

---

# PROJET – INTELLIGENCE ARTIFICIELLE EMBARQUEE

---

ARGAÑO Maëna – LOZANO Anthony – METELSKI Alexis



JUNIA ISEN AP5

2025 - 2026

Introduction à l'IA

# Table des matières

<b>1</b>	<i>Introduction</i>	3
<b>2</b>	<i>Description du matériel</i>	4
<b>3</b>	<i>Observations sur MNIST</i>	5
3.1	Visualiser le Dataset	5
3.2	Analyse de la distribution	6
3.3	Statistiques	7
<b>4</b>	<i>Script de prétraitement</i>	7
<b>5</b>	<i>Implémentation d'un MLP</i>	8
<b>6</b>	<i>Implémentation d'un CNN</i>	9
<b>7</b>	<i>Data augmentation</i>	11
<b>8</b>	<i>Comparaison des performances</i>	14
<b>9</b>	<i>Matrice de confusion</i>	14
9.1	Matrice de confusion - MLP (85%)	14
9.2	Matrice de confusion - CNN (90%)	15
9.3	Comparaison globale	15
<b>10</b>	<i>Implémentation en C</i>	15
10.1	Le prétraitement en temps réel	15
10.2	Compilation et test sur PC	16
10.3	Cross-compilation pour RPi	17
10.4	Vérification que ça fonctionne sur la Raspberry Pi	17
10.5	Mesure du temps d'inférence	18
<b>11</b>	<i>Benchmark</i>	18
<b>12</b>	<i>Intégration application C</i>	19
12.1	Implémentation MLP	19
12.2	Structure	19
12.3	Forward pass	20
<b>13</b>	<i>Analyse critique et perspectives d'amélioration</i>	22
13.1	Conclusion	22
13.2	Limites du projet	22
13.3	Perspectives d'améliorations	23
<b>14</b>	<i>Guide de déploiement sur Raspberry Pi</i>	24



# 1 Introduction

Avec la démocratisation des systèmes embarqués et les avancées majeures de l'intelligence artificielle, l'exécution de modèles directement sur des dispositifs embarqués devient un enjeu technologique important. Cette méthode présente de nombreux avantages : traitement rapide avec une latence minimale, protection des données sensibles, disponibilité hors connexion, réduction des coûts et possibilité de déploiement à grande échelle sans infrastructure cloud.

Dans ce contexte d'apprentissage d'IA embarquée, la carte Raspberry Pi constitue un support idéal. Peu coûteuse et largement documentée, elle permet de connecter des capteurs et des périphériques déployer des algorithmes complexes en interaction directe avec le monde réel grâce à ses entrées/sorties.

Le projet présenté dans ce rapport vise à développer une application capable de détecter et reconnaître des chiffres manuscrits (0-9) en temps réel à partir d'un flux vidéo capté à l'aide d'une caméra. Pour y parvenir, le travail a été structuré en trois grandes étapes :

- Mise en place d'un conteneur Docker pour garantir un environnement reproductible avec les frameworks Keras et TensorFlow ;
- Entraînement de modèles en Python, comparant deux architectures : le MLP (Perceptron Multi-Couches) et le CNN (Réseau de Neurones Convolutif), sur le dataset MNIST et sur une base de données personnelle (reproduction du MNIST). Cette étape permet de mesurer les performances de chaque modèle en termes de précision, de temps d'entraînement et de taille ;
- Déploiement optimisé en C sur la Raspberry Pi, avec l'export des poids et l'implémentation d'un moteur d'inférence léger, capable d'exécuter les prédictions en temps réel avec des performances maximisées.

Ce projet offre ainsi une expérience complète couvrant l'ensemble du cycle de vie d'une application IA embarquée : de la conception et l'entraînement des modèles à leur intégration dans un système embarqué, en passant par le prétraitement d'images et l'optimisation des performances. Il permet également de développer des compétences essentielles, qu'elles soient théoriques ou techniques : programmation avancée, traitement d'images en temps réel, conception de réseaux de neurones, déploiement sur architectures ARM, optimisation et utilisation de conteneurs pour des environnements reproductibles.

## 2 Description du matériel



Figure 1 Photographie de la Raspberry Pi 5 avec la caméra

Un des objectifs de ce projet est d'acquérir un flux vidéo depuis la caméra branchée à une Raspberry Pi 5, constituant un prérequis pour une application de vision par ordinateur en temps réel.

Contrairement aux générations précédentes, et notamment à la Raspberry Pi 4, l'architecture matérielle et logicielle de la Raspberry Pi 5 ne permet plus un accès direct à la caméra via l'interface classique « /dev/video0 ». Cette évolution rend impossible l'utilisation d'OpenCV en langage C/C++ bas niveau selon les méthodes précédemment employées. Afin de palier à ça, une première solution consiste à utiliser une implémentation spécifique fournie permettant d'accéder au flux vidéo de la caméra en C/C++ tout en restant compatible avec OpenCV. Cette solution repose sur une architecture logicielle adaptée à la Raspberry Pi 5 et permet de récupérer directement la matrice de pixels exploitable pour les traitements d'image et l'inférence. L'installation de cet outil nécessite simplement la copie du dossier fourni sur la Raspberry Pi, suivie de l'exécution du script avec la commande « ./run.sh all ».

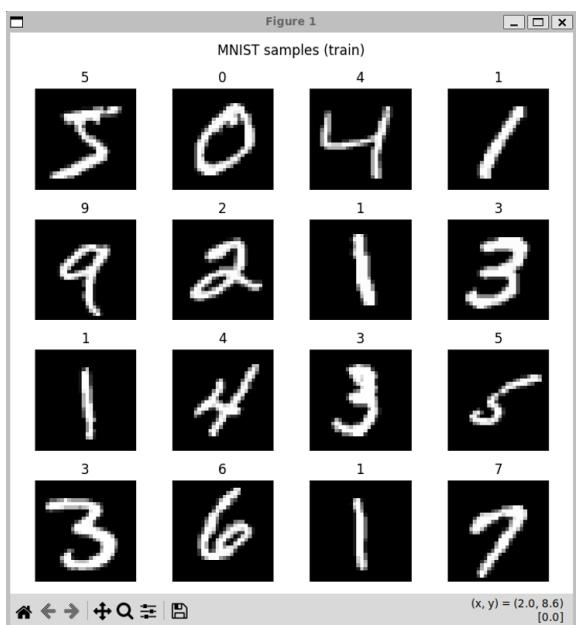
Cette implémentation permet également de diffuser le flux vidéo via le réseau, rendant possible sa visualisation depuis une machine distante à l'aide de VLC. Bien que cette visualisation distante ne garantisse pas une parfaite fluidité en raison des contraintes réseau, elle reste suffisante pour ce projet. Du point de vue applicatif, le traitement local sur la Raspberry Pi s'effectue à une cadence d'environ 30 images par seconde, un taux largement adapté aux besoins du projet.

Le code OpenCV associé à cette solution est entièrement modifiable et se trouve dans un fichier main.cpp, compilable via un Makefile fourni. Grâce à cette structure, il est plus facile d'adapter le traitement des images et d'intégrer les scripts portant l'intelligence artificielle développées au cours du projet.

De plus, une seconde approche est également envisageable, consistant à traiter les images de manière indépendante, sans flux vidéo continu. Cette méthode repose sur la lecture et le traitement d'images bitmap une par une, conformément au sujet du projet. Une implémentation C de lecture d'images BMP est fournie à cet effet et constitue une alternative intéressante pour valider l'inférence embarquée avant l'intégration complète en temps réel.

