons souligner :

- L'entrainement de modèles légers d'IA afin de les rendre performants non seulement sur MNIST mais aussi sur des données provenant d'un dataset personnalisé (ici notre dataset personalisé).
- L'implémentation de l'application entière via un code optimisé en C/C++, y compris les modèles de réseaux de neurones (CNN et MLP) afin d'atteindre les performances attendues.
- La gestion des différents environnements de développement et execution via Docker.

2 Architecture MLP et CNN

Pour mener à bien le projet, l'entraînement des modèles et leur inférence au niveau de l'application se feront dans deux environnements différents. Pour l'entrainement, nous utiliserons Pytorch avec Python dans un conteneur Docker contenant toutes les bibliothèques nécéssaires aux expérimentations. L'applicatif supportant l'inférence sera lui déployé via un autre conteneur et exécutera l'application en C/C++. Nous définissions deux architectures : un MLP et un CNN, les plus simples possibles pour obtenir une baseline et comparer les résultats obtenus avec ceux attendus dans le cahier des charges (Annexe A). Les modèles décrit par la suite sont définis avec Pytorch dans les fichiers suivants du livrable : `train_cnn.py` et `train_mlp.py`.

2.1 Architecture MLP

L'architecture du MLP est extrêmement minimaliste et n'est composée que d'une seule couche caché de 512 neurones. La couche cachée (Couche Dense 1) totalise $(784 \text{ entrées} \times 512 \text{ neurones}) + 512 \text{ biais} = 401\,920$ paramètres. La couche de sortie permet de revenir à 10 sorties chacune représentant la probabilité d'appartenance de l'image d'entrée à un chiffre.

Opération	**Type de couche**	**Dimensions de sortie**	**Paramètres**
Entrée	Image MNIST	$1 \times 28 \times 28$	0
Aplatissement	Flatten	784	0
Couche Dense 1	Linéaire + ReLU	512	401 920
Couche de Sortie	Linéaire	10	5 130
Total			**407 050**

Table 1: Détails de l'architecture du modèle MinimalMLP.

2.2 Architecture CNN

Le modèle CNN (Convolutional Neural Network) est conçu pour capturer les dépendances spatiales des chiffres tout en restant léger pour l'inférence. Les résultats de nos entraînements (détaillés dans la partie suivante) montrent que cette force lui permet de conserver une bonne précision pour des données qui s'éloignent du set d'apprentissage, contrairement au MLP. Notre architecture de CNN utilise deux couches de convolution avec un *stride* de 2, permettant de diviser par deux la résolution spatiale à chaque étape sans avoir recours à une couche de *Pooling* explicite.

Opération	Configuration	Dimension Sortie	Activation	Paramètres
Entrée	Image MNIST	$1 \times 28 \times 28$	-	0
Convolution 1	16 filtres 5×5, st. 2	$16 \times 14 \times 14$	ReLU	416
Convolution 2	32 filtres 5×5, st. 2	$32 \times 7 \times 7$	ReLU	12 832
Aplatissement	Flatten	1568	-	0
Couche de Sortie	Linéaire	10	Softmax	15 690
Total				**28 938**

Table 2: Détails de l'architecture du modèle CNN.

Le nombre de paramètres dans chaque couche convolutive est calculé ainsi :

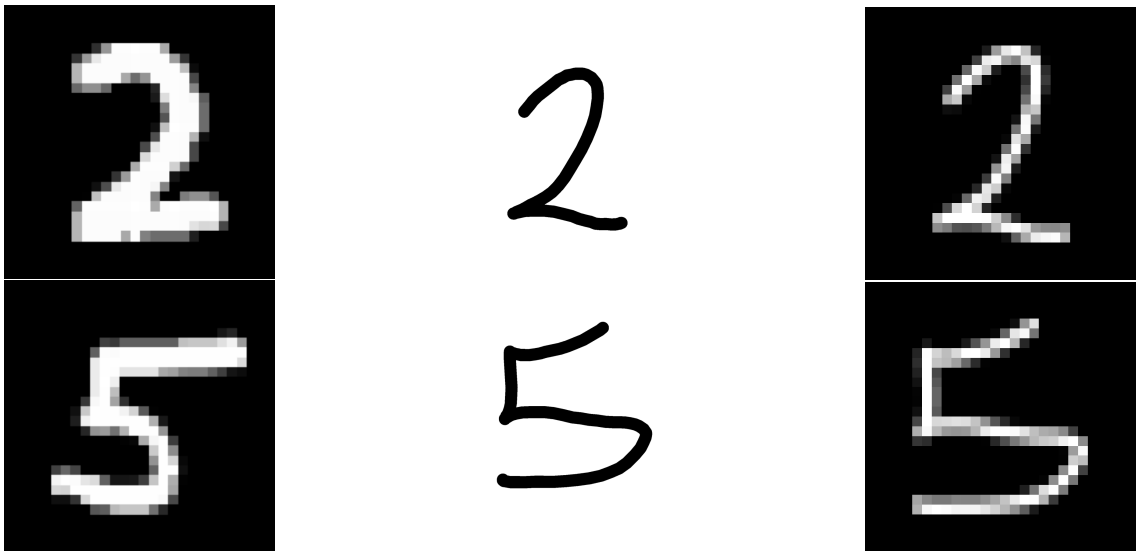
$$N_{\text{params}} = (k_d \times k_h \times k_w \times C_{\text{in}})\, C_{\text{out}} + C_{\text{out}} \tag{1}$$

Dans cette expression, k_d, k_h et k_w désignent respectivement la profondeur, la hauteur et la largeur du noyau de convolution 3D, C_{in} correspond au nombre de canaux en entrée, C_{out} au nombre de filtres de la couche, et C_{out} représente les biais associés à chaque filtre.

