ENSEIRB-MATMECA
Bordeaux-INP

Électronique 3ème Année : TSI

IA embarquée

Étudiants:
Meriem LIATENI
Elisa SIUDA

Encadrant:
Frédéric Chatrie

Table des matières

I Introduction 2
II Environnement de travail 2
III Création et entrainement du réseaux 2
III.1 Les bases de données 2
III.2 MLP 3
III.3 CNN 7
IV Inférence 8
IV.1 MLP 8
IV.2 CNN 10
V Conclusion 11

I Introduction

Dans un contexte où l'intelligence artificielle (IA) embarquée prend une place croissante dans de nombreux domaines, notre projet vise à développer un modèle d'IA capable d'être exécuté sur une plateforme embarquée. Cependant, les contraintes matérielles des dispositifs embarqués, telles que la puissance de calcul limitée et la mémoire restreinte, nécessitent une optimisation rigoureuse des modèles d'IA. Pour relever ce défi, nous avons adopté une méthodologie en deux étapes : l'entraînement du modèle sur une machine puissante en utilisant Python, suivi de l'exportation des poids du modèle pour une inférence efficace en C sur une Raspberry Pi.

Pour garantir un développement efficace et reproductible, nous avons structuré notre environnement de travail à l'aide de Docker. Cette approche nous permet d'assurer la portabilité et la cohérence des dépendances tout au long du cycle de développement, depuis l'entraînement du modèle jusqu'à son déploiement sur le matériel cible.

Ce rapport détaille les différentes étapes du projet, depuis la conception du modèle jusqu'à son intégration sur la Raspberry Pi, en passant par la conversion des poids et l'optimisation du code d'inférence en C.

II Environnement de travail

Utilisation de Docker pour créer un environnement de travail.....

III Création et entrainement du réseaux

Lorsqu'il s'agit de traiter des images, un réseau Convolutional Neural Network (CNN) est généralement plus performant qu'un Multilayer Perceptron (MLP). En effet, il prend en compte la structure spatiale des images et il est plus robuste aux variations et translations. Par contre il a souvent une complexité plus élevé ainsi que plus de couches donc besoin de plus de mémoires ce qui n'est pas idéal pour de l'embarqué. C'est pourquoi nous étudierons ces deux réseaux.

III.1 Les bases de données

Nous ne voulons pas utiliser la base de données MNIST, Modified National Institue of Standards and Technology, qui est un ensemble de données de référence utilisé dans le domaine de l'apprentissage automatique mais plutôt une base similaire. La base de données MNIST contient des images de chiffres manuscrits, chaque image représentant un chiffre de 0 à 9. Chaque image est en niveaux de gris, de taille 28x28 pixels, et est stockée sous forme de tableau de nombres correspondant à l'intensité des pixels. MNIST comprend 60 000 images pour l'entraînement du modèle, accompagnées de leurs étiquettes. Il y a également 10 000 images pour tester le modèle après l'entraînement.

Nous allons créer notre propre base de données, inspirée de MNIST, mais beaucoup plus petite et simple. Chaque classe contiendra 10 images, et il y aura 10 classes au total. On numérotera les images à l'intérieur tel que 0_0.png.....9_9.png Pour les tests, nous utiliserons 5 images pour chacune des 10 classes.

```
/training
├── /0
│   ├── 0_0.png
│   ├── 0_1.png
│   ├── ...
│   └── 0_9.png
├── /1
│   ├── 1_0.png
│   ├── 1_1.png
│   ├── ...
│   └── 1_9.png
├── /2
│   ├── 2_0.png
│   ├── 2_1.png
│   ├── ...
│   └── 2_9.png
├── /...
└── /9
    ├── 9_0.png
    ├── 9_1.png
    ├── ...
    └── 9_9.png
```

