BORDEAUX INP Enseirb-Matmeca

Développement d'une IA légère sur un système embarqué

Implémentation d'un modèle de reconnaissance de chiffres sur Raspberry Pi 3

Réalisé par :
JAOUHARI Mouad
CHMITI Taha

Encadré par :
M. CHATRIE Frédéric

Module : EN345
Filière : TSI

Table des matières

1 Introduction

Dans le cadre de ce projet, deux architectures de réseaux de neurones ont été mises en œuvre pour la reconnaissance de chiffres manuscrits : un Perceptron Multicouche (MLP) et un Réseau de Neurones Convolutif (CNN). Ces modèles ont été entraînés sur une base de données spécifique, constituée d'images BMP, présentant un fond noir et des chiffres en écriture blanche de taille approximative 68 px. Les images ont été soigneusement prétraitées afin d'extraire et de normaliser les informations pertinentes pour la tâche de classification.

L'utilisation du framework **PyTorch** [2] a été déterminante dans ce projet. En effet, PyTorch offre une grande flexibilité pour le prototypage rapide et l'expérimentation grâce à :

— Une interface intuitive pour la construction et l'entraînement de modèles de réseaux de neurones.
— Des capacités de calcul différentiel automatique, facilitant la mise en œuvre et l'optimisation des algorithmes d'apprentissage.
— Une compatibilité native avec le calcul sur GPU, accélérant significativement l'entraînement, en particulier pour des architectures complexes comme les CNN.
— Une intégration aisée avec d'autres outils et langages, notamment pour l'exportation des poids qui seront ensuite utilisés dans un environnement C pour l'inférence.

Ces avantages justifient pleinement le choix de PyTorch pour la phase d'entraînement des modèles, garantissant ainsi une robustesse et une efficacité dans le traitement des données.

2 Prétraitement de la Base de Données d'Entraînement

Les images initiales, de dimensions 1000×1000, contiennent un chiffre manuscrit en blanc sur un fond noir. Afin d'optimiser l'entraînement des modèles, un prétraitement spécifique a été appliqué par M. CHATRIE, consistant en plusieurs étapes :

1. **Cadrage et Centrage :** La détection des contours du chiffre permet d'éliminer les bordures noires superflues et de recentrer l'image sur la zone d'intérêt. Cette étape assure que le chiffre occupe la majeure partie de l'image.
2. **Seuillage :** Un seuillage est appliqué pour convertir l'image en une version binaire, éliminant ainsi les niveaux de gris intermédiaires. Cette opération garantit que seules deux valeurs sont présentes (noir et blanc), facilitant la distinction entre le fond et le chiffre.
3. **Redimensionnement :** Après cadrage et seuillage, l'image est redimensionnée à une taille standard de 28×28 pixels. Ce format réduit permet non seulement de diminuer la charge computationnelle lors de l'entraînement, mais il correspond également à des standards couramment utilisés pour la reconnaissance de chiffres (cf. MNIST [1]).

3 Prétraitement des images d'entraînement

Les images brutes (1000x1000 pixels) sont traitées pour :

— **Centrage et cadrage** : Suppression des bordures noires via la détection de la boîte englobante.
— **Binarisation** : Seuillage à 20 pour éliminer les niveaux de gris (blanc/noir strict).
— **Redimensionnement** : Ajustement en 28x28 pixels par interpolation voisine.
— **Retournement vertical** : Correction de l'orientation pour correspondre au format MNIST.