Analyse comparative : Bien que le CNN semble plus "complexe" conceptuellement, il est environ **14 fois plus léger** en nombre de paramètres que le MLP. Cette efficacité vient du partage de poids propre aux couches convolutives qui devrait théoriquement offrir une meilleure robustesse face aux variations de position du chiffre dans l'image caméra.

3 Résultats expérimentaux (entrainement et tests des modèles)

Cette section détaille le protocole expérimental mis en place pour l'entraînement des modèles, la gestion du jeu de données hybride, ainsi qu'une analyse des performances obtenues. Ci-dessous nous représentons des exemples de données originales de MNIST, de données de notre dataset personnel, et de ce dernier traité pour une conversion au format MNIST. Nous utiliserons exclusivement les données de format MNIST pour les entrainement et l'inférence des modèles qui sont dimmensionnés et entrainés spécifiquement pour ce format.



(a) Exemples MNIST (b) Données Personnelles (c) Sortie mises au format MNIST

Figure 1: Comparaison des données MNIST de référence, des données personnelles brutes et du résultat après la chaîne de prétraitement.

3.1 Gestion du jeu de données et split stratifié

Ensemble	Ratio	Effectif	Usage principal
Train	80 %	200	Mise à jour des poids (Backpropagation)
Validation	10 %	25	Suivi de la généralisation et tuning
Test	10 %	25	Évaluation finale (données "aveugles")
Total	100 %	**250**	

Table 3: Répartition stratifiée du jeu de données personnel.

Le split est de type **stratifié** : la proportion de chaque classe (chiffres de 0 à 9) est conservée dans chaque sous-ensemble (autant que possible comme les effectifs ne sont pas pairs pour la validation et le test). Les données sont organisées selon la structure `ImageFolder` de PyTorch, facilitant leur chargement.

3.2 Protocole d'entraînement

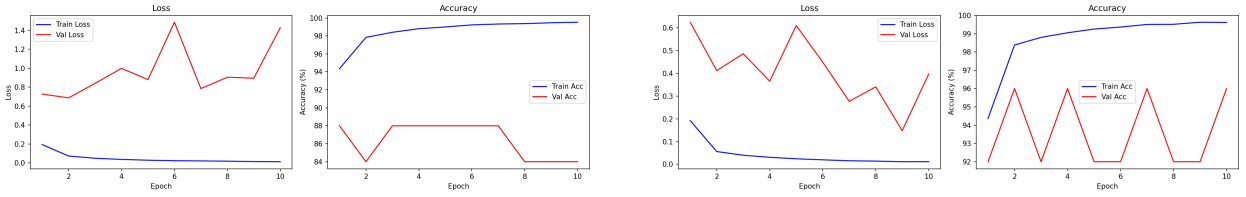
Le défi majeur réside dans le déséquilibre entre le dataset MNIST (60 000 images) et notre dataset personnel (quelques dizaines ou centaines d'images). Pour pallier à cela, nous avons adopté une stratégie de **sur-échantillonnage** : lors de la création du `train_dataset`, les données personnelles sont répétées 50 fois et concaténées à MNIST.

$$\texttt{Train\_Total} = \text{MNIST} \cup (50 \times \text{Perso\_Train})$$

Les deux modèles ont été entraînés avec l'optimiseur **Adam** (learning rate de 0.001) et la fonction de perte **CrossEntropyLoss**. Les statistiques de normalisation (μ et σ) ont été calculées sur l'ensemble global pour assurer une cohérence parfaite entre l'entraînement et l'inférence en C. Nous notons la moyenne et la variance du dataset `Train_total` utilisé pour l'entrainement du model afin d'effectuer la même normalisation au niveau de l'inférence en C. On obtient des statistiques légèrement différentes de celles du dataset MNIST seul : $Mean = 0.1305, Std = 0.3013$

3.3 Analyse des performances

Les figures 6a et 6b présentent l'évolution de la perte (*Loss*) et de la précision (*Accuracy*) sur 10 epochs.



(a) Courbes MLP

(b) Courbes CNN

Figure 2: Évolution de l'apprentissage sur les datasets hybrides.

Analyse du MLP : On observe une précision d'entraînement proche de 99 %, mais une précision de validation très instable oscillant entre 84 % et 88 %. La *Val Loss* a tendance à diverger, signe d'un **overfitting** marqué. Le MLP "mémorise" MNIST mais peine à généraliser les caractéristiques morphologiques de notre dataset personnel.

Analyse du CNN : Le CNN montre une meilleure robustesse. Bien que la précision de validation oscille également (due à la petite taille du set de validation), elle atteint des pics plus élevés (jusqu'à 96 %). La perte de validation reste globalement plus basse que celle du MLP, confirmant que l'extraction de caractéristiques spatiales par convolution est mieux adaptée à la variabilité de nos prises de vues réelles.

4 Export et chargement des poids

Le transfert des poids (*weights*) et des biais (*biases*) du format PyTorch vers l'application C repose sur une symétrie entre la classe Python et les structures de données en C.