FIGURE 1 – Courbe d'évolution du taux de précision sur MNIST (Entraînement)

```
/testing
├── /0
│   ├── 0_0.png
│   ├── 0_1.png
│   ├── 0_2.png
│   ├── 0_3.png
│   └── 0_4.png
├── /1
│   ├── 1_0.png
│   ├── 1_1.png
│   ├── 1_2.png
│   ├── 1_3.png
│   └── 1_4.png
├── /2
│   ├── 2_0.png
│   ├── 2_1.png
│   ├── 2_2.png
│   ├── 2_3.png
│   └── 2_4.png
├── /...
```

FIGURE 2 – Courbe d'évolution du taux de précision sur MNIST (Test)

Il y a 10 classes (de 0 à 9). Dans la partie 'training', chaque classe contient 10 images. Le nombre total d'images d'entraînement est donc 10 classes×10 images par classe=100 images d'entraînement. Pour la partie test, chaque classe contient 5 images, le nombre total d'images de test est de 50.

Tel que des données MNIST, les images ont une résolution de 28 x 28 pixels et sont ensuite converties au format BMP.

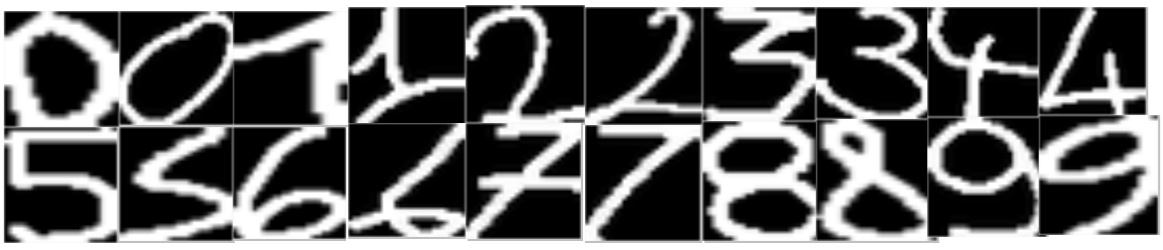

FIGURE 3 – Échantillon des données d'entraînement en format BMP

Nous allons utiliser cette nouvelle base dans le reste du projet.

III.2 MLP

Nous souhaitons tout d'abord implémenter une classification d'images type MNIST en utilisant un réseau de neurones multi-couches (MLP) avec Pytorch. Il y a tout d'abord une prepartion

des données, la définition du modèle, l'entrainement puis la validation.

Préparation des données

On utilise `torchvision.transforms` pour appliquer une transformation aux images, plus précisément `T.ToTensor()`, qui convertit les images en tenseurs avec des valeurs normalisées entre 0 et 1.

```
transform = T.Compose([T.ToTensor(),])
```

On a créé une classe `Dataset` personnalisée qui implémente trois méthodes : `__init__`, `__len__` et `__getitem__`. Il est utilisé pour charger les données d'entraînement et de validation depuis les chemins spécifiés

```
path_train = '/home/docker/Work/training_bmp'
training_set = Dataset(path_train, transform=transform)
```

Les données sont ensuite chargées dans un DataLoader, qui permet un traitement efficace en lots (batch_size=16) et active le mélange des échantillons (shuffle=True) pour améliorer la généralisation du modèle.

```
train_loader = DataLoader(dataset=training_set, batch_size=batch_size,
    shuffle=True, num_workers=2)
```

Définition du modèle

On utilise un réseau de neurones simple en utilisant la classe MLP héritant de torch.nn.Module. Il est composé de la façon suivante :

- une couche entièrement connectée `fc1` de taille D x H (784 x 30). Applique une transformation linéaire X1W1+b1 où W1 est une matrice de taille D×H (784 × 30). Cette transformation apprend des représentations utiles en réduisant la dimensionnalité et en capturant des motifs dans les images
- une activation `ReLu` pour introduire de la non-linéarité et faciliter la propagation du gradient et atténue le problème du gradient qui s'annule.
- une seconde couche entièrement connectée `fc2` de taille H x C (30 x 10). Seconde transformation linéaire X2W2+b2 où W2 est de taille H×C (30 × 10). Elle projette l'espace de représentation vers l'espace de classification des 10 chiffres.
- une activation `Softmax` pour normaliser les scores de sortie en probabilités.

```
class MLP(nn.Module):
    def __init__(self, H, input_size):
        super(MLP, self).__init__()
        self.C = 10
        self.D = input_size
        self.H = H
        self.fc1 = nn.Linear(self.D, self.H)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(self.H, self.C)
        self.softmax = nn.Softmax(dim=1)
```