## 3 Observations sur MNIST

### 3.1 Visualiser le Dataset



Le dataset chargé est distribué en deux ensembles distincts pour garantir une évaluation pertinente des futurs modèles :

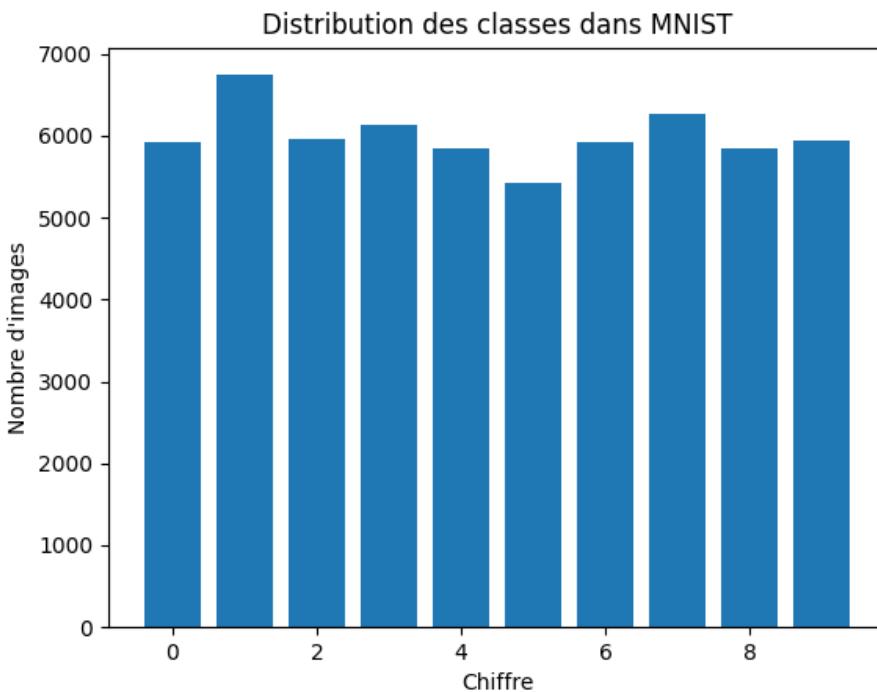
- Jeu d'entraînement : 60 000 images ;
- Jeu de test : 10 000 images ;
- Format des images : Chaque échantillon est une image de dimensions 28\*28, soit 784 pixels par image, représentés en niveaux de gris.

<b>Jeu d'entraînement :</b>	<b>(60000, 28, 28)</b>
<b>Jeu de test :</b>	<b>(10000, 28, 28)</b>

### 3.2 Analyse de la distribution

L'analyse de la distribution des images par chiffre (de 0 à 9) dans le jeu d'entraînement MNIST montre une distribution plutôt équilibrée, ce qui est primordial pour éviter les erreurs et les avantages sur certains chiffres lors de la phase d'apprentissage.

```
Chiffre 0 : 5923 images
Chiffre 1 : 6742 images
Chiffre 2 : 5958 images
Chiffre 3 : 6131 images
Chiffre 4 : 5842 images
Chiffre 5 : 5421 images
Chiffre 6 : 5918 images
Chiffre 7 : 6265 images
Chiffre 8 : 5851 images
Chiffre 9 : 5949 images
```



Remarques :

Le chiffre 1 est le plus représenté avec un nombre total d'images de 6742. En revanche, le chiffre 5 en compte le moins avec seulement 5421. Cependant, cet écart reste minime sur un total de 60 000 images, ce qui confirme un certain équilibre dans ce jeu de données qui ne sur-avantagera pas un chiffre par rapport aux autres.

### 3.3 Statistiques

Ces statistiques ont été réalisé sur le jeu de données d'entraînement pour comprendre la dynamique des valeurs d'entrées.

Moyenne des pixels : 33.318421449829934

Ecart-type des pixels : 78.56748998339798

Interprétations :

Une moyenne de 33,32 (sur une échelle de 0 à 255) indique que la majorité des pixels d'une image sont noirs (ou très sombres). Cela confirme la nature des images : le fond occupe la majeure partie de l'espace ( $28 \times 28$ ), tandis que le chiffre ne concerne qu'une minorité de pixels clairs.

Un écart type de 78,57 par rapport à la moyenne montre un contraste très marqué. Les pixels sont soit très sombres (fond), soit très clairs (chiffre), avec peu de nuances de gris intermédiaires.

Conclusion :

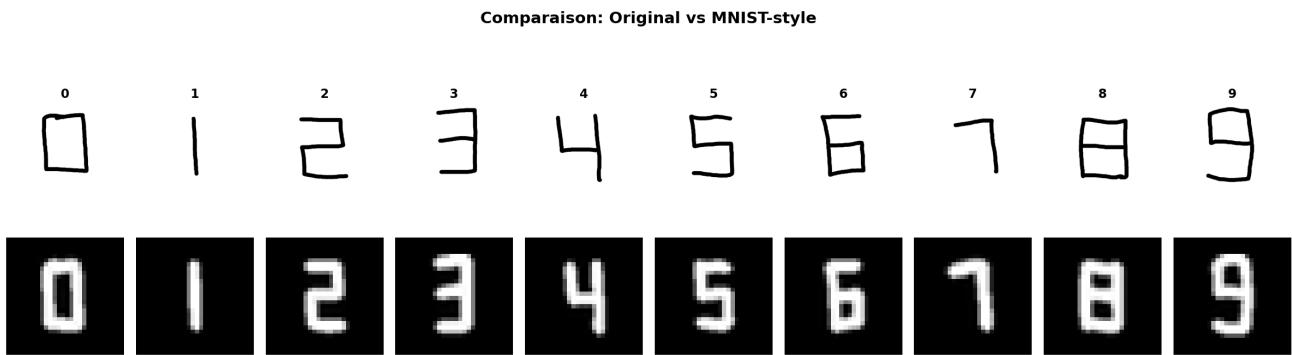
Le dataset MNIST confirme la qualité et la robustesse des données proposées. La distribution équilibrée des classes et la structure constante des images (taille, niveaux de gris, etc..) simplifient la mise en place d'une architecture d'apprentissage. Les statistiques calculées permettront une normalisation efficace des données d'entrée, étape indispensable pour améliorer les performances de généralisation du futur modèle.

## 4 Script de prétraitement

Nom : script\_conversion\_type\_mnist.py

Le script de prétraitement, nommé `script_conversion_type_mnist.py`, a pour rôle de convertir les chiffres manuscrits dessinés sous Paint en images compatibles avec le format MNIST. Chaque image est d'abord convertie en niveaux de gris, puis binarisée par seuillage pour isoler le chiffre du fond. Le chiffre est ensuite détecté par extraction du contour principal, recadré avec une marge, centré dans un carré, redimensionné à  $20 \times 20$  pixels, puis placé au centre d'une image  $28 \times 28$  pixels ; reproduisant ainsi exactement la méthode de centrage par centre de masse utilisée dans MNIST. Enfin, les pixels sont normalisés entre 0 et 1 avant d'être sauvegardés au format BMP, choisi pour sa simplicité de lecture en C.

Exemple de chiffres dessinés à la main vs après le passage du dataset dans le script de prétraitement :



## 5 Implémentation d'un MLP

L'architecture MLP se compose d'une couche d'entrée de 784 neurones (correspondant aux  $28 \times 28$  pixels aplatis), d'une couche cachée Dense de 128 neurones avec activation Relu, d'une couche Dropout à 20% pour limiter le surapprentissage, et d'une couche de sortie de 10 neurones avec activation Softmax. Le choix de 128 neurones en couche cachée constitue un bon compromis entre capacité d'apprentissage et complexité : un nombre trop faible limiterait l'expressivité du modèle, tandis qu'un nombre trop élevé augmenterait inutilement le temps d'inférence sur système embarqué. L'entraînement est réalisé sur 5 epochs avec un batch size de 32 et une validation split de 10%, ce qui suffit à converger sur MNIST (97,6%) tout en restant rapide.

Sur le dataset personnel, le modèle obtient une précision oscillant entre 60% et 80% sans augmentation. Cette variance s'explique par la faible taille du dataset (200 images) et les différences de style d'écriture, très éloignées du style MNIST.