4.1 Export des paramètres (Python)

PyTorch stocke les paramètres dans un dictionnaire ordonné (*OrderedDict*) suivant l'ordre de déclaration des couches. Le script d'exportation parcourt ce dictionnaire via `model.named_parameters()` et linéarise chaque tenseur en une liste continue de flottants avec la méthode `.flatten()` de NumPy.
L'ordre d'écriture dans le fichier `.txt` est séquentiel : *Poids de la couche n, suivis de ses biais ; Poids de la couche $n+1$, suivis de ses biais ; ...*

4.2 Chargement en mémoire (C)

Pour intégrer ces données, l'application utilise des structures (`MLPModel` et `CNNModel`) dont l'organisation mémoire correspond à l'architecture des classes Python, permettant de charger le fichier `.txt` par une lecture séquentielle simple. Le code C alloue dynamiquement des pointeurs dimensionnés selon les paramètres du modèle. La fonction de chargement remplit ensuite ces espaces dans l'ordre exact de l'exportation (Extrait des fonctions de chargement en Annexe B et C)

4.3 Correspondance des dimensions

La validité de l'inférence dépend de la cohérence entre les constantes du header C (`INPUT_SIZE`, `KERNEL_SIZE`, etc.) et l'architecture Python. Le tableau 4 détaille le mapping pour le modèle CNN.

Composant Python	Structure C	Taille (float)
`layer1.0.weight`	`model->conv1_w`	400
`layer1.0.bias`	`model->conv1_b`	16
`layer2.0.weight`	`model->conv2_w`	12 800
`layer2.0.bias`	`model->conv2_b`	32
`fc.weight`	`model->fc_w`	15 680
`fc.bias`	`model->fc_b`	10

Table 4: Mapping séquentiel entre les paramètres PyTorch et les buffers C.

5 Description de la chaine de prétraîtement (inférence)

La reconnaissance de chiffres manuscrits repose fortement sur la cohérence entre les données utilisées à l'entraînement et celles fournies au réseau de neurones en phase d'inférence. Dans ce projet, les modèles ont été entraînés sur une base de données personnelle, (mélange du jeu *MNIST* et de chiffres manuscrits). Les images de la BDD (Base De Données) personnelle ont été soumises à un prétraitement similaire à celui de *MNIST* afin de respecter ses standards (format, centrage et normalisation). Il est donc essentiel que les images issues de la caméra suivent rigoureusement ce même pipeline de prétraitement afin de garantir des performances de reconnaissance optimales.

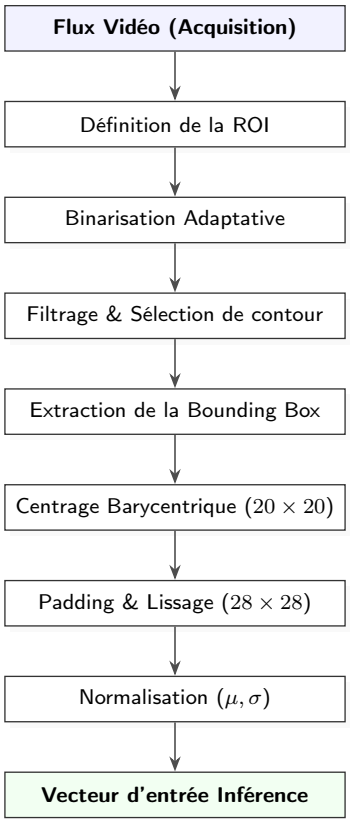


Figure 3: Pipeline technique de prétraitement (version compacte).

5.1 Caractéristiques du jeu de données MNIST

Le jeu de données MNIST est composé d'images de chiffres manuscrits (0–9) présentant les propriétés suivantes :

- Les chiffres sont représentés en blanc sur un fond noir.
- Chaque chiffre est contenu dans une image de taille 28×28 pixels.
- Le chiffre occupe une zone centrale équivalente à un carré de 20×20 pixels.
- Le barycentre du chiffre est aligné avec le centre du carré 20×20.
- Une bordure de 4 pixels est ajoutée autour de ce carré pour obtenir la taille finale 28×28.
- Les valeurs des pixels sont normalisées avant l'inférence.

Afin d'assurer une compatibilité maximale avec le modèle entraîné, ces contraintes doivent être rigoureusement respectées lors du traitement des images issues de la caméra.

5.2 Réduction de la zone d'intérêt

Dans un premier temps, une *Region of Interest* (ROI) est définie au centre de l'image d'entrée. Cette ROI correspond à environ 60 % de la hauteur et 30 % de la largeur de l'image.
Cette restriction spatiale permet :

- de réduire la charge computationnelle,
- d'éviter la détection de contours parasites provenant de l'arrière-plan,
- de se concentrer sur la zone où l'utilisateur est le plus susceptible de présenter un chiffre manuscrit.

5.3 Binarisation et détection de contours

L'image contenue dans la ROI est convertie en niveaux de gris, puis binarisée à l'aide d'un seuillage adaptatif. Ce choix permet de gérer des variations d'éclairage tout en mettant en évidence les structures du chiffre.
Une détection de contours est ensuite effectuée afin d'identifier les formes présentes. Chaque contour est évalué selon :

- son aire,
- son rapport largeur/hauteur,
- son taux de remplissage (rapport entre l'aire du contour et celle de sa boîte englobante).

Le contour présentant le score le plus élevé est sélectionné comme candidat représentant le chiffre manuscrit. Une *bounding box* est alors calculée autour de ce contour.

5.4 Extraction et préparation du chiffre

À partir de la *bounding box*, le masque binaire du chiffre est extrait. Une binarisation stricte est appliquée afin de garantir des valeurs de pixels exclusivement égales à 0 (noir) ou 255 (blanc), correspondant respectivement au chiffre et au fond.
L'image est ensuite inversée de manière à respecter le format MNIST : chiffre blanc sur fond noir.