L'entrée du MLP est une image de 28x28 pixels en niveaux de gris ce qui donne 784 (D). Chaque pixel est considéré comme une caractéristique d'entrée, ce qui explique pourquoi la première couche entièrement connectée a 784 neurones en entrée. On a H = 30 qui définit la taille de la couche cachée, c'est-à-dire le nombre de neurones de la couche intermédiaire. Ici on a choisis 30 neurones cachés. Un trop grand H pourrait entraîner un sur-ajustement, tandis qu'un H trop

faible limiterait la capacité du modèle à apprendre des représentations utiles. Ce choix a été fait empiriquement en fontion des perofrmances sur l'ensemble de validation. C est le nombre de classes en sortie, correspondant au nombre de catégories à prédire. Ici, on traite un problème de classification à 10 classes (avec les chiffres de 0 à 9).

Entraînement et validation

On réalise l'entrainement sur 20 épochs avec un taux d'apprentissage de 1e-2. L'optimiseur choisi est Adam.

```
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
```

et la fonction de coût utilisée est `CrossEntropyLoss`, adaptée au problèmes de lcassification multi-classes.

```
criterion = nn.CrossEntropyLoss()
```

À chaque époque, le modèle convertit les images en vecteurs 1D, passe les données dans le MLP, calcule la perte et effecute une rétropropagation. Enfin, les poids du réseau sont mis à jour avec `optimizer.step()`.
Après chaque epcoh, la précision du modèle est calculée sur l'ensemble de validation grâce à une fonction `validation()`

```
(correct, total) = validation(valid_loader, model)
valid_acc_v.append(correct / total)
```

Visualisation des résultats

Pour valider le modèle, nous avons d'abord effectué l'entraînement et les tests en utilisant la base de données MNIST. Nous avons ajusté le nombre d'épochs à 15 et le batch_size à 30 pour nous adapter à la taille importante des données. Après l'entraînement, nous avons obtenu un taux de précision sur l'ensemble de validation de 94,8 %, ce qui est un bon indicateur de la performance du modèle sur des données standardisées.

La première courbe présente l'évolution de la précision au fil des épochs. Cette courbe montre comment le modèle améliore sa capacité à classer correctement les images tout au long de l'entraînement.

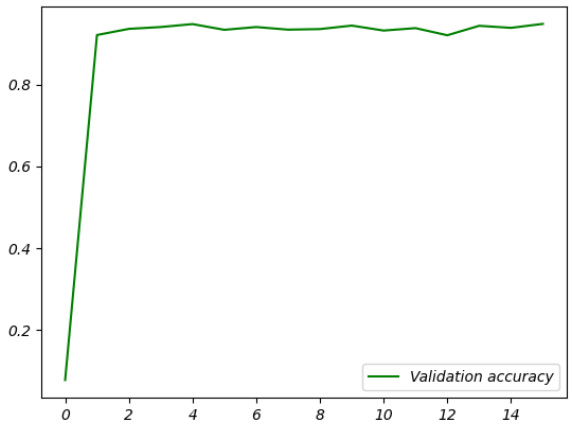

FIGURE 4 – Courbe d'évolution du taux de précision sur MNIST

La deuxième courbe illustre l'évolution de la perte d'entraînement (training loss). La perte diminue progressivement, indiquant que le modèle améliore sa capacité à prédire les bonnes classes et à s'ajuster aux données d'entraînement.