Listing 1 – Extrait du prétraitement en C (digits_id.c)

```c
int preprocess(const char *filename, uint8_t out[IMAGE_SIZE *
    IMAGE_SIZE]) {
    FILE *input_file = fopen(filename, "rb");
    if (!input_file) {
        perror("Error opening BMP file");
        return -1;
    }

    BMPFileHeader bmp_file_header;
    BMPInfoHeader bmp_info_header;
    fread(&bmp_file_header, sizeof(BMPFileHeader), 1, input_file);
    fread(&bmp_info_header, sizeof(BMPInfoHeader), 1, input_file);

    // Check BMP format
    if (bmp_file_header.bfType != 0x4D42 || bmp_info_header.biBitCount
        != 24) {
        fprintf(stderr, "Error: Unsupported BMP format (must be 24-bit)
            \n");
        fclose(input_file);
        return -2;
    }

    int width  = (int)bmp_info_header.biWidth;
    int height = (bmp_info_header.biHeight > 0) ? bmp_info_header.
        biHeight : -bmp_info_header.biHeight;
    int row_padded = (width * 3 + 3) & ~3;

    // Read raw pixel data
    uint8_t *bmp_data = (uint8_t*)malloc(height * row_padded);
    if (!bmp_data) {
        fclose(input_file);
        return -3;
    }
    fseek(input_file, bmp_file_header.bfOffBits, SEEK_SET);
    fread(bmp_data, 1, height * row_padded, input_file);
    fclose(input_file);

    // Convert to grayscale + threshold
    uint8_t *grayscale = (uint8_t*)malloc(width * height);
    if (!grayscale) {
        free(bmp_data);
        return -4;
    }
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int idx = y * row_padded + x * 3;
            uint8_t b = bmp_data[idx + 0];
            uint8_t g = bmp_data[idx + 1];
            uint8_t r = bmp_data[idx + 2];
            uint8_t gray = (uint8_t)(0.3f*r + 0.59f*g + 0.11f*b);
            if (gray > THRESHOLD) grayscale[y*width + x] = 255;
            else                  grayscale[y*width + x] = 0;
```

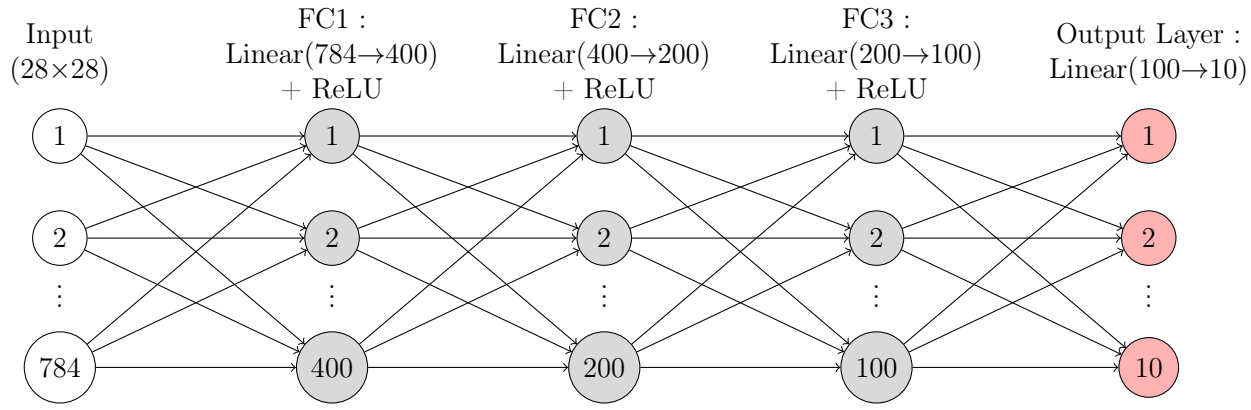
```
        }
    }
    free(bmp_data);

    // Bounding box of white pixels
    int x_min = width, x_max = 0;
    int y_min = height, y_max = 0;
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            if (grayscale[y*width + x] == 255) {
                if (x < x_min) x_min = x;
                if (y < y_min) y_min = y;
                if (x > x_max) x_max = x;
                if (y > y_max) y_max = y;
            }
        }
    }

    // If no white pixels found
    if (x_max < x_min || y_max < y_min) {
        memset(out, 0, IMAGE_SIZE*IMAGE_SIZE);
        free(grayscale);
        return 0;
    }

    // Crop
    int cropped_width  = x_max - x_min + 1;
    int cropped_height = y_max - y_min + 1;
    uint8_t *cropped = (uint8_t*)malloc(cropped_width*cropped_height);
    if (!cropped) {
        free(grayscale);
        return -5;
    }
    for (int yy = 0; yy < cropped_height; yy++) {
        for (int xx = 0; xx < cropped_width; xx++) {
            cropped[yy*cropped_width + xx] =
                grayscale[(y_min+yy)*width + (x_min+xx)];
        }
    }
    free(grayscale);

    // Resize to 28x28 (nearest neighbor)
    uint8_t *resized = (uint8_t*)malloc(IMAGE_SIZE*IMAGE_SIZE);
    if (!resized) {
        free(cropped);
        return -6;
    }
    for (int yy = 0; yy < IMAGE_SIZE; yy++) {
        for (int xx = 0; xx < IMAGE_SIZE; xx++) {
            int src_x = xx * cropped_width  / IMAGE_SIZE;
            int src_y = yy * cropped_height / IMAGE_SIZE;
            resized[yy*IMAGE_SIZE + xx] = cropped[src_y*cropped_width +
                src_x];
```

```
        }
    }
    free(cropped);

    memcpy(out, resized, IMAGE_SIZE*IMAGE_SIZE);
    free(resized);

    // Flip vertically if desired
    vertical_flip(out, IMAGE_SIZE, IMAGE_SIZE);

    // Optional save for debugging
    save_bmp("output.bmp", out, IMAGE_SIZE, IMAGE_SIZE);

    return 0;
}
```