5.5 Redimensionnement et alignement du barycentre

Le barycentre du chiffre est calculé à partir des pixels blancs du masque. Cette étape est essentielle afin de garantir un centrage correct du chiffre, indépendamment de sa position initiale dans la boîte englobante.
Le chiffre est ensuite redimensionné de façon proportionnelle pour tenir dans un carré de 20×20 pixels, tout en conservant son rapport d'aspect. Le barycentre du chiffre est aligné avec le centre de ce carré, situé en $(10, 10)$.

5.6 Padding et format final

Un *padding* de 4 pixels noirs est réalisé autour de l'image 20×20, ce qui permet d'obtenir une image finale de taille 28×28, conforme au format MNIST.
Un léger lissage gaussien est appliqué afin d'adoucir les transitions et de se rapprocher de l'apparence des chiffres manuscrits du jeu de données d'origine.

5.7 Normalisation des valeurs

Avant l'inférence, les valeurs de pixels sont converties en nombres flottants et normalisées. Les valeurs sont d'abord ramenées dans l'intervalle $[0, 1]$, puis normalisées à l'aide de la moyenne et de l'écart-type de notre BDD (Base De Données) personnelle :

$$\mu = 0.1307 \qquad \sigma = 0.3081$$

Cette étape garantit une distribution des entrées cohérente avec celle utilisée lors de l'entraînement du réseau.

5.8 Outil de validation visuelle

Afin de valider le bon fonctionnement du prétraitement, l'image 28×28 finale est affichée en *overlay* sur le flux vidéo, après agrandissement. Cet outil de débogage visuel permet de vérifier en temps réel :

- le centrage du chiffre,
- la qualité de la binarisation,
- la cohérence globale avec le format MNIST.

Cette visualisation a permis d'affiner les étapes de prétraitement et d'assurer une entrée optimale pour le réseau de neurones.

5.9 Illustration du pipeline de prétraitement

Afin d'illustrer concrètement les différentes étapes du pipeline de prétraitement décrit précédemment, la Figure 4 présente trois exemples représentatifs. Les visualisations présentées ont été obtenues en exécutant exactement le même code de prétraitement que celui de l'application, sur des images d'entrée fixes, simulant le comportement réel du système.

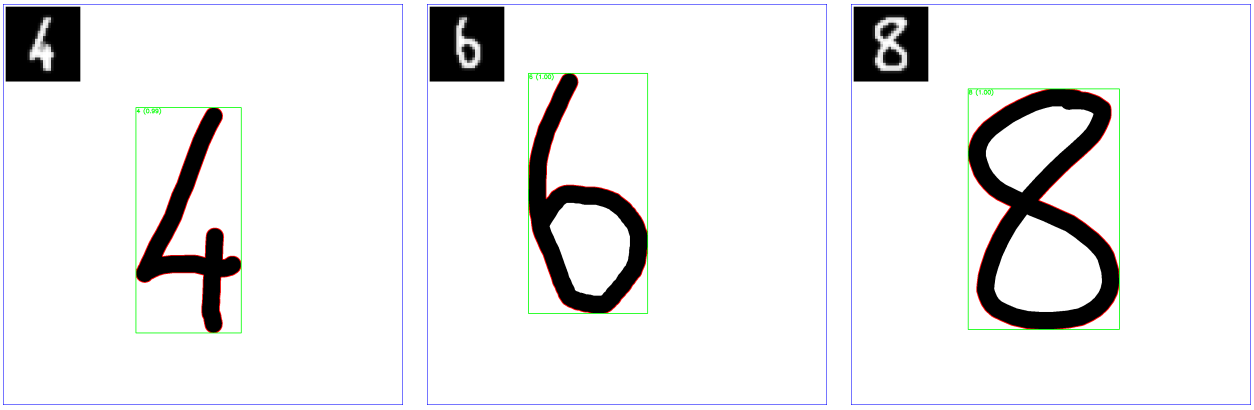


Figure 4: Illustration du pipeline de prétraitement sur trois exemples.

La zone d'intérêt (ROI) est matérialisée par un carré bleu. Les contours détectés sont affichés en rouge et conduisent à la définition de la *bounding box* entourant le chiffre manuscrit. La prédiction du réseau suite à la *forward_pass* y est indiquée dans un coin. En haut à gauche, l'image prétraitée (28×28) est affichée en *overlay*.

Ces exemples mettent en évidence l'ensemble des concepts introduits précédemment : la restriction spatiale via la ROI, la sélection du contour pertinent, l'extraction et le centrage du chiffre, ainsi que la génération de l'image finale normalisée destinée à l'inférence. L'affichage en *overlay* de l'image 28×28 permet de vérifier visuellement une forte similarité avec les images du jeu de données MNIST, tant en termes de centrage que de contraste et de structure, confirmant ainsi la cohérence du prétraitement avec les données d'entraînement.

6 Structure de l'implémentation C

L'application d'inférence en C/C++ a été conçue selon une architecture modulaire privilégiant la séparation des responsabilités. Cette approche facilite la maintenance, le débogage et l'extension future du système. L'implémentation repose sur un découpage en modules spécialisés, chacun encapsulant une étape spécifique du pipeline de traitement.

6.1 Architecture modulaire

Le code est organisé autour de six modules principaux, dont les interfaces sont définies par des fichiers d'en-tête (.h) et les implémentations dans des fichiers source (.c/.cpp). Cette séparation permet une compilation séparée et une intégration flexible des composants. Le tableau 5 présente une vue synthétique de cette organisation.

