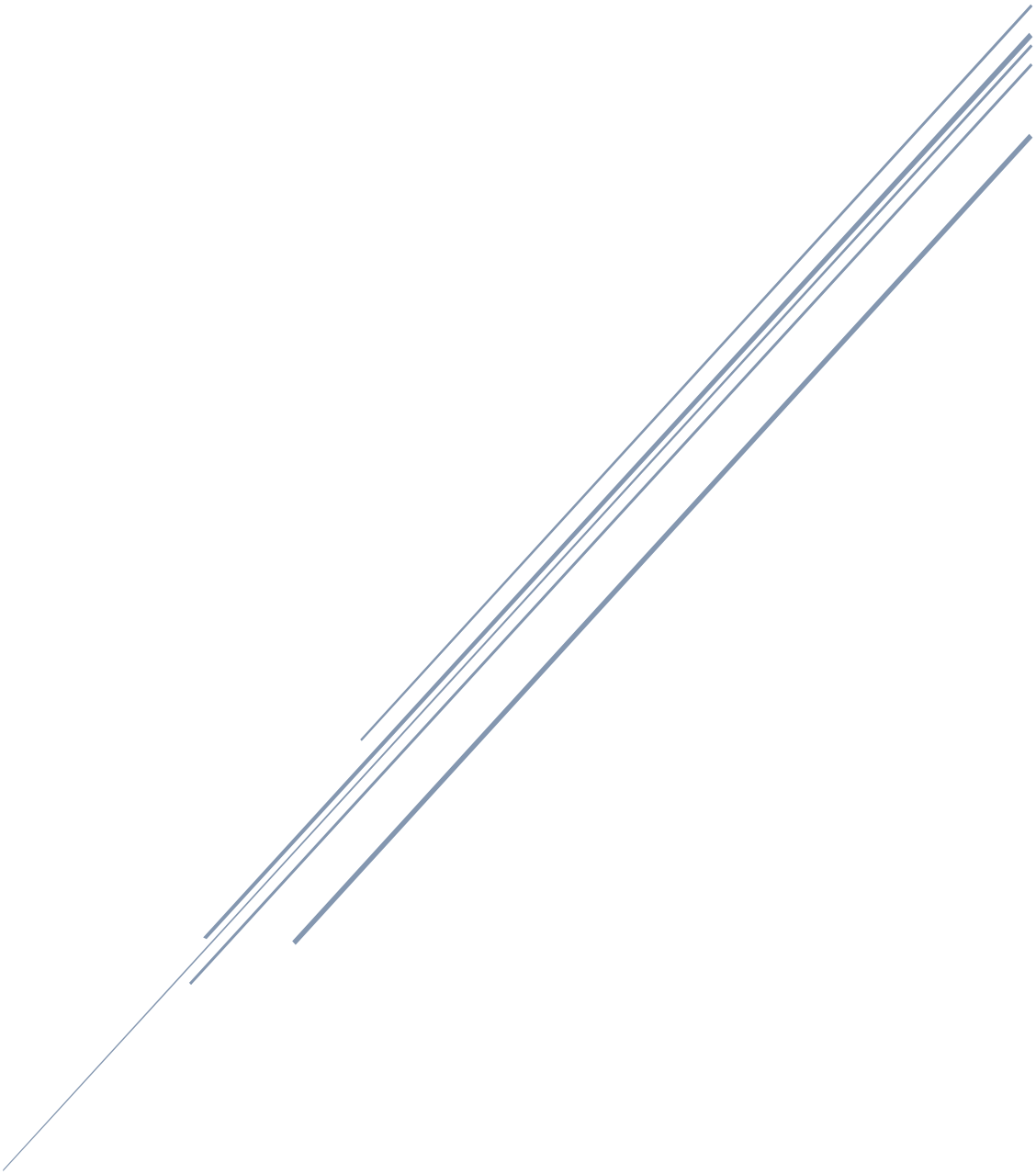
RAPPORT IA

Développement d’une intelligence artificielle déployé sur un système embarqué



Junia ISEN
Jordan Bazin

Table des matières

1 – Introduction

Le projet proposé consiste en la réalisation d'une application embarquée de reconnaissance de caractères. Ce rapport va donc présenter toutes les étapes de création du modèle de données, en passant par la création de la base de données, du choix du modèle utilisé ainsi que la détermination des hyperparamètres, jusqu'au développement de la phase d'inférence adapté à un système embarqué.

Pour cela, nous commencerons par une description du fonctionnement d'une intelligence artificielle par apprentissage profond (Deep Leanring). Ensuite, nous verrons comment la base de données a été mis en place et optimisée, puis, les étapes de développement du modèle de données ainsi que celle de la phase d'inférence. Nous conclurons sur chacune des approches via des tests.