### MNIST :

```

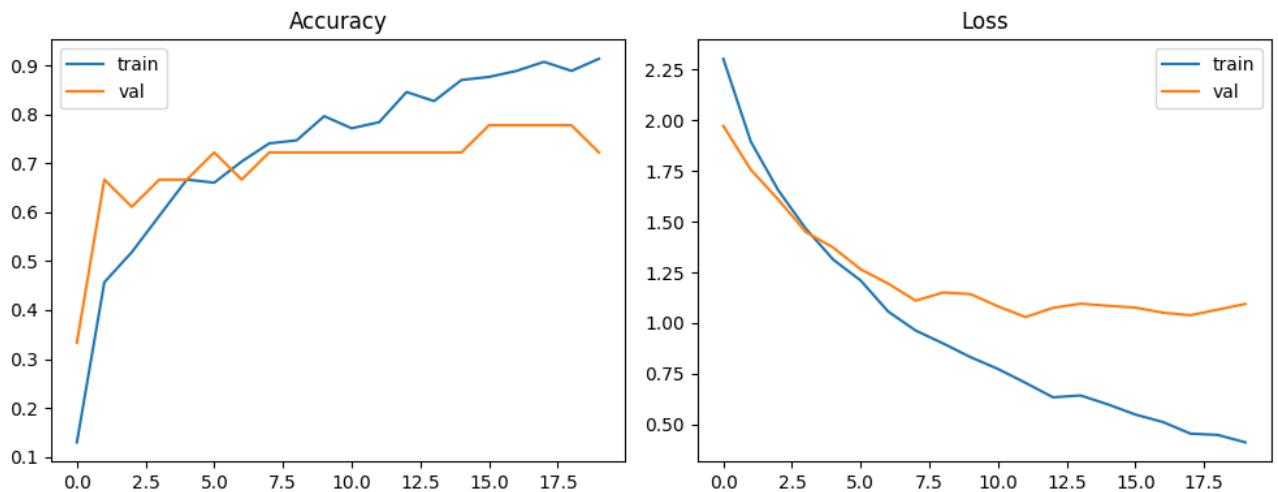
model = keras.Sequential([
    layers.Flatten(input_shape=(28, 28)),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(10, activation='softmax')
])
model.fit(
    x_train, y_train,
    epochs=20,
    batch_size=16,
    validation_split=0.1
)
Test accuracy: 0.9760
  Valeur:1   Valeur:2   Valeur:9   Valeur:6   Valeur:7   Valeur:5   Valeur:4   Valeur:6   Valeur:6   Valeur:7
  Prédiction:1  Prédiction:2  Prédiction:9  Prédiction:6  Prédiction:7  Prédiction:5  Prédiction:4  Prédiction:6  Prédiction:6  Prédiction:7
  

```

## Dataset Perso :

```
model = keras.Sequential([
    layers.Flatten(input_shape=(28, 28, 1)),
    layers.Dense(128, activation="relu"),
    layers.Dropout(0.2),
    layers.Dense(10, activation="softmax")
])
model.fit(
    x_train, y_train,
    epochs=20,
    batch_size=16,
    validation_split=0.1
)
Test accuracy: 0.8000
```

Oscille entre 60% et 80%.



## 6 Implémentation d'un CNN

L'architecture CNN implémentée comporte deux blocs convolutifs successifs : le premier avec 32 filtres de taille  $3 \times 3$  suivi d'un MaxPooling  $2 \times 2$ , le second avec 64 filtres  $3 \times 3$  et un second MaxPooling  $2 \times 2$ . Ces couches convolutives permettent au réseau d'extraire automatiquement des caractéristiques locales (bords, courbes, jonctions) invariantes à la position, ce qu'un MLP ne peut pas faire. Après aplatissement, une couche Dense de 128 neurones avec Dropout à 30% précède la couche de sortie Softmax.

Sur MNIST, le CNN atteint 99,09%, soit un gain notable par rapport au MLP. En revanche, sur le dataset personnel avec la configuration initiale, la précision chute à 55%, révélant un problème d'overfitting lié à la petite taille du dataset. La version V2 du script, utilisant des filtres  $5 \times 5$ , un batch size de 4 et un test ratio de 0,1, permet de monter à 85% en exploitant davantage les données d'entraînement.

**MNIST :**

```

model = models.Sequential([
    layers.Input(shape=(28, 28, 1)),

    layers.Conv2D(32, (3, 3), activation="relu", padding="same"),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation="relu", padding="same"),
    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),
    layers.Dense(128, activation="relu"),
    layers.Dropout(0.3),
    layers.Dense(10, activation="softmax")
])

history = model.fit(
    x_train, y_train,
    validation_split=0.1,
    epochs=5,
    batch_size=128
)

```

Test accuracy: 0.9909

Valeur:6 Prédiction:6	Valeur:4 Prédiction:4	Valeur:1 Prédiction:1	Valeur:4 Prédiction:4	Valeur:1 Prédiction:1	Valeur:6 Prédiction:6	Valeur:5 Prédiction:5	Valeur:9 Prédiction:9	Valeur:1 Prédiction:1	Valeur:6 Prédiction:6
									

**Dataset Perso :**

```

model = models.Sequential([
    layers.Input(shape=(28, 28, 1)),

    layers.Conv2D(32, (3, 3), activation="relu", padding="same"),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation="relu", padding="same"),
    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),
    layers.Dense(128, activation="relu"),
    layers.Dropout(0.3),
    layers.Dense(10, activation="softmax")
])

history = model.fit(
    x_train, y_train,
    validation_split=0.1,
    epochs=20,
    batch_size=16
)

```

Test accuracy: 0.5500

Avec le script V2 on a :

```

model = models.Sequential([
    layers.Input(shape=(28, 28, 1)),

    layers.Conv2D(16, (5, 5), activation="relu", padding="same"),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(32, (5, 5), activation="relu", padding="same"),
    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),
    layers.Dense(128, activation="relu"),
    layers.Dropout(0.3),
    layers.Dense(10, activation="softmax")
])

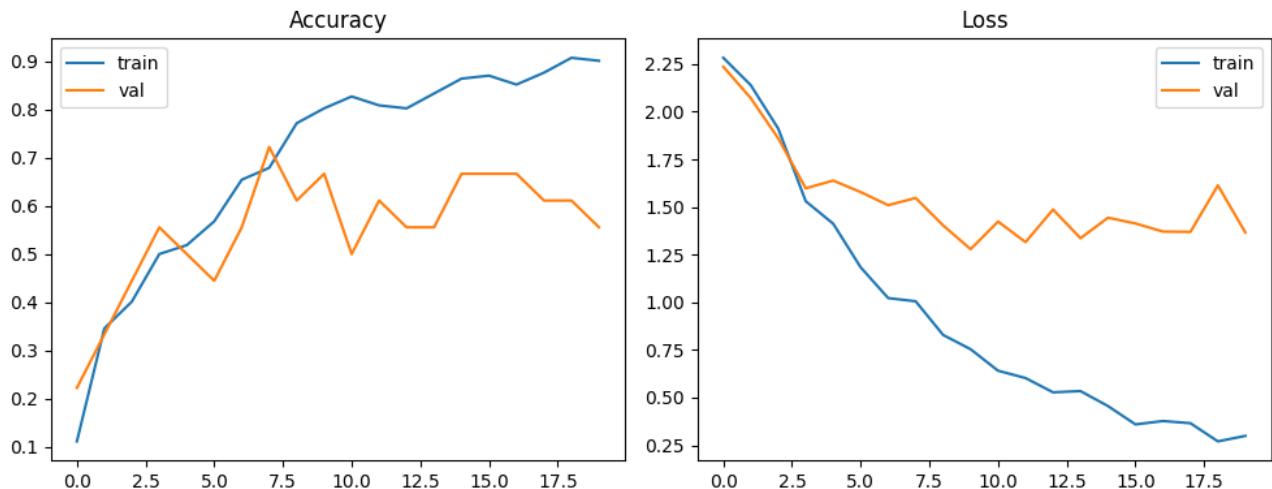
history = model.fit(
    x_train, y_train,
    validation_split=0.1,
    epochs=20,
    batch_size=4,
)