Module	Langage	Responsabilité
`neural_network`	C	Chargement des poids, structures de données des modèles (MLP/CNN), inférence (forward pass)
`process_frame`	C++	Orchestration du pipeline complet : acquisition, prétraitement, inférence et affichage
`contour_detection`	C++	Détection de tous les contours dans l'image binarisée via OpenCV
`contour_selector`	C++	Sélection du contour le plus pertinent selon des critères morphologiques
`digit_extractor`	C++	Extraction, centrage barycentrique, padding et normalisation au format MNIST
`main`	C++	Point d'entrée, gestion des flux vidéo (GStreamer), boucle principale

Table 5: Modules constitutifs de l'application C/C++.

6.2 Module neural_network

Le module `neural_network` constitue le cœur de l'inférence. Implémenté en C pur pour maximiser les performances et la portabilité, il expose des structures de données correspondant exactement aux architectures Python décrites précédemment.

Structures de données : Les structures `MLPModel` et `CNNModel` encapsulent l'ensemble des paramètres apprenables :

- `MLPModel` : 4 pointeurs vers les matrices de poids (`W1`, `W2`) et les vecteurs de biais (`b1`, `b2`).
- `CNNModel` : 6 pointeurs pour les deux couches convolutives (`conv1_w/b`, `conv2_w/b`) et la couche dense finale (`fc_w/b`).

Fonctions principales :

- `load_mlp_model` / `load_cnn_model` : Allocation dynamique de la mémoire et lecture séquentielle des poids depuis les fichiers `.txt`.
- `forward_pass_mlp` / `forward_pass_cnn` : Implémentation explicite de la propagation avant (produits matriciels, convolutions, activations ReLU).
- `get_prediction` : Détermination de la classe prédite (argmax sur les scores de sortie).
- `free_mlp_model` / `free_cnn_model` : Libération de la mémoire allouée.

L'implémentation des convolutions a été réalisée manuellement (sans bibliothèque externe de calcul matriciel) afin de maintenir un contrôle total sur les opérations et d'optimiser la localité des accès mémoire.

6.3 Module process_frame

Le module `process_frame` joue le rôle d'orchestrateur. Il coordonne l'ensemble des étapes de traitement pour chaque image acquise :

1. **Extraction de la ROI** : Définition d'une zone d'intérêt centrée (30 % × 60 % pour la caméra, image entière pour les tests).
2. **Prétraitement initial** : Conversion en niveaux de gris et binarisation adaptative.
3. **Détection et sélection** : Appel aux modules `contour_detection` et `contour_selector`.
4. **Extraction du chiffre** : Appel au module `digit_extractor` pour obtenir l'image 28×28 normalisée.

5. **Inférence** : Mesure du temps d'exécution et appel à `forward_pass_mlp` ou `forward_pass_cnn`.
6. **Post-traitement** : Calcul de la confiance via softmax, affichage des résultats (overlay, bounding box, texte).

Une structure `FrameProcessorState` maintient l'état entre deux trames (dernière prédiction, confiance, temps d'inférence) pour éviter l'affichage répété d'informations identiques.

6.4 Module contour_detection

Ce module encapsule les fonctionnalités OpenCV de détection de contours :

- `contour_det_find_all` : Applique `cv::findContours` sur l'image binarisée pour extraire tous les contours fermés.
- `contour_det_draw` : Superpose visuellement les contours détectés sur l'image (utile pour le débogage).

6.5 Module contour_selector

La fonction `contour_sel_find_best` implémente la logique de sélection du contour candidat. Elle applique une série de filtres morphologiques :

- **Aire minimale** : Rejet des contours trop petits (bruit) via `CONTOUR_SEL_MIN_AREA = 150.0`.
- **Rapport d'aspect** : Contrainte sur le ratio largeur/hauteur ($0.2 \leq r \leq 2.0$) pour éliminer les formes aberrantes.
- **Taux de remplissage** : Ratio entre l'aire du contour et celle de sa boîte englobante (privilégie les formes pleines).

Le contour ayant le score le plus élevé (produit de l'aire et du taux de remplissage) est retenu.

6.6 Module digit_extractor

Ce module implémente rigoureusement les spécifications MNIST décrites en section précédente. Il se décompose en deux fonctions principales :

digit_extr_extract : Transformation de la région d'intérêt binaire en image 28×28 :

1. Calcul du barycentre du chiffre via `digit_extr_compute_barycenter`.
2. Redimensionnement proportionnel pour tenir dans 20×20 pixels (préservation du rapport d'aspect).
3. Centrage du barycentre au point $(10, 10)$ via `digit_extr_resize_and_center`.
4. Ajout d'un padding de 4 pixels noirs pour obtenir 28×28.
5. Application d'un post-traitement morphologique (fermeture, lissage gaussien) via `digit_extr_apply_mnist_processing`.

digit_extr_to_nn_input : Conversion de l'image 28×28 en vecteur d'entrée normalisé :

1. Conversion des pixels en flottants dans l'intervalle $[0, 1]$.
2. Normalisation avec les statistiques du dataset d'entraînement ($\mu = 0.1307$, $\sigma = 0.3081$).
3. Linéarisation en un tableau de 784 flottants.

6.7 Module main

Le point d'entrée de l'application gère l'infrastructure système :

- **Paramétrage** : Lecture des arguments (hôte, ports, résolution).
- **Chargement des modèles** : Initialisation des structures `MLPModel` et `CNNModel`.
- **Pipelines GStreamer** : Configuration des flux vidéo entrant (H.264 depuis `rpicam-vid`) et sortant (réencodage H.264 pour diffusion TCP).
- **Boucle principale** : Lecture des trames, appel à `process_frame`, écriture dans le flux de sortie, mesure du débit (FPS).
- **Gestion des signaux** : Capture de `SIGINT`/`SIGTERM` pour un arrêt propre.