2 - Intelligence artificielle par apprentissage profond

Le développement d'une intelligence artificielle peut se résumer en trois étapes :

- La création ou acquisition d'une base de données
- Entrainement du modèle
- Calcul de la sortie du modèle (phase d'inférence)

Il existe plusieurs familles d'IA, nous allons ici nous focaliser sur la « supervisé ». Cette méthode, utilisée notamment pour de la régression et de la classification, utilise un système d'apprentissage avec « professeur ». Les données d'entrées et de sorties, c'est-à-dire les différentes classes, sont fournies. Ainsi, pour chaque prédiction du modèle, on peut la comparer avec la sortie attendue.

L'intelligence artificielle par apprentissage profond, souvent appelé « Deep Learning » est une un procédé s'inspirant du fonctionnement du cerveau humain, en particulier du réseau de neurones. L'objectif est de reproduire leur fonctionnement via des réseaux de neurones artificielles et d'opérations mathématiques. Ces réseaux sont constitués de plusieurs couches dites « cachées », composées d'un certain nombre de neurones. Chaque neurone est associé à une valeur appelée « biais » et à une fonction d'activation. Les connexions entre les neurones sont associées à une autre valeur appelée « poids ». La force du deep learning réside dans ces poids et ces biais qui seront ajustés pendant le processus d'apprentissage pour minimiser l'erreur du modèle. La phase d'apprentissage en deep learning se déroule en 4 étapes :

- Forward propagation : Le modèle parcourt les différentes couches
- Calcul de la fonction de coût : À la suite de cette première étape, une prédiction est réalisée et une fonction de coût y est associée
- Back propagation : Le modèle parcourt les couches dans le sens inverse
- Descente de gradient : Pour chaque couche parcourue pendant la back propagation, une descente de gradient va être effectué avec la fonction de coût afin de trouver les valeurs des poids et des biais avec lesquelles l'erreur est la plus faible.

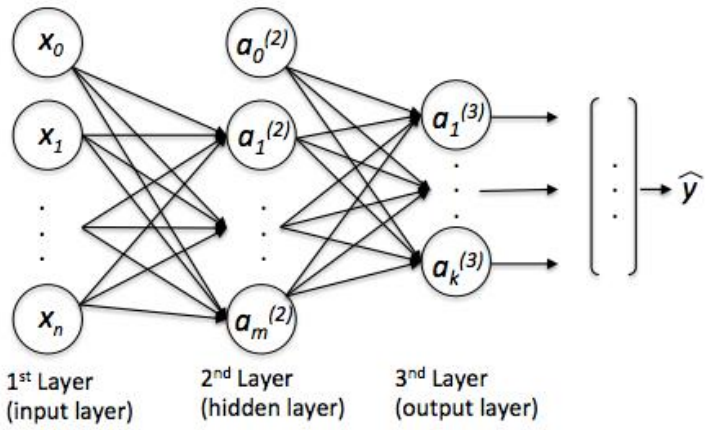


Figure 1 - schéma de fonctionnement d'un réseau de neurones

Lors de la phase d'inférence, ces poids et ces biais déterminés, seront utilisés pour calculer les sorties de chaque couche, jusqu'à la couche finale de sortie. Par exemple, dans le cas d'une IA utilisant l'architecture d'un perceptron multicouche, le calcul pour toutes les couches est le suivant :

$$y_{neural} = f\left(\left(\sum_{i \in \mathbb{N}} w_i * x_i\right) + b\right)$$

Figure 2- calcul d'inférence pour un perceptron multicouche

Ce calcul appliqué à la dernière couche, renvoie un vecteur de la taille du nombre de classe. Avec comme valeurs, les probabilités de correspondance de l'entrée avec chacune des classes. Les valeurs des poids et des biais sont donc cruciales, et dépendent entièrement de l'architecture et des hyperparamètres choisis. Il est intéressant de remarquer que ce calcul est très simple et ne demande pas une forte puissance de calcul. Par conséquent, la phase d'inférence est très légère et sera donc rapide à exécuter.

C'est donc ainsi que fonctionne les intelligences artificielles par apprentissage profond. Il existe de nombreuses architectures au sein de cette famille (réseau de neurones convolutionnels, perceptron multicouche, etc.) de complexité et d'utilité différentes.

3 - Base de données

La base de données est un élément capital dans le processus de création d'une intelligence artificielle. Les données stockées seront utilisées pour entraîner, valider et tester le modèle que l'on créer. Il est donc important d'avoir une base de données la plus complète et précise possible.