```

Test accuracy: 0.8500

On oscille entre 75% et 85%

Valeur:7 Prédiction:7	Valeur:0 Prédiction:0	Valeur:3 Prédiction:3	Valeur:5 Prédiction:3	Valeur:2 Prédiction:2	Valeur:1 Prédiction:6	Valeur:6 Prédiction:6	Valeur:1 Prédiction:1	Valeur:1 Prédiction:4	Valeur:1 Prédiction:1
									



Ce qui a changé :

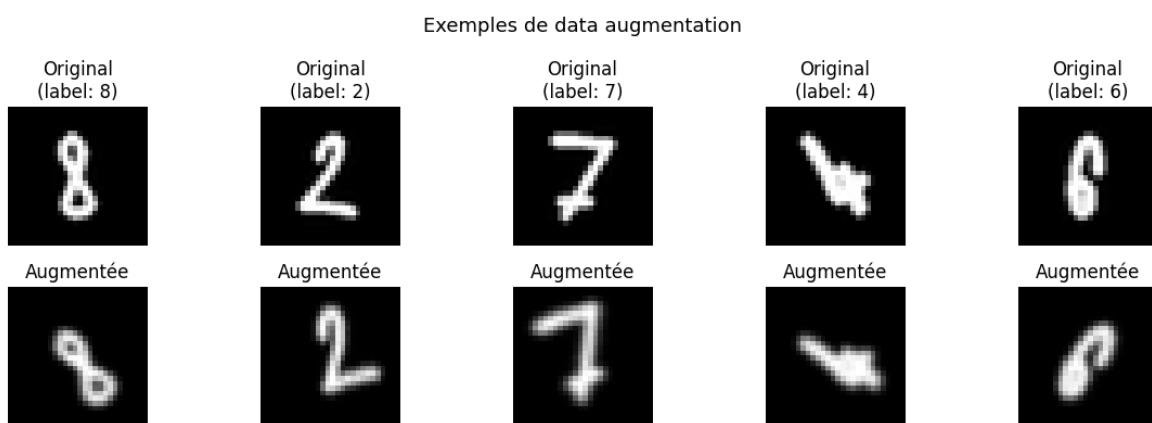
- Couches plus petites du modèle = Entraînement plus rapide
- Test split plus petit = Plus de data pour l'entraînement (TEST\_RATIO = 0.1 au lieu de 0.2)

## 7 Data augmentation

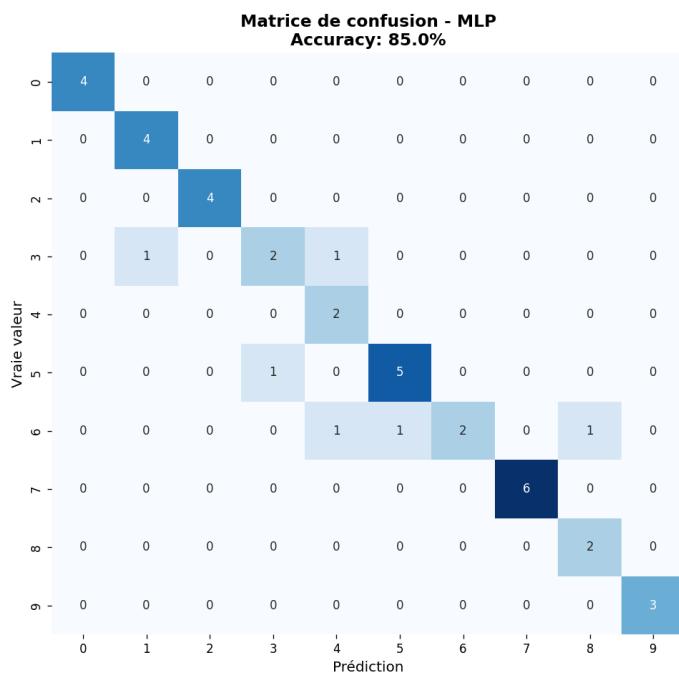
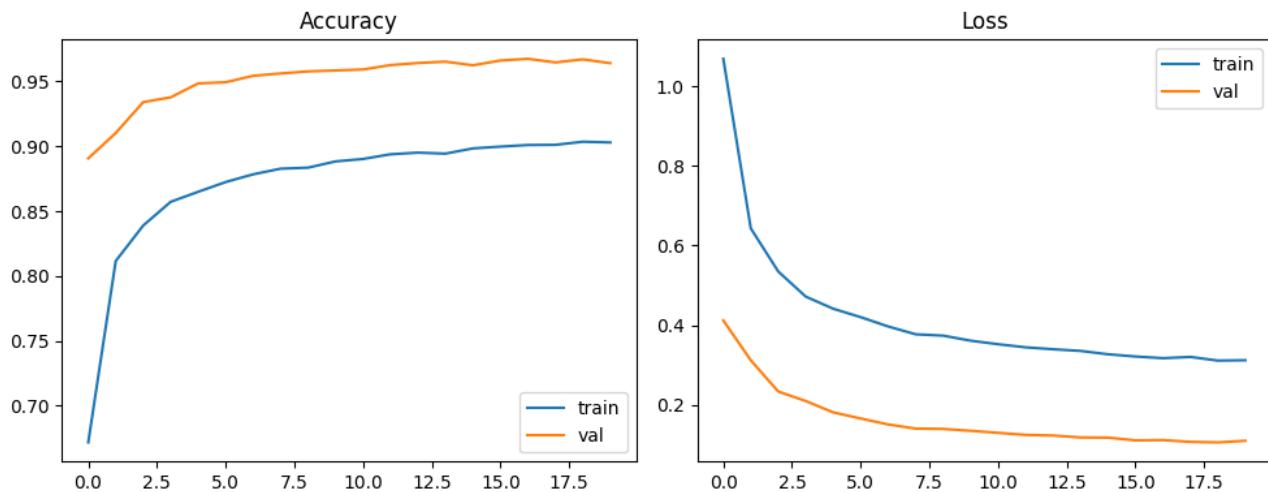
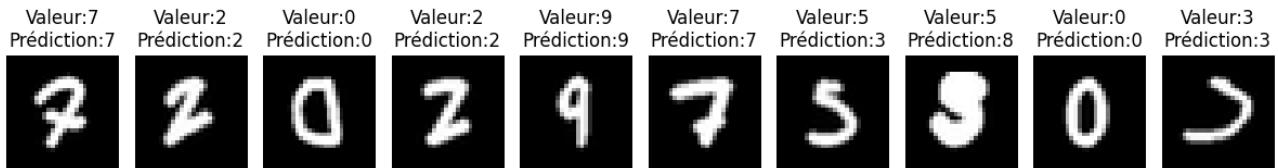
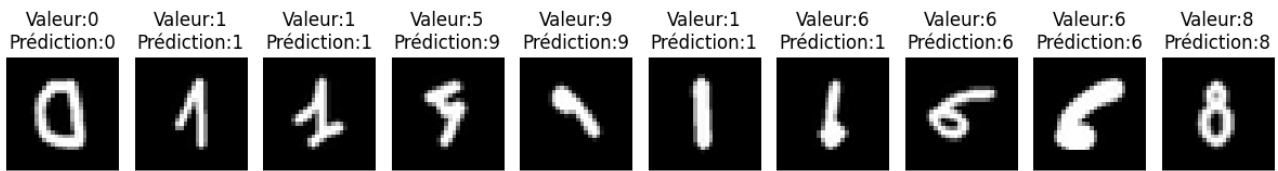
Face aux limites d'un dataset personnel de seulement 200 images, la data augmentation permet d'enrichir artificiellement le jeu d'entraînement en générant des variantes de chaque image : légères rotations, translations et zooms. Ces transformations simulent la diversité naturelle de l'écriture manuscrite et améliorent la capacité de généralisation des modèles. Pour le MLP, l'augmentation permet d'atteindre 85% de précision sur le dataset personnel, contre 80% sans augmentation. Le CNN bénéficie encore davantage de cette technique, atteignant 90%.