6.8 Flux de données

La figure 5 illustre les dépendances et le flux de données entre les modules lors du traitement d'une trame.

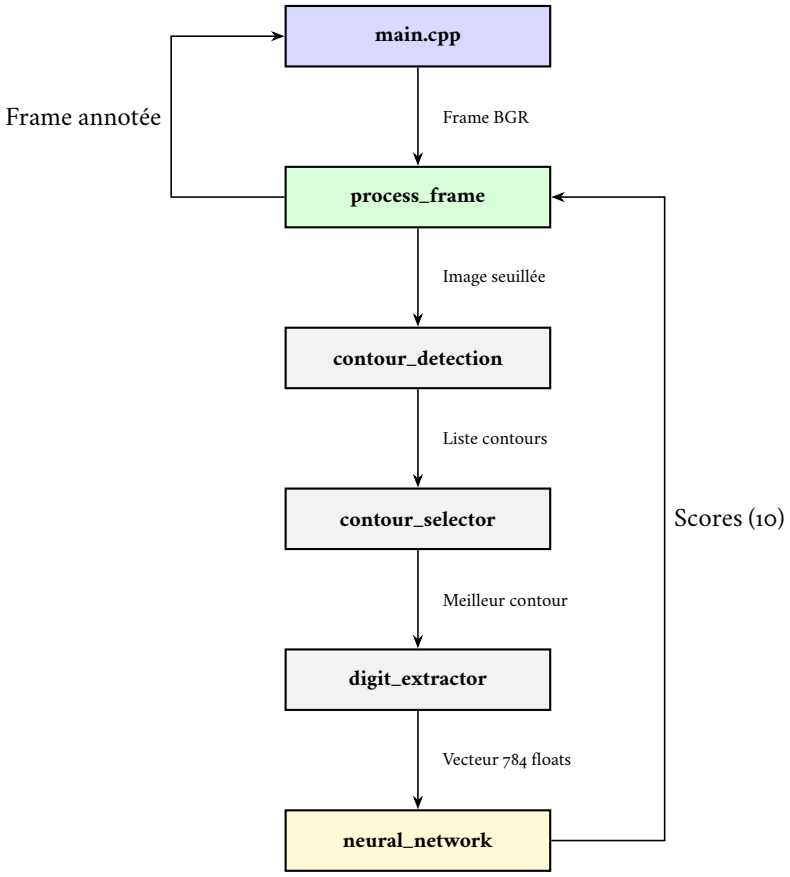


Figure 5: Flux de données entre les modules lors du traitement d'une trame.

6.9 Compilation et dépendances

Le `Makefile` organise la compilation en deux cibles :

- **app** : Application principale pour flux caméra (`main.cpp`).
- **app_images** : Variante pour tests sur images statiques (`main_images.cpp`).

Les dépendances externes se limitent à :

- **OpenCV 4** : Traitement d'images, gestion des flux GStreamer.
- **Bibliothèque mathématique standard** (`-lm`) : Fonctions `exp`, `sqrt` pour les activations.

Les flags de compilation privilégient les performances (`-O3`) tout en conservant les avertissements (`-Wall`). L'option `-std=c++17` est requise pour les fonctionnalités modernes de C++ utilisées dans les modules de traitement d'images.

6.10 Modularité et extensibilité

Cette architecture présente plusieurs avantages :

- **Testabilité** : Chaque module peut être testé indépendamment (exemple : `app_images` pour valider le prétraitement).
- **Réutilisabilité** : Le module `neural_network` est totalement découplé d'OpenCV et peut être intégré dans d'autres projets.
- **Maintenabilité** : Les responsabilités clairement délimitées facilitent les modifications (ex : ajout d'un nouveau type de modèle).
- **Optimisation ciblée** : Les parties critiques (convolutions, produits matriciels) peuvent être remplacées par des implémentations SIMD ou accélérées matériellement sans modifier l'interface.

7 Comparaison du MLP et du CNN avec une étude statistique

Afin d'évaluer les performances de nos deux architectures de réseaux de neurones, nous avons réalisé un *benchmark* sur un jeu de test composé de 25 images manuscrites prétraitées selon le standard MNIST. Les métriques calculées incluent la précision, les temps d'inférence moyens et les débits (FPS). Toutes ces mesures ont été obtenues sur un ordinateur équipé d'une puce M2 Pro.

Métrique	MLP	CNN
Précision (%)	84.00	92.00
Temps moyen (ms)	0,033	0,138
Temps médian (ms)	0,027	0,124
Écart-type temps (ms)	0,029	0,050
Temps min / max (ms)	0,024 / 0,173	0,100 / 0,302
Débit (FPS)	30 744	7 251
Confiance moyenne	0,976	0,939
Nombre d'échantillons	25	25

Table 6: Comparaison statistique entre le MLP et le CNN sur le jeu de test

7.1 Précision de classification

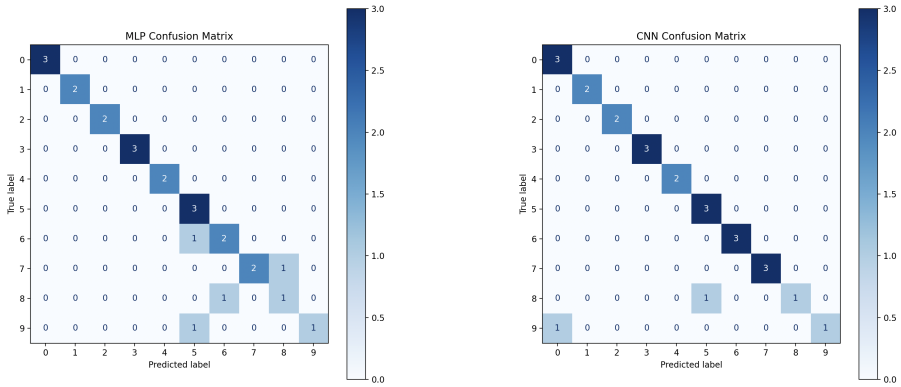
Le CNN atteint une précision de classification de **92.00%**, supérieure aux **84.00%** obtenus par le MLP sur le même jeu de données. Cette différence s'explique par la capacité du CNN à exploiter les caractéristiques spatiales locales des images grâce aux convolutions, alors que le MLP traite les images aplaties et ne capture pas efficacement la structure spatiale des chiffres.