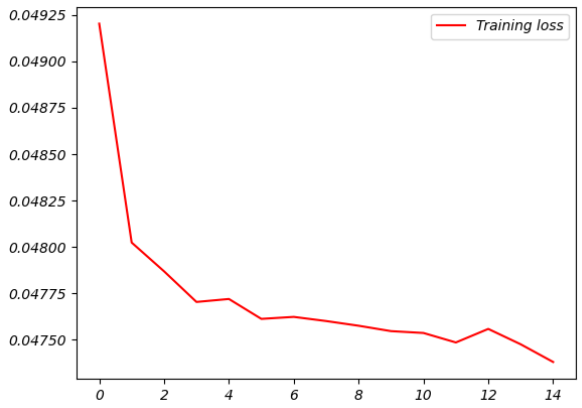

FIGURE 5 – Courbe d'évolution de la perte d'entraînement sur MNIST

Ensuite, nous avons testé le modèle sur notre propre base de données, qui est composée d'images de format 28x28 en niveaux de gris avec les caractéristiques spécifiées dans la partie de la définition du modèle. La courbe ci-dessous représente l'évolution du taux de précision sur cette base de données spécifique. Même si la performance est moins bonne qu'avec le MNIST, les résultats montrent un taux de précision à 80%.

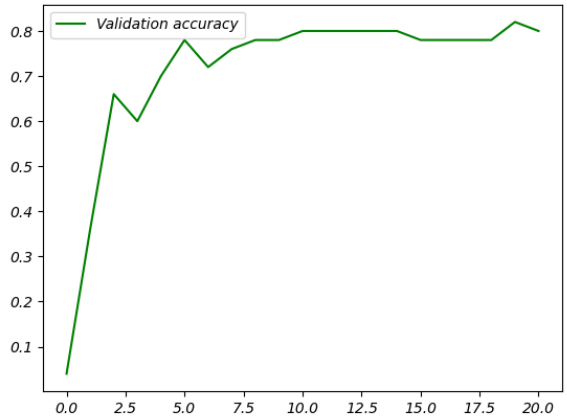

FIGURE 6 – Courbe d'évolution du taux de précision sur notre base de données personnalisée

Enfin, la courbe suivante illustre l'évolution de la perte d'entraînement sur notre propre base de données. Comme sur la base MNIST, la perte diminue avec l'entraînement, ce qui indique que le modèle s'ajuste bien aux caractéristiques de nos données.

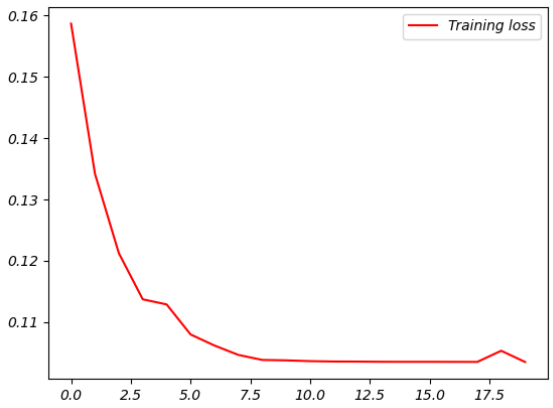

FIGURE 7 – Courbe d'évolution de la perte d'entraînement sur notre base de données personnalisée

En conclusion, bien que les résultats sur la base de données MNIST soient très efficaces, nous avons pu tester le modèle pour évaluer sa précision sur notre propre base de données, qui présente des caractéristiques différentes de celles de MNIST.

III.3 CNN

On préparera les données d'entrée de la même façon que pour le MLP avant de définir et d'entrainer le modèle.