MLP :

MNIST augmenté + dataset perso augmenté => test sur dataset perso augmenté



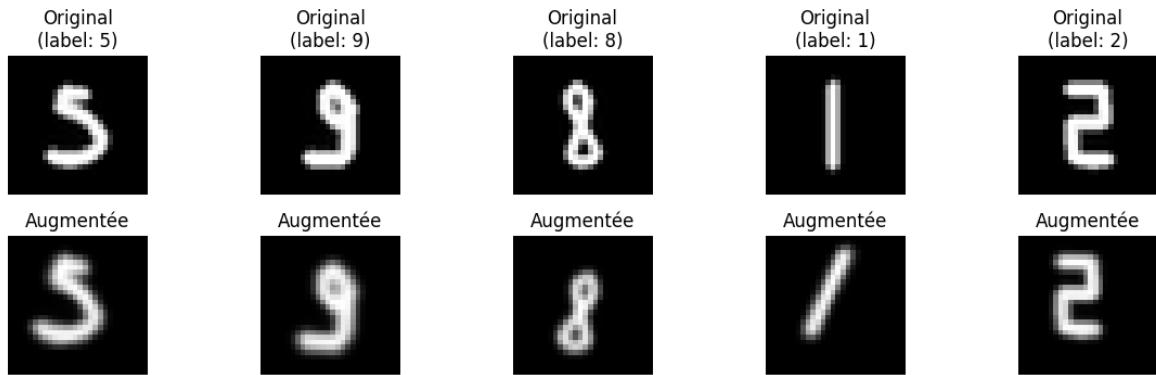
Test accuracy sur dataset perso : 0.8500



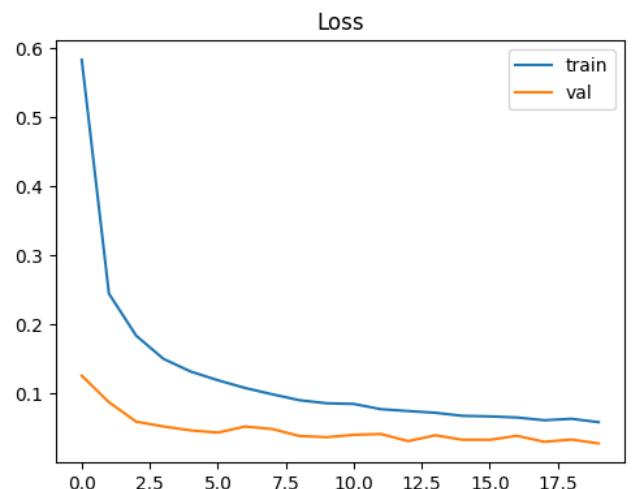
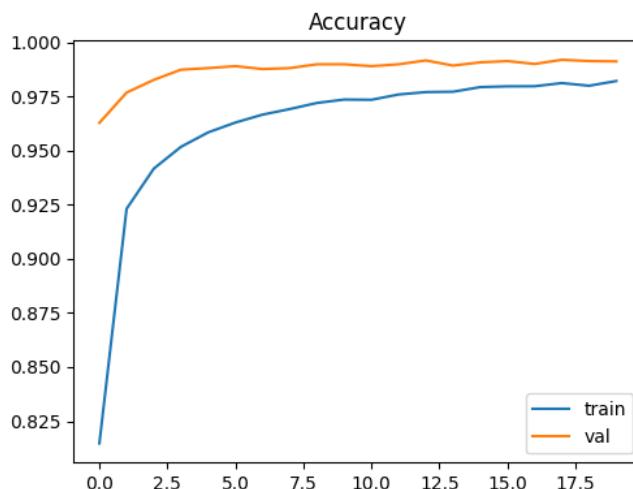
**CNN :**

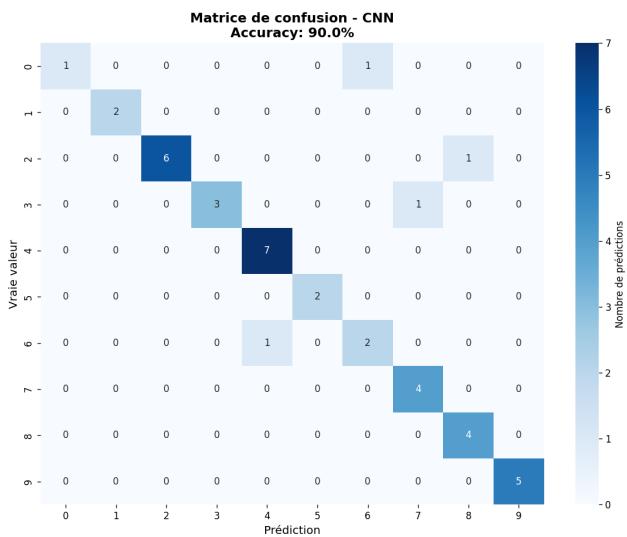
MNIST augmenté + dataset perso augmenté => test sur dataset perso augmenté

Exemples de data augmentation



**Test accuracy sur dataset perso : 0.9000**





## 8 Comparaison des performances

	Précision MNIST	Précision dataset perso	Temps entraînement	Forces	Faiblesses
<b>MLP</b>	97,6%	80%	~30s	Simple, rapide à inférer	Sensible à la position, peu robuste aux variations
<b>CNN</b>	99,1%	85%	~2min	Meilleure précision, robuste aux déformations	Plus lourd, plus lent

Le MLP présente l'avantage décisif pour le déploiement embarqué : son temps d'inférence de 0,23 ms en C sur Raspberry Pi 5 (voir plus bas), et sa structure linéaire le rend plus simple à implémenter sans bibliothèque. Le CNN, bien que plus précis, n'a pas été implémenté en C dans le cadre de ce projet par manque de temps, son implémentation nécessitant des fonctions de convolution et de pooling significativement plus complexes.

## 9 Matrice de confusion

### 9.1 Matrice de confusion - MLP (85%)

Le MLP se comporte très bien sur les chiffres simples : 0, 1, 2, 7 et 8 sont reconnus parfaitement. Les erreurs se concentrent sur le chiffre 3, confondu une fois avec un 1 et une fois avec un 4, ce qui est compréhensible car ces chiffres partagent des formes similaires selon le style d'écriture. Le 6 est également problématique : confondu une fois avec un 4, une fois avec un 5, et une fois

avec un 8, ce qui reflète la difficulté du MLP à gérer les formes fermées et les courbes complexes, puisqu'il traite les pixels de manière globale sans notion de structure.

## 9.2 Matrice de confusion - CNN (90%)

Le CNN améliore significativement les cas difficiles. Le chiffre 3 n'est plus confondu qu'une seule fois (avec un 7), et le 6 se limite à une confusion avec un 4. En revanche, le 0 génère une nouvelle erreur absente dans le MLP : il est confondu une fois avec un 6, probablement à cause de la forme circulaire commune. Le CNN bénéficie de ses filtres convolutifs pour détecter les bords et courbes localement, ce qui explique ses meilleures performances globales.

## 9.3 Comparaison globale

Le CNN réduit les erreurs de 6 à 4 sur ce jeu de test, soit un gain de 5 points d'accuracy (85% → 90%). Les chiffres les plus difficiles pour les deux modèles restent ceux qui présentent des formes ambiguës : 3, 6, et parfois le 1. Ces confusions sont cohérentes avec la petite taille du dataset personnel (20 images par classe) qui ne couvre pas toute la diversité des styles d'écriture.