4 Architectures des modèles

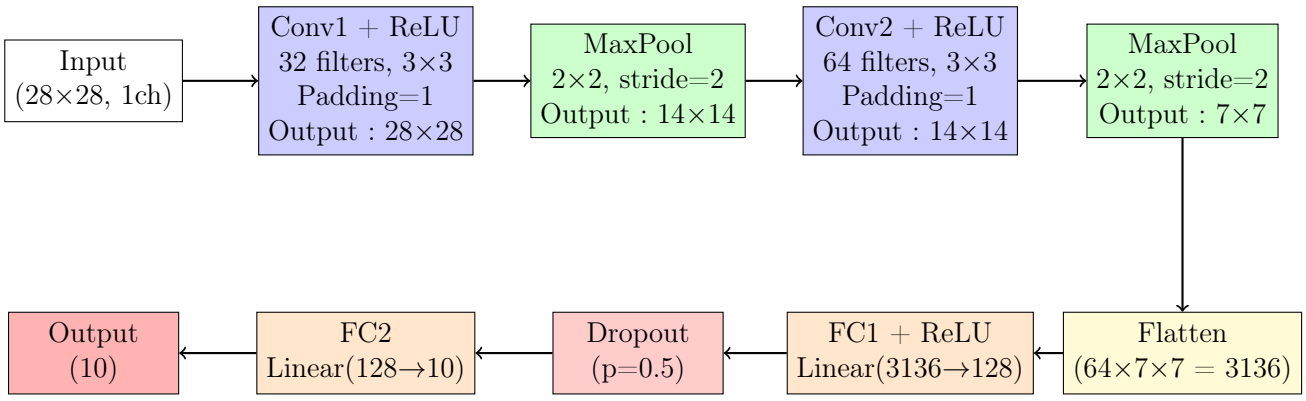
4.1 MLP (Perceptron Multicouche)

- Structure : 784 → 400 → 200 → 100 → 10.
- Fonctions d'activation : ReLU pour les couches cachées.
- Entraînement : 20 époques avec Adam (lr=0.001) et Cross-Entropy Loss.



4.2 CNN (Réseau de Neurones Convolutif)

- Couches :
  - Conv2D (1 → 32, kernel 3x3, padding 1) + MaxPool(2x2).
  - Conv2D (32 → 64, kernel 3x3, padding 1) + MaxPool(2x2).
  - Fully Connected (64×7×7 → 128 → 10).
- Dropout (50%) pour éviter le surajustement.



4.3 Export des Poids

Les poids des modèles entraînés sont exportés dans des fichiers `.h` via des scripts Python :

— Formatage des tensors PyTorch en tableaux C.

— Structuration des couches avec des métadonnées (tailles, activations).