Définition du modèle

Le modèle est structuré en plusieurs couches :

- Une premiére couche de convolution :
  Elle applique un filtre de convolution de 3x3 avec 32 canaux qui permet d'extraire des caractéristiques de bas niveau comme les contours et les textures. Puis une fonction `ReLU` est utilisée après la convolution pour introduire la non-linéarité et permettre au réseau d'apprendre des relations complexes. Enfin, Un `MaxPooling` 2x2 suit afin de réduire la dimension spatiale de moitié tout en conservant les informations les plus importantes.
- Une deuxième couche de convolution :
  Le filtre de convolution toujours 3x3 mais avec 64 canaux, affine l'extraction des caractéristiques et détecte des motifs plus complexes. Un `MaxPooling` 2x2 est de nouveau utilisé pour réduire la taille des cartes de caractéristiques, ce qui diminue le nombre de paramètres et améliore l'efficacité du modèle.
- Une couche `Fully connected(FC)` :
  Elle applique une transformation non linéaire pour apprendre des représentations plus abstraites des chiffres.
- Une couche de sortie, elle aussi `FC` :
  Elle applique une activation softmax (implicite dans CrossEntropyLoss) pour produire des probabilités de classification. On a donc une sortie de taille 10.

```
def __init__(self, num_classes):
        super(CNN, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
```

```
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, num_classes)
```

L'utilisation de seulement deux couches de convolution permet de limiter le nombre de paramètres, ce qui réduit le risque d'overfitting sur notre petite base de données. De plus, en gardant en tête l'objectif qui est d'exécuter ce modèle sur un système embarqué, l'architecture choisie est un bon compromis entre performance et légèreté.

Entraînement et Résultat

On réalise l'entrainement sur 18 epochs avec un batch de taille 15 et un taux d'apprentissage de 0,001. Tout comme pour le MLP, on utilise l'optimiseur Adam et la fonction de coût `CrossEntropyLoss`.

On obtient un taux de précision allant de 80% à 84% selon les entrainements. Le résultat est à peine supérieur à celui obtenu avec le MLP. On peut donc supposé que nos résultats sont limité par la taille de la base de données.

IV Inférence

IV.1 MLP

Ce programme en C permet d'effectuer la classification d'images de chiffres en utilisant un réseau de neurones à deux couches entièrement connectées. Il est conçu pour traiter des images en niveaux de gris de 28x28 pixels, comme celles de notre base de données. L'objectif est de prédire le chiffre représenté dans une image en appliquant une propagation avant à travers le réseau de neurones.

Tout d'abord, le programme charge une image fournie en argument au format BMP. Pour cela, il utilise la bibliothèque 'stb_image.h', qui permet de lire des images et d'extraire leurs valeurs de pixels. L'image est convertie en un vecteur de 784 valeurs (28×28), où chaque pixel est normalisé entre 0 et 1 en divisant son intensité par 255.

```
1  int main(int argc, char *argv[])
2  {
3      if (argc != 2) {
4          printf("Usage: %s <image_path>\n", argv[0]);
5          return -1;
6      }
7      char *image_path = argv[1];
8      int width, height, channels;
9      unsigned char *image = stbi_load(image_path, &width, &height, &channels, 1);
10     if (image == NULL)
11     {
12         printf("Error loading image: %s\n", image_path);
13         return -1;
14     }
15     float input[INPUT_SIZE];
16     for (int i = 0; i < 28 * 28; i++)
17     {
18         input[i] = image[i] / 255.0f;
```

```
    }
    stbi_image_free(image);
    float output[OUTPUT_SIZE];
    inference(input, output);
```

Le réseau de neurones utilisé est composé de :

- Une couche d'entrée contenant 784 neurones, correspondant aux pixels de l'image.
- Une couche cachée avec 30 neurones, activés à l'aide de la fonction ReLU, qui remplace les valeurs négatives par zéro.
- Une couche de sortie avec 10 neurones, chacun représentant une classe possible (les chiffres de 0 à 9).