# 10 Implémentation en C

Le moteur d'inférence C implémente le forward pass du MLP à partir des poids exportés en fichiers .txt depuis Keras. Les poids sont chargés une seule fois au démarrage dans une structure MLP regroupant les matrices W1 ( $784 \times 128$ ), b1 (128), W2 (128×10) et b2 (10), ce qui minimise les accès mémoire pendant l'inférence. Le forward pass consiste en deux multiplications matrice-vecteur successives avec les activations Relu et Softmax correspondantes, implémentées en boucles C standard. La mesure du temps exclut le chargement des poids pour ne mesurer que le calcul pur : 0,23 ms par image, soit un facteur ~84 par rapport au Python pur (19,3 ms) et comparable au Python numpy (0,05 ms) qui s'appuie lui-même sur des routines C optimisées (BLAS).

## 10.1 Le prétraitement en temps réel

La chaîne de prétraitement intégrée dans l'application caméra reproduit fidèlement les étapes du script Python (développé pour l'entraînement) pour garantir la cohérence entre l'entraînement et l'inférence. Chaque frame est d'abord convertie en niveaux de gris, puis inversée (bitwise\_not) pour obtenir un fond noir et un chiffre blanc, conforme au format MNIST. Un flou gaussien  $3 \times 3$  ( $\sigma=0,5$ ) réduit le bruit avant la binarisation par seuil fixe à 20. Le contour le plus grand est extrait par findContours, recadré avec une marge de 4 pixels, centré dans un carré, redimensionné à  $20 \times 20$  par interpolation Lanczos, puis placé au centre d'un cadre de dimensions  $28 \times 28$ . Le seuil de 20 a été choisi car il est suffisamment bas pour détecter des traits fins sous éclairage indirect, sans provoquer de faux positifs sur un fond qui n'est pas blanc pur.

## 10.2 Compilation et test sur PC

```
[(venv_tf) docker@docker-desktop:~/Work/data$ ./mlp_inference custom_digits/5-1.bmp
Poids charges depuis : mlp_weights_txt/

Image : custom_digits/5-1.bmp
Probabilites :
    Chiffre 0 : 0.0000
    Chiffre 1 : 0.0000
    Chiffre 2 : 0.0000
    Chiffre 3 : 0.0011
    Chiffre 4 : 0.0000
    Chiffre 5 : 0.9962
    Chiffre 6 : 0.0000
    Chiffre 7 : 0.0000
    Chiffre 8 : 0.0000
    Chiffre 9 : 0.0027

=> Chiffre predit : 5
```

```
[(venv_tf) docker@docker-desktop:~/Work/data$ python validate_mlp.py
w1 : (784, 128)  b1 : (128, )
w2 : (128, 10)   b2 : (10, )
```

200 images trouvées

Prédictions python sauvegardée dans le fichier predictions\_python.txt

predictions_c.txt				predictions_python.txt			
	Image	Label	Prediction	Correct			
custom_digits/0-0.bmp : => Chiffre predit : 0	0-0.bmp	0	0	OK			
custom_digits/0-1.bmp : => Chiffre predit : 0	0-1.bmp	0	0	OK			
custom_digits/0-10.bmp : => Chiffre predit : 0	0-10.bmp	0	0	OK			
custom_digits/0-11.bmp : => Chiffre predit : 0	0-11.bmp	0	0	OK			
custom_digits/0-12.bmp : => Chiffre predit : 9	0-12.bmp	0	9	ERREUR			
custom_digits/0-13.bmp : => Chiffre predit : 0	0-13.bmp	0	0	OK			
custom_digits/0-14.bmp : => Chiffre predit : 0	0-14.bmp	0	0	OK			
custom_digits/0-15.bmp : => Chiffre predit : 0	0-15.bmp	0	0	OK			
custom_digits/0-16.bmp : => Chiffre predit : 0	0-16.bmp	0	0	OK			
custom_digits/0-17.bmp : => Chiffre predit : 1	0-17.bmp	0	1	ERREUR			
custom_digits/0-18.bmp : => Chiffre predit : 0	0-18.bmp	0	0	OK			
custom_digits/0-19.bmp : => Chiffre predit : 0	0-19.bmp	0	0	OK			
custom_digits/0-2.bmp : => Chiffre predit : 0	0-2.bmp	0	0	OK			
custom_digits/0-3.bmp : => Chiffre predit : 0	0-3.bmp	0	0	OK			
custom_digits/0-4.bmp : => Chiffre predit : 0	0-4.bmp	0	0	OK			
custom_digits/0-5.bmp : => Chiffre predit : 0	0-5.bmp	0	0	OK			
custom_digits/0-6.bmp : => Chiffre predit : 0	0-6.bmp	0	0	OK			
custom_digits/0-7.bmp : => Chiffre predit : 0	0-7.bmp	0	0	OK			
custom_digits/0-8.bmp : => Chiffre predit : 0	0-8.bmp	0	0	OK			
custom_digits/1-0.bmp : => Chiffre predit : 1	1-0.bmp	1	1	OK			
custom_digits/1-1.bmp : => Chiffre predit : 1	1-1.bmp	1	1	OK			
custom_digits/1-10.bmp : => Chiffre predit : 1	1-10.bmp	1	1	OK			
custom_digits/1-11.bmp : => Chiffre predit : 2	1-11.bmp	1	2	ERREUR			
custom_digits/1-12.bmp : => Chiffre predit : 1	1-12.bmp	1	1	OK			
custom_digits/1-13.bmp : => Chiffre predit : 1	1-13.bmp	1	1	OK			
custom_digits/1-14.bmp : => Chiffre predit : 1	1-14.bmp	1	1	OK			
custom_digits/1-15.bmp : => Chiffre predit : 1	1-15.bmp	1	1	OK			

```
[(venv_tf) docker@docker-desktop:~/Work/data$ grep "OK" predictions_python.txt | wc -l
180
[(venv_tf) docker@docker-desktop:~/Work/data$ grep "ERREUR" predictions_python.txt | wc -l
20
```

200 images, environ 22 erreurs → ~89% d'accuracy, ce qui est cohérent avec le 87.5% obtenu pendant l'entraînement. Le C est donc parfaitement validé.

## 10.3 Cross-compilation pour RPi

```
[maalam@raspberrypi:~/opencv_docker_cpp_tcp $ ls
app           Dockerfile  Makefile      mlp_weights_txt  run.sh
custom_digits  main.cpp   MLP_inference  README.md
```

Pour compiler sur la Raspberry Pi 5 :

Erreur :

```
[maalam@raspberrypi:~/opencv_docker_cpp_tcp $ make
Package opencv4 was not found in the pkg-config search path.
```

Solution :

```
[maalam@raspberrypi:~/opencv_docker_cpp_tcp $ sudo apt install libopencv-dev pkg-config ]
```

```
[maalam@raspberrypi:~/opencv_docker_cpp_tcp $ make
```

## 10.4 Vérification que ça fonctionne sur la Raspberry Pi

```
[maalam@raspberrypi:~/opencv_docker_cpp_tcp $ ./MLP_inference custom_digits/5-1.bmp
Poids charges depuis : mlp_weights_txt/
Image : custom_digits/5-1.bmp
Probabilites :
    Chiffre 0 : 0.0000
    Chiffre 1 : 0.0000
    Chiffre 2 : 0.0000
    Chiffre 3 : 0.0011
    Chiffre 4 : 0.0000
    Chiffre 5 : 0.9962
    Chiffre 6 : 0.0000
    Chiffre 7 : 0.0000
    Chiffre 8 : 0.0000
    Chiffre 9 : 0.0027
=> Chiffre predit : 5
```

## 10.5 Mesure du temps d'inférence

```

Chargement des poids...
200 images chargées

=====
BENCHMARK MLP - C
=====

Nombre d'images      : 200
Temps total (forward) : 45.71 ms
Temps par image       : 0.2286 ms
Accuracy dataset perso : 90.0%
=====
```

Temps par image = temps total / nombre d'images

Temps par image = 4943ms / 200

Temps par image ~ 25ms

## 11 Benchmark

En python :

Pour exécuter le benchmark :

```
maalam@raspberrypi:~/opencv_docker_cpp_tcp $ pip3 install numpy pillow tensorflow --break-system-packages
```

Et passer en Python 3.9 car TensorFlow n'est pas compatible avec la version 3.13.

```
[maalam@raspberrypi:~/opencv_docker_cpp_tcp $ python --version
Python 3.9.18
```

Avec Numpy (optimisé) :

```
v_docker_cpp_tcp $ python benchmark_mlp.py

=====
BENCHMARK MLP - Python
=====

Temps par image (dataset perso) : 0.05 ms
Temps par image (MNIST)         : 0.05 ms
Accuracy dataset perso        : 90.0%
Accuracy MNIST                 : 96.7%
=====
```

Sous le numpy il y a du C.