Listing 2 – Extrait de l'export CNN (train_CNN.py)

```
# ----------------------------------------------------
# Export Weights => cnn_weights.h
# ----------------------------------------------------
model.load_state_dict(torch.load("cnn_digits_model.pth", weights_only=
    True))
model.eval()

conv_layers = [model.conv1, model.conv2]
fc_layers = [model.fc1, model.fc2]

with open("cnn_weights.h", "w") as f:
    f.write("// cnn_weights.h : Exported weights for CNN\n")
    f.write("// Generated by cnn.py\n\n")
    .
    .
    .
    f.write("    const float *bias;\n")
    f.write("} FCLayerDef;\n\n")

    # Export Conv Layers
    for idx, conv in enumerate(conv_layers):
        W = conv.weight.detach().cpu().numpy()
        b = conv.bias.detach().cpu().numpy()
        .
        .
        .
        f.write(f"static const float {bias_name}[] = {{ ")
        f.write(", ".join(str(x) for x in b))
        f.write("};\n\n")

    f.write("static ConvLayerDef CNN_CONV_LAYERS[CNN_NUM_CONV_LAYERS] =
        {\n")
    for idx, conv in enumerate(conv_layers):
        W = conv.weight.detach().cpu().numpy()
```

```
        out_c, in_c, kH, kW = W.shape
        padding = conv.padding[0]
        weight_name = f"CONV{idx}_WEIGHT"
        bias_name = f"CONV{idx}_BIAS"
        f.write(f"    {{ {in_c}, {out_c}, {kH}, {kW}, {padding}, {
            weight_name}, {bias_name} }},\n")
    f.write("};\n\n")

    # Export FC Layers
    for idx, fc in enumerate(fc_layers):
        W = fc.weight.detach().cpu().numpy()
        b = fc.bias.detach().cpu().numpy()
        .
        .
        .
        f.write(", ".join(str(x) for x in b))
        f.write("};\n\n")

    f.write("static FCLayerDef CNN_FC_LAYERS[CNN_NUM_FC_LAYERS] = {\n")
    for idx, fc in enumerate(fc_layers):
        W = fc.weight.detach().cpu().numpy()
        out_f, in_f = W.shape
       .
       .
        f.write(f"    {{ {in_f}, {out_f}, {act_type}, {weight_name}, {
            bias_name} }},\n")
    f.write("};\n\n")

print("CNN export completed. File generated: cnn_weights.h")
```

4.4 Inférence en C

Pour la phase d'inférence, un programme en langage C (digits_id.c) a été développé. Ce programme implémente un prétraitement similaire à celui appliqué aux images d'entraînement afin d'assurer une cohérence entre les données utilisées lors de l'apprentissage et celles soumises à la prédiction. Les principales étapes sont les suivantes :

1. **Encadrement :** Le programme détecte et supprime les bordures noires pour isoler le chiffre.
2. **Redimensionnement :** L'image est ensuite redimensionnée en une matrice de 28×28 pixels, assurant ainsi une uniformité avec les données d'entraînement.
3. **Seuillage :** Un seuillage est à nouveau appliqué pour maintenir la binarisation de l'image et garantir que seules deux valeurs (noir et blanc) sont présentes.

Ce processus de prétraitement, tant en phase d'entraînement qu'en phase d'inférence, est essentiel pour assurer que les modèles puissent généraliser efficacement et produire des prédictions précises, même lorsque les images de test présentent des variations en termes de taille et de style d'écriture.

Ce code présente une implémentation unique de l'inférence pour deux architectures de modèles (MLP et CNN). Le choix de l'architecture se fait au moment de la compilation grâce à l'utilisation de macros de préprocesseur. Ainsi, en compilant avec l'option `-D USE_MLP` ou `-D USE_CNN`, le code compilera respectivement la partie dédiée au MLP ou au CNN.

4.4.1 Structures BMP et Fonctions Utilitaires

La première partie du code définit les structures `BMPFileHeader` et `BMPInfoHeader` permettant de lire les en-têtes des fichiers BMP 24-bit. Ces structures servent à récupérer des informations essentielles (dimensions, décalage vers les données, etc.). On retrouve également plusieurs fonctions d'aide :

- `save_bmp` : Sauvegarde une image (par exemple, l'image prétraitée) au format BMP pour vérification.
- `vertical_flip` : Effectue un retournement vertical de l'image.
- `preprocess` : Cette fonction effectue le prétraitement de l'image en ouvrant le fichier BMP, en le convertissant en niveaux de gris, en appliquant un seuillage binaire, en détectant et en recadrant la zone contenant le chiffre, puis en redimensionnant l'image en une matrice 28×28. Ce prétraitement assure la cohérence entre les données d'entraînement et celles utilisées lors de l'inférence.

4.4.2 Implémentation du MLP

Si l'option `-D USE_MLP` est utilisée lors de la compilation, le code compile la section dédiée au MLP. Celle-ci utilise le fichier `mlp_weights.h` (généré par un script Python) qui exporte les poids et les biais sous forme de tableaux statiques ainsi que les structures de définition de couches (`FCLayerDef`). Les fonctions clés sont :

- `fully_connected_mlp` : Réalise le passage linéaire d'une couche entièrement connectée.
- `relu_inplace_mlp` : Applique l'activation ReLU sur les sorties.
- `mlp_forward_pass` : Effectue le passage avant complet pour le MLP. Elle convertit l'image prétraitée en un vecteur normalisé (valeurs dans $[-1, 1]$), passe ce vecteur à travers les différentes couches linéaires (en appliquant ReLU aux couches cachées) et retourne la classe prédite par l'opération d'argmax sur la dernière couche.