Les poids et les biais du réseau sont chargés à partir de fichiers texte (‘fc1_weights.txt‘, ‘fc1_biases.txt‘, etc.). Ces fichiers contiennent les valeurs pré-entraînées qui permettent de réaliser les calculs nécessaires à la classification. La première étape de l'inférence consiste à multiplier les valeurs des pixels par les poids de la première couche et à ajouter les biais correspondants.

```
void mat_mult(float *input, float *weights, float *output, int input_size, int
    output_size, float *biases)
{
    for (int i = 0; i < output_size; i++)
    {
        output[i] = 0.0f;
        for (int j = 0; j < input_size; j++) {
            output[i] += input[j] * weights[i * input_size + j];
        }
        output[i] += biases[i];
    }
}
```

Ensuite, la fonction d'activation ReLU est appliquée pour introduire de la non-linéarité dans le modèle.

```
void relu(float *tensor, const unsigned int size)
{
    unsigned int i;
    for (i = 0; i < size; i++)
    {
        tensor[i] = MAX(tensor[i], 0);
    }
}
```

Une seconde multiplication matricielle est ensuite effectuée avec les poids et biais de la dernière couche pour obtenir la prédiction finale. Une fois ces calculs effectués, la sortie du réseau contient un vecteur de 10 valeurs, chacune représentant un score pour une classe.

Le programme identifie la classe prédite en trouvant la position de la valeur maximale dans ce vecteur. Il affiche alors le chiffre reconnu à l'écran.

```
int predicted_class = 0;
for (int i = 1; i < OUTPUT_SIZE; i++)
{
    if (output[i] > output[predicted_class]) {
        predicted_class = i;
    }
}
```

On réalise un modèle de classification simple mais efficace, basé sur un réseau de neurones entièrement connecté. Il traite une image en l'aplatissant sous forme de vecteur, applique une série

d'opérations mathématiques pour en extraire des caractéristiques, et prédit finalement le chiffre correspondant en utilisant les poids et biais pré-entraînés. Cette approche permet de reconnaître des chiffres manuscrits avec une bonne précision, en exploitant un modèle simple.

On crée un `main.c` qui automatise l'exécution du modèle de classification (`mlp_inference`) sur un ensemble d'images de test. Les images sont organisées dans des dossiers nommés de 0 à 9, correspondant aux classes des chiffres. Chaque dossier contient 5 images au format BMP, nommées selon la convention classe_index.bmp (par exemple 0_0.bmp, 0_1.bmp, etc.).

L'objectif du programme est de parcourir ces dossiers et exécuter l'inférence sur chaque image en appelant `mlp_inference`. L'image ci-dessous affiche les résultats de l'inférence d'un modèle de classification sur un ensemble d'images de test. Chaque ligne correspond à une image test et présente son chemin d'accès ainsi que la classe prédite par le modèle.

```
Image: /home/docker/Work/testing_bmp/0/0_0.bmp -> Classe prédite: 0
Image: /home/docker/Work/testing_bmp/0/0_1.bmp -> Classe prédite: 0
Image: /home/docker/Work/testing_bmp/0/0_2.bmp -> Classe prédite: 0
Image: /home/docker/Work/testing_bmp/0/0_3.bmp -> Classe prédite: 0
Image: /home/docker/Work/testing_bmp/0/0_4.bmp -> Classe prédite: 0
Image: /home/docker/Work/testing_bmp/1/1_0.bmp -> Classe prédite: 1
Image: /home/docker/Work/testing_bmp/1/1_1.bmp -> Classe prédite: 1
Image: /home/docker/Work/testing_bmp/1/1_2.bmp -> Classe prédite: 1
Image: /home/docker/Work/testing_bmp/1/1_3.bmp -> Classe prédite: 1
Image: /home/docker/Work/testing_bmp/1/1_4.bmp -> Classe prédite: 1
Image: /home/docker/Work/testing_bmp/2/2_0.bmp -> Classe prédite: 4
Image: /home/docker/Work/testing_bmp/2/2_1.bmp -> Classe prédite: 2
Image: /home/docker/Work/testing_bmp/2/2_2.bmp -> Classe prédite: 2
Image: /home/docker/Work/testing_bmp/2/2_3.bmp -> Classe prédite: 2
Image: /home/docker/Work/testing_bmp/2/2_4.bmp -> Classe prédite: 2
Image: /home/docker/Work/testing_bmp/3/3_0.bmp -> Classe prédite: 3
Image: /home/docker/Work/testing_bmp/3/3_1.bmp -> Classe prédite: 3
Image: /home/docker/Work/testing_bmp/3/3_2.bmp -> Classe prédite: 3
Image: /home/docker/Work/testing_bmp/3/3_3.bmp -> Classe prédite: 3
Image: /home/docker/Work/testing_bmp/3/3_4.bmp -> Classe prédite: 2
```