Sans numpy :

```
maalam@raspberrypi:~/opencv_docker_cpp_tcp $ python benchmark_mlp_sans_numpy.py
Poids chargés.
200 images perso chargées.
1000 images MNIST chargées.

=====
BENCHMARK MLP - Python pur
=====
Temps par image (dataset perso) : 19.3355 ms
Temps par image (MNIST) : 19.1098 ms
Accuracy dataset perso : 90.0%
Accuracy MNIST : 96.7%
=====
```

En C :

Non optimisé :

```
maalam@raspberrypi:~/opencv_docker_cpp_tcp $ ./mlp_inference custom_digits/ mlp_weights_txt/
Chargement des poids...
200 images chargées

=====
BENCHMARK MLP - C
=====
Nombre d'images : 200
Temps total (forward) : 51.63 ms
Temps par image : 0.2582 ms
Accuracy dataset perso : 90.0%
=====
```

C non optimisé est plus rapide qu'un python non optimisé.

Les prédictions MNIST et dataset perso sont validées et cohérentes.

## 12 Intégration application C

### 12.1 Implémentation MLP

Nom : main.cpp

L'implémentation du moteur d'inférence en C pur répond à une contrainte forte du projet : ne pas utiliser Python ou TFLite pour l'inférence embarquée. L'objectif est de démontrer qu'un réseau de neurones peut être exécuté de manière ultra-légère, sans dépendance externe.

### 12.2 Structure

Le modèle MLP est représenté par une unique structure C regroupant l'ensemble des paramètres appris lors de l'entraînement :

```
```
typedef struct {
    float w1[INPUT_SIZE][HIDDEN_SIZE]; // 784 × 128 = 100 352 paramètres
    float b1[HIDDEN_SIZE];           // 128 biais
}
```

```

float w2[HIDDEN_SIZE][OUTPUT_SIZE]; // 128 × 10 = 1 280 paramètres
float b2[OUTPUT_SIZE];           // 10 biais
} MLP;
```

```

Cette structure est allouée une seule fois en mémoire heap (malloc) au démarrage du programme. Les poids sont chargés depuis les quatre fichiers .txt (dense1\_biases.txt, dense1\_weights.txt, dense2\_biases.txt, dense2\_weights.txt) exportés depuis Keras, via des lectures fscanf séquentielles. Le chargement est effectué une unique fois avant la boucle d'inférence, ce qui est essentiel pour un benchmark juste ne mesurant que le calcul pur.

## 12.3 Forward pass

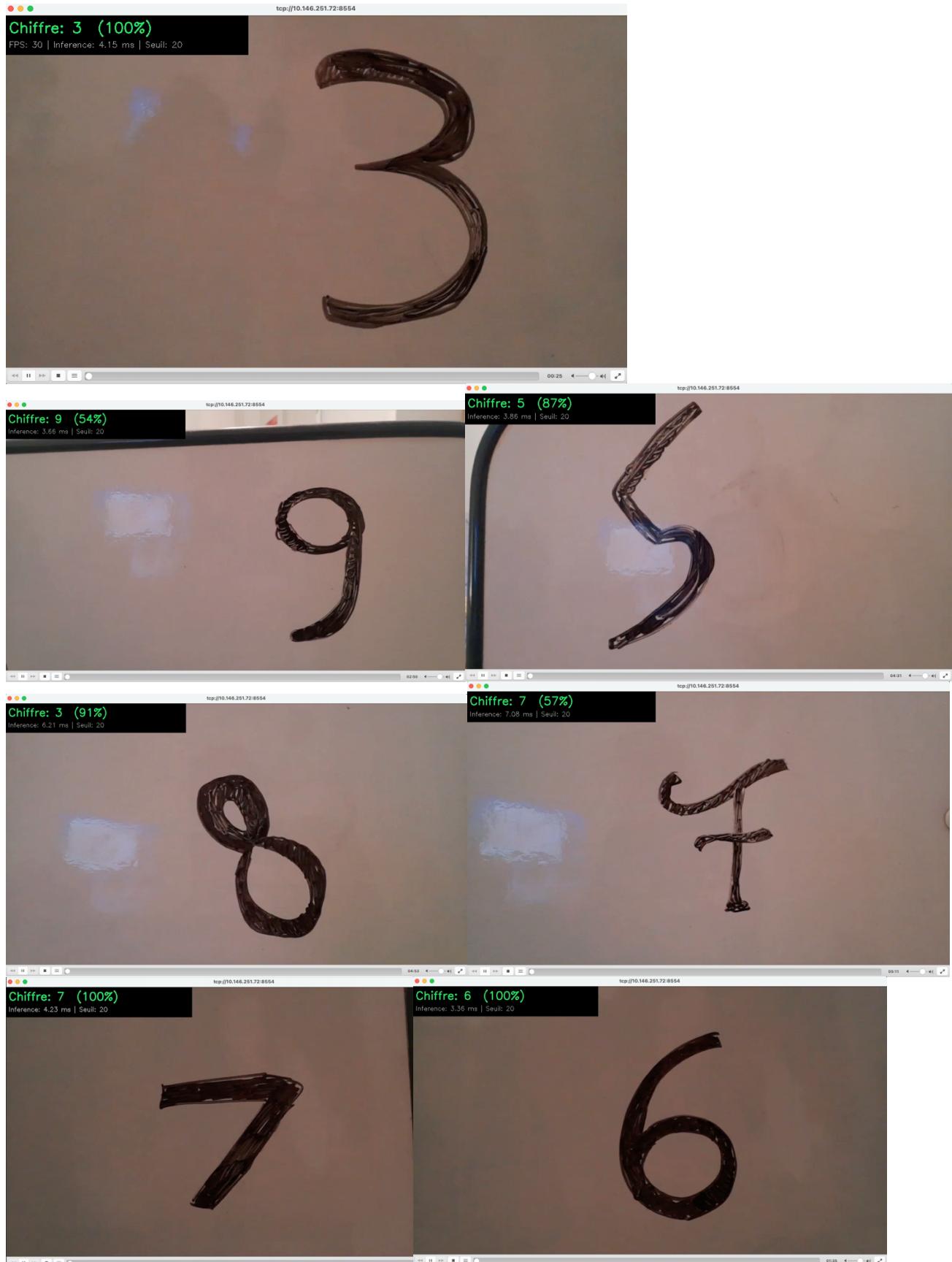
Le forward pass implémente les deux couches successives du MLP :