4.4.3 Implémentation du CNN

Si l'option `-D USE_CNN` est définie, c'est la section dédiée au CNN qui sera compilée. Le fichier `cnn_weights.h` (également généré par un script Python) fournit les poids et biais des couches de convolution et des couches entièrement connectées, ainsi que leur description via des structures telles que `ConvLayerDef` et `FCLayerDef`. Les fonctions principales sont :

- `conv2d_cnn` : Effectue une convolution 2D sur l'image d'entrée en appliquant les filtres de la couche de convolution.
- `relu_inplace_cnn` : Applique l'activation ReLU sur les sorties de la convolution.
- `maxpool2d_cnn` : Réalise un sous-échantillonnage (max pooling) sur les sorties de la convolution.
- `fully_connected_cnn` : Exécute la transformation linéaire d'une couche entièrement connectée.
- `cnn_forward_pass` : Ordonne les opérations du CNN de la même manière qu'en Python : normalisation de l'image d'entrée, deux couches de convolution suivies de pooling, aplatissement (*flatten*), passage dans les couches entièrement connectées et sélection par argmax de la classe prédite.

4.4.4 Fonction `main`

La fonction `main` réalise les étapes suivantes :

1. Elle vérifie qu'un chemin vers un fichier BMP est fourni en argument.
2. Elle appelle la fonction `preprocess` pour obtenir une image 28×28 en niveaux de gris.

3. Selon la macro définie lors de la compilation (`USE_MLP` ou `USE_CNN`), elle appelle respectivement `mlp_forward_pass` ou `cnn_forward_pass`.
4. Enfin, le programme affiche la classe (chiffre) prédite.

4.4.5 Compilation et Exécution

Le programme peut être compilé de la manière suivante :

Listing 3 – Compilation

```
-- Compilation avec MLP
gcc digits_id.c -o digits_id -D USE_MLP

-- Compilation avec CNN
gcc digits_id.c -o digits_id -D USE_CNN
```

Puis, l'exécution se fait par :

Listing 4 – Exécution

```
./digits_id chemin_vers_image.bmp
```

Ce code modulaire et conditionnel permet ainsi de disposer d'une seule base de code pour effectuer l'inférence avec l'architecture de notre choix (MLP ou CNN), garantissant ainsi une cohérence entre le prétraitement des images et le format attendu par les modèles entraînés.

5 Résultats

5.1 Entraînement des Modèles

— **MLP** : Précision de 95% sur le jeu de test.

```
PS C:\Projet\Work> python3 train_MLP.py
Training MLP...
Epoch 1, Average Loss: 1.6008
Epoch 2, Average Loss: 0.3148
Epoch 3, Average Loss: 0.0434
Epoch 4, Average Loss: 0.0187
Epoch 5, Average Loss: 0.0116
Epoch 6, Average Loss: 0.0041
Epoch 7, Average Loss: 0.0032
Epoch 8, Average Loss: 0.0012
Epoch 9, Average Loss: 0.0008
Epoch 10, Average Loss: 0.0007
Epoch 11, Average Loss: 0.0006
Epoch 12, Average Loss: 0.0005
Epoch 13, Average Loss: 0.0004
Epoch 14, Average Loss: 0.0004
Epoch 15, Average Loss: 0.0003
Epoch 16, Average Loss: 0.0003
Epoch 17, Average Loss: 0.0003
Epoch 18, Average Loss: 0.0002
Epoch 19, Average Loss: 0.0002
```

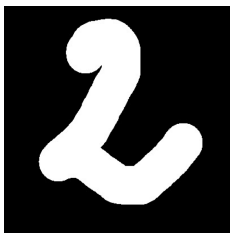
```
Epoch 20, Average Loss: 0.0002

MLP Test Accuracy: 95.00%
MLP export completed. File generated: mlp_weights.h
```