7.2 Temps d'inférence

Le temps d'inférence moyen pour le MLP est de **0,033 ms** par image, tandis que pour le CNN il est de **0,138 ms**. Le MLP est donc environ quatre fois plus rapide que le CNN. La variance des temps d'inférence est également plus faible pour le CNN (**0,050 ms**) que pour le MLP (**0,029 ms**), reflétant une exécution plus stable sur cet échantillon.
On remarque également que la médiane est légèrement supérieure à la moyenne, ce qui indique la présence de valeurs extrêmes (confirmé par les valeurs max). Certaines images prennent plus de temps à être traitées, ce qui crée des outliers dans la distribution des temps d'inférence.
On peut expliquer que certaines images prennent plus de temps à être traitées par la complexité de leur contenu (plus de pixels d'encre, contours irréguliers).

7.3 Matrice de confusion

Pour compléter cette analyse, nous affichons les matrices de confusion pour chaque architecture, afin de visualiser les erreurs de classification pour chaque chiffre et d'identifier les classes les plus difficiles à distinguer.



(a) MLP (b) NN

Figure 6: Matrices de confusion sur les 25 images de test.

7.4 Analyse

Le MLP présente un temps d'inférence très faible et un débit extrêmement élevé, ce qui le rend adapté à des applications embarquées nécessitant un traitement rapide avec des contraintes strictes de latence. Cependant, sa précision moindre peut limiter son utilisation lorsque la fiabilité est critique.
Le CNN offre une précision supérieure grâce à sa capacité à extraire des caractéristiques spatiales complexes. Le temps d'inférence reste très faible (<0,2 ms par image), mais plus élevé que celui du MLP. Cette architecture est donc plus adaptée lorsqu'une haute précision est requise, même si le coût en temps est légèrement supérieur.

7.5 Interprétation des métriques

Il est important de noter que ces métriques doivent être interprétées à leur juste valeur. En effet, elles permettent de comparer les architectures sur une même base (même environnement et même machine, ici un M2 Pro), mais elles diffèrent des temps d'inférence que nous observons sur le Raspberry Pi 5. En pratique, les valeurs de temps mesurées ici manquent de signification quantitative pour l'embarqué, et servent surtout à fournir un ordre de grandeur et un comparatif relatif entre les architectures.

7.6 Conclusion

Ces résultats illustrent le compromis classique entre vitesse et précision : le MLP est plus rapide mais moins précis, tandis que le CNN est plus précis mais légèrement plus lent. Le choix de l'architecture dépend donc des priorités de l'application

finale.

Dans notre application déployée sur le Raspberry Pi 5, le temps d'inférence pour le CNN est d'environ 0,6 ms (soit $\approx 1667\ prdictions/sec$). Bien que ce temps concerne uniquement l'inférence et ne prenne pas en compte le temps nécessaire pour le prétraitement, avec un flux maximal de 30 FPS sur le Raspberry Pi Camera Module v3, nous pouvons facilement nous permettre d'utiliser le CNN pour bénéficier d'une meilleure précision, malgré le temps d'inférence plus élevé.

8 Guide de déploiement sur Raspberry Pi

Cette section décrit le déploiement de l'application sur un *Raspberry Pi 5* équipé du *Raspberry Pi Camera Module v3*. Ce guide peut également être retrouvé sur la page GitHub du projet.

8.1 Prérequis

- **Raspberry Pi 5 :** Docker, rpicam-vid
- **Client :** VLC

8.2 Installation

Assurez-vous que le Raspberry Pi est connecté au même réseau que votre machine hôte. Accédez-y via SSH et exécutez les commandes suivantes :

```
git clone --no-checkout git@github.com:htalerfrais/edge-ai-cnn pi5-digit-ai
cd pi5-digit-ai

git sparse-checkout init --cone
git sparse-checkout set --skip-checks run.sh docker inference_c models

git checkout main
chmod +x run.sh
```

Ces commandes permettent de cloner uniquement les fichiers et dossiers essentiels au fonctionnement de l'application afin de garder le dépôt local léger (sans les documentations ou les fichiers d'entraînement par exemple). La structure du répertoire sur la RPi5 est la suivante :

```
pi5-digit-ai
├── docker
│   └── Dockerfile
├── inference_c
│   ├── main.cpp
│   ├── Makefile
│   └── ...
├── models
│   ├── cnn_weights.txt
│   └── mlp_weights.txt
└── run.sh
```

8.3 Démarrage de l'application

Il faut construire l'image Docker et lancer l'application :

```
./run.sh all            # Build + start
./run.sh logs           # Affiche les logs
vlc "tcp://IP:8554"     # Affiche le flux depuis un client distant
```

8.4 Autres commandes utiles

```
./run.sh build          # Build l'image Docker
./run.sh start          # Démarrer le conteneur
./run.sh stop           # Arrêter le conteneur
./run.sh view           # Afficher le flux dans le terminal
```