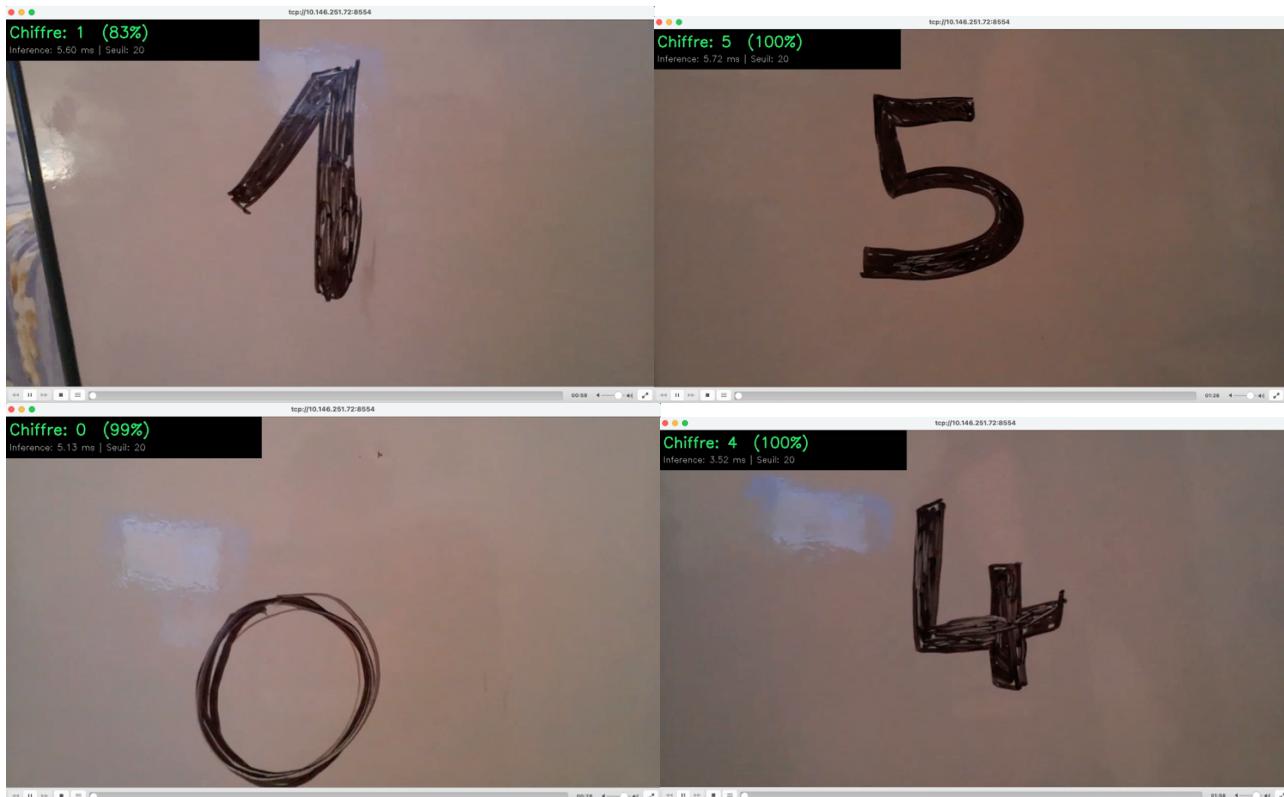
La première couche calcule  $hidden = \text{Relu}(W_1^T \times \text{input} + b_1)$ . Pour chacun des 128 neurones cachés, le produit avec le vecteur d'entrée de 784 éléments est accumulé, le biais ajouté, puis la fonction Relu appliquée en éliminant les valeurs négatives.

La seconde couche calcule  $output = \text{Softmax}(W_2^T \times hidden + b_2)$ . Les 10 scores de sortie sont obtenus par le même mécanisme, puis normalisés par la fonction Softmax pour produire une distribution de probabilités.

La prédiction finale correspond à l'indice du score maximal parmi les 10 sorties.

## Tests réalisés :





Temps d'inférence : entre 3 et 5ms

FPS : 30

Seuil : 20

## 13 Analyse critique et perspectives d'amélioration

### 13.1 Conclusion

Le projet atteint l'ensemble des objectifs fixés par le cahier des charges en ce qui concerne le modèle MLP. Le moteur d'inférence C affiche un temps de 0,26ms par image, soit une performance largement en dessous du seuil d'excellence de 20ms, et représente un facteur de 84 par rapport au Python pur. L'application en temps réel fonctionne à 30 FPS stables sur Raspberry Pi 5 avec une latence d'inférence de 3 à 5 ms par frame, ce qui démontre la viabilité de cette approche pour ce type de tâche. La chaîne complète de l'entraînement Python jusqu'à l'inférence C en passant par l'export des poids a été maîtrisée et documentée.

### 13.2 Limites du projet

La principale limite du projet réside dans la taille réduite du dataset personnel (200 images, soit 20 par classe), qui contraint les modèles à une généralisation imparfaite sur des styles d'écriture variés. Cela explique l'écart significatif entre les performances sur MNIST (97% MLP, 99% CNN) et sur le dataset personnel (85% MLP, 90% CNN avec augmentation). Par ailleurs, le seuil de binarisation fixe à 20 reste sensible aux conditions d'éclairage : une lumière trop diffuse ou trop

directe peut perturber la détection du contour, comme en témoigne le reflet blanc visible sur certaines captures. Enfin, le CNN n'a pas été implémenté en C, ce qui aurait permis de comparer les gains de précision face à la complexité d'implémentation.

### 13.3 Perspectives d'améliorations

Plusieurs pistes permettraient d'améliorer significativement le système. Premièrement, enrichir le dataset personnel à au moins 100 images par classe, réduirait l'écart de performance entre MNIST et les données réelles. Deuxièmement, implémenter le CNN en C constituerait une avancée technique notable : bien que plus complexe, les fonctions de convolution et de pooling sont parfaitement réalisables en C pur et permettraient d'exploiter la meilleure précision du CNN tout en conservant des temps d'inférence compatibles avec le temps réel (objectif < 40 ms selon le cahier des charges). Enfin, l'intégration d'une quantification des poids (passage de float32 à int8) réduirait la mémoire utilisée d'un facteur 4 et accélérerait encore les calculs sur l'architecture ARM du Raspberry Pi 5.

## 14 Guide de déploiement sur Raspberry Pi

Le déploiement de l'application sur Raspberry Pi 5 repose sur une architecture Docker qui encapsule l'environnement d'exécution OpenCV, garantissant la reproductibilité et évitant les conflits de dépendances avec le système local.

### Prérequis :

La Raspberry Pi 5 doit disposer de Docker installé ainsi que de rpicam-vid, l'outil de capture vidéo natif fourni avec Raspberry Pi OS. La caméra Module V3 doit être connectée via le port CSI et détectable avec la commande <rpicam-hello --list-cameras>.

### Transfert des fichiers :

L'ensemble du projet est transféré sur la Pi via SCP depuis la machine locale :

```
scp -r opencv_docker_cpp_tcp/ rpi@<IP>:~/  
scp -r mlp_weights_txt/ rpi@<IP>:~/opencv_docker_cpp_tcp/
```

### Construction et lancement :

Le script run.sh fourni par l'enseignant automatise l'ensemble des étapes. La commande ./run.sh all effectue dans l'ordre : la construction de l'image Docker (compilation du main.cpp via le Makefile interne), le démarrage de rpicam-vid qui capture le flux H264 depuis la caméra et l'expose sur le port TCP 5000, puis le lancement du conteneur Docker qui consomme ce flux, applique le traitement OpenCV et le MLP, et réemet le flux traité sur le port 8554.

Le dossier « mlp\_weights\_txt/ » est monté comme volume dans le conteneur Docker pour être accessible au programme C sans recompilation :

```
A modifier : docker run -d --rm --name cam --network host \-v  
$(pwd)/mlp_weights_txt:/app/mlp_weights_txt cam
```

### Visualisation distante :

Depuis un PC, sur le même réseau, le flux vidéo traité est visualisable via VLC :

```
vlc "tcp://@IP:8554"
```

La fluidité de la visualisation distante dépend de la bande passante réseau et peut ne pas être parfaite. En revanche, le traitement local sur la Pi s'effectue à 30 FPS constants, ce qui est largement suffisant pour la reconnaissance en temps réel.