— **CNN** : Précision de 100% grâce aux caractéristiques spatiales.

```
PS C:\Projet\Work> python3 train_CNN.py
Training CNN...
Epoch 1, Average Loss: 2.0891
Epoch 2, Average Loss: 0.9151
Epoch 3, Average Loss: 0.4943
Epoch 4, Average Loss: 0.2716
Epoch 5, Average Loss: 0.1804
Epoch 6, Average Loss: 0.1896
Epoch 7, Average Loss: 0.1177
Epoch 8, Average Loss: 0.0704
Epoch 9, Average Loss: 0.0935
Epoch 10, Average Loss: 0.0581
Epoch 11, Average Loss: 0.0580
Epoch 12, Average Loss: 0.0911
Epoch 13, Average Loss: 0.0449
Epoch 14, Average Loss: 0.0229
Epoch 15, Average Loss: 0.0188
Epoch 16, Average Loss: 0.0285
Epoch 17, Average Loss: 0.0147
Epoch 18, Average Loss: 0.0214
Epoch 19, Average Loss: 0.0233
Epoch 20, Average Loss: 0.0251

CNN Test Accuracy: 100.00%
CNN export completed. File generated: cnn_weights.h
```

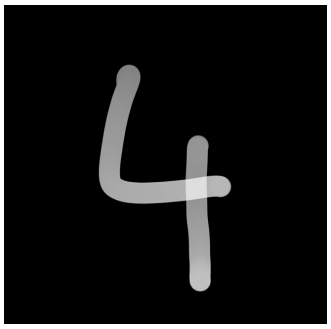
5.2 Validation du Prétraitement

Des tests avec des images variées (tailles, niveaux de gris) confirment la robustesse :
— Centrage correct même pour des chiffres excentrés.
— Binarisation efficace malgré des artefacts mineurs.



(a) Avant traitement

(b) Après traitement



(c) Avant traitement

(d) Après traitement

(e) Avant traitement

(f) Après traitement

FIGURE 1 – Exemple d'images de chiffres avant et après prétraitement implémenté dans le code C d'inférence

5.3 Test de l'inférence avec le MLP

Les tests d'inférence pour le modèle MLP ont été effectués en compilant le programme avec la directive `-D USE_MLP` et en exécutant le binaire sur plusieurs images d'inférence. Ci-dessous, quelques exemples d'exécution dans le terminal :

```
mj@MJ:/mnt/c/Projet/Work$ gcc digits_id.c -D USE_MLP
mj@MJ:/mnt/c/Projet/Work$ ./a.out ./Inference_digits/2_4.bmp
Processed image saved as BMP to output.bmp
Running MLP inference...
Predicted digit: 2
mj@MJ:/mnt/c/Projet/Work$ ./a.out ./Inference_digits/7_2.bmp
Processed image saved as BMP to output.bmp
Running MLP inference...
Predicted digit: 7
mj@MJ:/mnt/c/Projet/Work$ ./a.out ./Inference_digits/8_2.bmp
Processed image saved as BMP to output.bmp
Running MLP inference...
Predicted digit: 6
mj@MJ:/mnt/c/Projet/Work$
```

FIGURE 2 – La sortie du programme python de test

Ces résultats montrent que l'inférence avec le MLP s'exécute correctement sur la majorité des images testées. Cependant, le cas de l'image `8_2.bmp` où le chiffre prédit est **6** suggère que le modèle rencontre certaines difficultés sur des exemples particuliers, ce qui pourrait nécessiter une analyse approfondie de la qualité du prétraitement ou des caractéristiques du modèle pour améliorer la robustesse de la prédiction.

5.4 Test de l'inférence avec le CNN

De même, pour évaluer la performance du modèle CNN, le programme a été compilé en utilisant la directive `-D USE_CNN`. Les tests ont été réalisés sur plusieurs images d'inférence. Ci-dessous, quelques exemples d'exécution dans le terminal :