9 Conclusion

Lors de ce projet, nous avons conçu et déployé une application de reconnaissance de chiffres manuscrits fonctionnelle sur Raspberry Pi 5, intégrant une pipeline complète, de l'acquisition vidéo à l'inférence temps réel. L'implémentation en C/C++ et l'utilisation de Docker assurent à la fois de bonnes performances et une portabilité maîtrisée.
L'étude de comparaison entre le MLP et le CNN a permis de mettre en évidence le compromis classique entre rapidité et précision. Le MLP offre des temps d'inférence très faibles mais une précision limitée sur des données réelles, quand le CNN est plus robuste et plus fiable, mais au prix d'un surcoût en calcul (modéré). Les tests sur Raspberry Pi confirment que ce surcoût reste largement compatible avec une acquisition à 30 FPS, rendant le CNN plus pertinent pour l'application finale.
Enfin, ce travail nous a permis de souligner l'importance d'un prétraitement rigoureux, aligné avec les données d'entraînement. Les objectifs fixés ont été atteints, et l'application constitue une base solide pour de futures optimisations, notamment l'ajout de modèles ou le déploiement de l'application sur la caméra d'un ordinateur portable.

Annexes

A Spécifications et performances attendues pour l'application

Critère	Insuffisant	Satisfaisant	Excellent
Précision MLP sur MNIST	< 95%	95-97%	> 97%
Précision CNN sur MNIST	< 97%	97-99%	> 99%
Précision MLP sur BDD personnel	< 60%	60-80%	> 80%
Précision CNN sur BDD personnel	< 70%	70-85%	> 85%
Précision avec caméra	< 60%	60-80%	> 80%
Temps inférence C (MLP)	> 50ms	20-50ms	< 20ms
Temps inférence C (CNN)	> 100ms	40-100ms	< 40ms
Code fonctionnel	Partiel	Complet	Optimisé

B Chargement du modèle MLP

Ce code détaille l'allocation dynamique et la lecture séquentielle des poids à partir du fichier texte généré par le script d'exportation.

```c
MLPModel* load_mlp_model(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Erreur lors de l'ouverture du fichier de poids");
        return NULL;
    }

    // Allocation de la structure principale
    MLPModel *model = (MLPModel*)malloc(sizeof(MLPModel));

    // Allocation de la memoire pour chaque couche via les dimensions de neural_network.h
    model->W1 = (float*)malloc(HIDDEN_SIZE * INPUT_SIZE * sizeof(float));
    model->b1 = (float*)malloc(HIDDEN_SIZE * sizeof(float));
    model->W2 = (float*)malloc(OUTPUT_SIZE * HIDDEN_SIZE * sizeof(float));
    model->b2 = (float*)malloc(OUTPUT_SIZE * sizeof(float));

    // Lecture sequentielle des Poids et Biais (Ordre identique a l'export Python)
    for (int i = 0; i < HIDDEN_SIZE * INPUT_SIZE; i++) {
        if (fscanf(file, "%f", &model->W1[i]) != 1) break;
    }
    for (int i = 0; i < HIDDEN_SIZE; i++) {
        if (fscanf(file, "%f", &model->b1[i]) != 1) break;
    }
    for (int i = 0; i < OUTPUT_SIZE * HIDDEN_SIZE; i++) {
        if (fscanf(file, "%f", &model->W2[i]) != 1) break;
    }
    for (int i = 0; i < OUTPUT_SIZE; i++) {
        if (fscanf(file, "%f", &model->b2[i]) != 1) break;
    }

    fclose(file);
    printf("Modele charge avec succes depuis %s\n", filename);
    return model;
}
```

Listing 1: Fonction de chargement du MLP (neural_network.c)

C Chargement du modèle CNN

```c
CNNModel* load_cnn_model(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Erreur lors de l'ouverture du fichier de poids");
        return NULL;
    }

    CNNModel *model = (CNNModel*)malloc(sizeof(CNNModel));

    // Allocation des buffers memoire
    model->conv1_w = (float*)malloc(C1_OUT_CH * C1_IN_CH * KERNEL_SIZE * KERNEL_SIZE *
        sizeof(float));
    model->conv1_b = (float*)malloc(C1_OUT_CH * sizeof(float));
    model->conv2_w = (float*)malloc(C2_OUT_CH * C2_IN_CH * KERNEL_SIZE * KERNEL_SIZE *
        sizeof(float));
    model->conv2_b = (float*)malloc(C2_OUT_CH * sizeof(float));
    model->fc_w    = (float*)malloc(OUTPUT_SIZE * FC_IN_FEATURES * sizeof(float));
    model->fc_b    = (float*)malloc(OUTPUT_SIZE * sizeof(float));

    // Lecture des blocs de poids
    int success = 1;
    success &= read_weights(file, model->conv1_w, C1_OUT_CH * C1_IN_CH * KERNEL_SIZE *
        KERNEL_SIZE);
    success &= read_weights(file, model->conv1_b, C1_OUT_CH);
    success &= read_weights(file, model->conv2_w, C2_OUT_CH * C2_IN_CH * KERNEL_SIZE *
        KERNEL_SIZE);
    success &= read_weights(file, model->conv2_b, C2_OUT_CH);
    success &= read_weights(file, model->fc_w,    OUTPUT_SIZE * FC_IN_FEATURES);
    success &= read_weights(file, model->fc_b,    OUTPUT_SIZE);

    fclose(file);
    return model;
}
```

Listing 2: Fonction de chargement du CNN (neural_network.c)