FIGURE 8 – Visualisation de quelque résultats pour l'inférence du MLP en C

Par exemple, l'image '/home/docker/Work/testing_bmp/0/0_0.bmp' appartient à la classe '0' et a bien été classée comme telle par le modèle. En analysant les résultats, on constate que le modèle prédit correctement les classes à 80% sur les 50 données, tel que sur le test en Python.

IV.2 CNN

On réalise les mêmes étapes que pour le MLP, c'est à dire le chargement des poids, le prétraitement des images et l'exécution des couches du réseau de neurones.

On commence par charger les poids et les biais qui ont été enregistrés au préalable dans un fichier texte. On définit une fonction `load_weights` qui ouvre le fichier contenant les poids, compte le nombre de valeurs pour allouer la mémoire correspondante, lit les valeurs et les stocke dans un tableau float.

Les images sont chargées en plusieurs étapes. D'abord, chaque image est lue depuis un fichier à l'aide de `stbi_load()`. Ensuite, elle est redimensionnée à 28×28 pixels avec `stbir_resize_uint8_srgb()`, pour s'adapter à l'entrée du réseau. Après cela, les valeurs des pixels sont normalisées entre 0 et 1 afin d'assurer une cohérence avec l'entraînement du modèle. Enfin, les images et leurs labels sont stockés dans des tableaux dynamiques pour être utilisées lors de l'inférence. Le programme

gère également le chargement en lot des images depuis un dossier, en attribuant une étiquette en fonction du sous-dossier d'origine.

Enfin, pour réaliser l'inférence, une boucle for parcourt toutes les images du datasetet applique le réseau de neurones couche par couche. Les différentes couches sont définies dans un fichier (`layers.c`) à part pour une meilleur lisibilité du code. Contrairement à ce qui a été fait en Python, la couche Softmax doit cette fois être déclaré explicitement. On oublie surtout pas pour finir de libérer la mémoire allouer pour les images et les poids.

Après avoir effectué l'inférence sur toutes les images du dataset, nous comparons les prédictions aux labels réels pour calculer la précision globale :

```
if (predicted_class == labels[i]) {
    correct_predictions++;
}
float accuracy = (float)correct_predictions / total_images * 100.0;
printf("Pr cision␣sur␣l'ensemble␣des␣images␣:␣%.2f%%\n", accuracy);
```

On obtient une précision de seulement 10% et on remarque que la classe 9 est presque tout le temps prédite. Comme les résultats de l'entrainement sont correcte avec l'inférence en python, on peut en déduire que le problème vient de la façon dont les poids sont chargés ou utilisés.

V Conclusion

Ce projet nous a offert une précieuse opportunité d'explorer les différentes étapes de l'embarquement d'une intelligence artificielle sur un système embarqué. L'utilisation de Docker nous a permis de mieux comprendre les avantages de la conteneurisation, notamment pour la gestion des dépendances et la reproductibilité de l'environnement de développement.

En travaillant sur l'adaptation d'un modèle d'intelligence artificielle à une petite base de données comme MNIST, nous avons pu approfondir les contraintes liées aux ressources limitées et ajuster notre approche en conséquence. Nous avons comparé différentes architectures, notamment un MLP et un CNN, et analysé leurs performances dans ce contexte spécifique.

Au-delà des aspects techniques, ce projet nous a permis de développer une réflexion critique sur les choix d'architecture, d'optimisation et d'environnement d'exécution, des compétences essentielles dans le domaine de l'IA embarquée. Alors que nous nous apprêtons à entrer dans le monde professionnel, ces connaissances et cette approche méthodique nous seront précieuses pour relever les défis que nous rencontrerons.