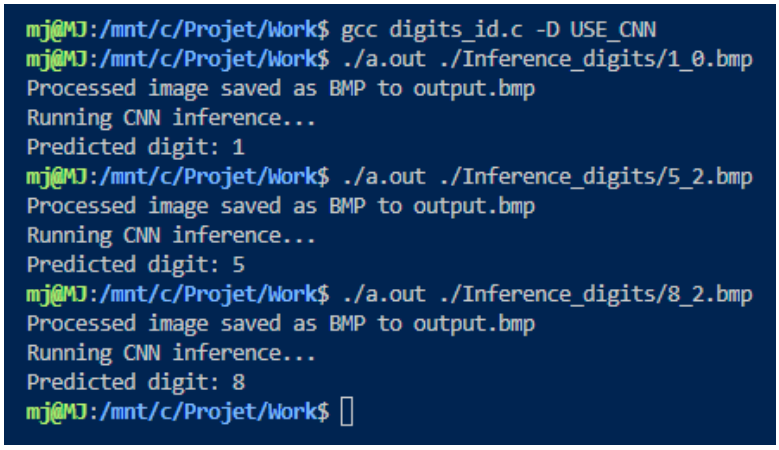


FIGURE 3 – La sortie du programme python de test

Les résultats obtenus montrent que le modèle CNN offre une meilleure robustesse pour la reconnaissance des chiffres par rapport au modèle MLP. En effet, bien que le MLP ait mal prédit le chiffre dans l'image `8_2.bmp`, le CNN a correctement identifié le chiffre 8. Ce constat souligne l'intérêt d'utiliser des architectures convolutionnelles pour des tâches de reconnaissance d'images, en particulier lorsque le prétraitement et la variabilité des images test représentent des défis pour les modèles entièrement connectés.

6 Exécution du projet

6.1 Execution avec fichier .sh

Pour lancer l'exécution du projet, un script shell (`run_project.sh`) a été développé. Ce script permet de choisir l'architecture (MLP ou CNN), de spécifier le chemin de l'image à traiter et de sélectionner le mode d'exécution (entraînement ou utilisation des poids déjà exportés).

Voici comment utiliser le script :

Listing 5 – Script `run_project.sh`

```
#!/bin/bash
# Usage:
#   ./run_project.sh <architecture> <image_path> <mode>
#
#   <architecture> : MLP ou CNN
#   <image_path>   : chemin vers l'image a tester (par exemple images/
    exemple.bmp)
```

```
#    <mode>          : "train" pour entrainer le modele et compiler,
#                      "notrain" pour utiliser les poids deja exportes
#
# Exemple d'utilisation :
#    ./run_project.sh CNN images/exemple.bmp train
#    ./run_project.sh MLP images/exemple.bmp notrain

if [ "$#" -ne 3 ]; then
    echo "Usage: $0 <architecture (MLP|CNN)> <image_path> <mode (train|notrain)>"
    exit 1
fi

ARCH=$1
IMAGE_PATH=$2
MODE=$3

# Definir les variables en fonction de l'architecture choisie
if [ "$ARCH" = "MLP" ]; then
    TRAIN_SCRIPT="train_MLP.py"
    WEIGHTS_FILE="mlp_weights.h"
    COMPILE_FLAG="-D USE_MLP"
elif [ "$ARCH" = "CNN" ]; then
    TRAIN_SCRIPT="train_CNN.py"
    WEIGHTS_FILE="cnn_weights.h"
    COMPILE_FLAG="-D USE_CNN"
else
    echo "Architecture inconnue : $ARCH. Utilisez MLP ou CNN."
    exit 1
fi

# Verification du mode
if [ "$MODE" = "train" ]; then
    echo "=== Entrainement du modele $ARCH ==="
    python3 "$TRAIN_SCRIPT"
    if [ $? -ne 0 ]; then
        echo "Erreur lors de l'entrainement du modele $ARCH."
        exit 1
    fi
    echo "Entrainement termine et export des poids realise dans $WEIGHTS_FILE."
elif [ "$MODE" = "notrain" ]; then
    echo "=== Mode non entrainement : Verification de l'existence de $WEIGHTS_FILE ==="
    if [ ! -f "$WEIGHTS_FILE" ]; then
        echo "Erreur : Le fichier $WEIGHTS_FILE contenant les poids de l'architecture $ARCH n'existe pas."
        exit 1
    fi
    echo "Le fichier $WEIGHTS_FILE existe."
else
    echo "Mode inconnu : $MODE. Utilisez 'train' ou 'notrain'."
    exit 1
```

```
fi

# Compilation du code C d'inference
echo "=== Compilation du code C d'inference avec l'architecture $ARCH ===
   "
gcc digits_id.c -o digits_id $COMPILE_FLAG
if [ $? -ne 0 ]; then
    echo "Erreur lors de la compilation du code C."
    exit 1
fi
echo "Compilation terminee."

# Execution du programme d'inference sur l'image specifiee
echo "=== Execution de l'inference sur l'image $IMAGE_PATH ==="
./digits_id "$IMAGE_PATH"
```

Ce script se charge :

— D'identifier l'architecture choisie (MLP ou CNN) et de configurer les variables associées (nom du script d'entraînement, fichier des poids et flag de compilation).

— D'exécuter, si demandé, l'entraînement du modèle via le script Python correspondant.

— De compiler le code C d'inférence avec la bonne macro (`-D USE_MLP` ou `-D USE_CNN`).

— D'exécuter le programme d'inférence sur l'image spécifiée.

6.2 Test effectué sur la RaspberryPi 3 à distance

Pour valider la portabilité et la performance du système d'inférence sur du matériel embarqué, des tests ont également été réalisés sur une RaspberryPi 3, exécutés à distance via SSH. Les résultats montrent que le système parvient à effectuer l'inférence en temps réel, même sur une plateforme aux ressources limitées.

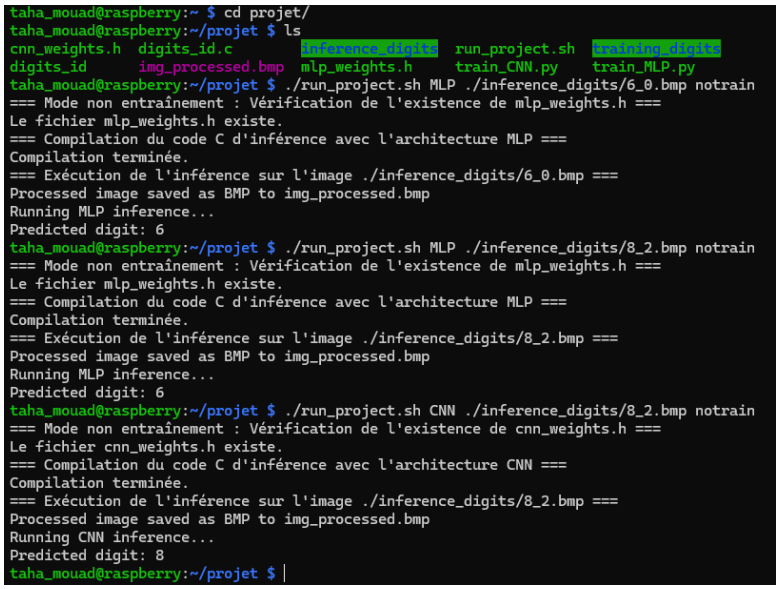


FIGURE 4 – Test de l'éxecution avec le fichier .sh sur la Raspberry Pi à distance

7 Conclusion

Ce projet a permis de démontrer l'efficacité de deux approches pour la reconnaissance de chiffres manuscrits : l'architecture MLP (Perceptron Multicouche) et l'architecture CNN (Réseau de Neurones Convolutif). L'utilisation de PyTorch pour l'entraînement a facilité la conception, l'expérimentation et l'optimisation des modèles, tandis que l'exportation des poids vers des fichiers `.h` a permis d'implémenter une phase d'inférence en langage C, adaptée à des environnements nécessitant des performances optimales et une intégration aisée dans des systèmes embarqués.

Le prétraitement rigoureux des images (cadrage, seuillage, redimensionnement en 28×28 pixels et retournement vertical) a assuré une homogénéité entre les données d'entraînement et d'inférence, garantissant ainsi une bonne robustesse des modèles. Les résultats expérimentaux montrent que, bien que le MLP atteigne une précision satisfaisante (environ 95%), le CNN offre une performance nettement supérieure en tirant parti des caractéristiques spatiales inhérentes aux images. Cette supériorité se traduit par une meilleure capacité de généralisation et une reconnaissance plus fiable des chiffres, même en présence de variations dans l'écriture ou le prétraitement.

L'approche modulaire adoptée dans le code C, qui permet de choisir l'architecture souhaitée au moment de la compilation grâce à des macros (`-D USE_MLP` ou `-D USE_CNN`), constitue un atout majeur pour l'adaptabilité et l'évolution future du système. Des améliorations ultérieures pourraient inclure l'intégration de techniques d'augmentation de données, le perfectionnement des méthodes de régularisation ainsi que l'optimisation de l'implémentation pour une utilisation en temps réel sur du matériel embarqué.

En définitive, ce travail souligne l'importance d'un prétraitement de qualité et du choix d'une architecture adaptée pour la réussite de la reconnaissance de chiffres manuscrits, tout en ouvrant la voie à des applications pratiques dans des contextes industriels et embarqués.

Références

[1] Yann LeCun and Corinna Cortes. The mnist database of handwritten digits. 2005.

[2] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch : An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703, 2019.