Dans le cadre de ce projet, la base de données est constituée de dix images de chiffres allant de 0 à 9, écrit à la main, en 28 pixels par 28 pixels. Nous avons donc une base de données brut de 100 images, ce qui représente relativement peu de données. Lors du remplissage de la base de données, la stratégie utilisée était d'écrire les 5 premiers chiffres de manière classique, puis, les cinq derniers, de manière moins habituelle. L'objectif est d'avoir la moitié de la base de données avec une écriture habituelle, et l'autre moitié, une écriture moins commune, pour varier les styles et donc l'adaptabilité de l'intelligence artificielle.

Figure 3 - Exemple caractères brut, normal et varié

Un traitement de ces images a été réalisé pour les découper, afin d'enlever les espaces « inutiles » autour des chiffres. Un second traitement, a permis de mettre les images en noir et blanc, afin de faciliter l'interprétation des chiffres par le modèle.

Figure 4 - Exemple caractère découpé et mis en noir et blanc

Nous verrons, au fil de ce rapport, différents tests mettant en évidence les avantages apportés par ces traitements, ainsi que certaines optimisations et augmentation de données qui auront pour but de rendre la base de données plus riche et adaptable à différentes écritures. Tout cela dans le but de pouvoir entraîner le modèle à reconnaître les caractères de n'importe qui avec la meilleure précision possible.

4 – Développement de l'intelligence artificielle

4.1 – Environnement de développement

Tous les développements et tests sont réalisé dans des conteneurs ubuntu, embarquant python 3.10 avec les librairies : tensorflow, matplotlib, numpy, pillow.

4.2 – Détermination du meilleur modèle

Cette partie est dédiée à la description de toutes les étapes qui ont mené à l'établissement des hyperparamètres les plus optimisés. Cette étape est très importante, elle conditionne grandement l'efficacité du modèle que l'on créer. Il est donc nécessaire de réaliser de nombreux tests et de s'assurer de ne pas faire de surapprentissage.

La première étape a été de trouver la meilleure architecture avec le nombre « d'epochs » et une taille de « batch size » la plus adaptée. Le nombre d'epochs représente le nombre de fois que les données d'entraînement seront parcourues lors de la phase d'entraînement. La valeur donnée au batch size va modifier le nombre de données parcouru durant une epoch. Par exemple, pour un batch_size d'un, à chaque epoch, toutes les données seront parcourues. A 2, la moitié des données seront consultées à chaque epoch. La notion d'architecture décrit le nombre et le type de couches que l'on va utiliser pour notre modèle.

Dans un premier temps, des tests ont été réalisés pour tester l'efficacité de trois architectures de perceptron multicouche différentes. Voici leurs spécificités :

```
tf.keras.layers.Flatten(input_shape=(28, 28, 1)),
tf.keras.layers.Dense(512, activation='relu', kernel_initializer=tf.keras.initializers.GlorotNormal()),
tf.keras.layers.Dropout(0.3),
tf.keras.layers.Dense(10, activation='softmax', kernel_initializer=tf.keras.initializers.GlorotNormal())
```

Figure 1 - Architecture 1

```
tf.keras.layers.Flatten(input_shape=(28, 28, 1)),
tf.keras.layers.Dense(512, activation='relu', kernel_initializer=tf.keras.initializers.GlorotNormal()),
tf.keras.layers.Dropout(0.3),
tf.keras.layers.Dense(256, activation='relu', kernel_initializer=tf.keras.initializers.GlorotNormal()),
tf.keras.layers.Dropout(0.2),
tf.keras.layers.Dense(10, activation='softmax', kernel_initializer=tf.keras.initializers.GlorotNormal())
```

Figure 2 - Architecture 2

```
tf.keras.layers.Flatten(input_shape=(28, 28, 1)),
tf.keras.layers.Dense(512, activation='relu', kernel_initializer=tf.keras.initializers.GlorotNormal()),
tf.keras.layers.Dropout(0.3),
tf.keras.layers.Dense(256, activation='relu', kernel_initializer=tf.keras.initializers.GlorotNormal()),
tf.keras.layers.Dropout(0.2),
tf.keras.layers.Dense(128, activation='relu', kernel_initializer=tf.keras.initializers.GlorotNormal()),
tf.keras.layers.Dropout(0.1),
tf.keras.layers.Dense(10, activation='softmax', kernel_initializer=tf.keras.initializers.GlorotNormal())
```

Figure 3 - Architecture 3

Ces différentes architectures ont été testé en allant de 1 à 50 epochs, 10 fois par epoch afin d'avoir une moyenne de leurs efficacités. Le batch size a été fixé à 1 au vu de la faible quantité de données dans la base, il est important qu'elle soit parcourue en entière. Les résultats sont visibles dans la figure ci-dessous :

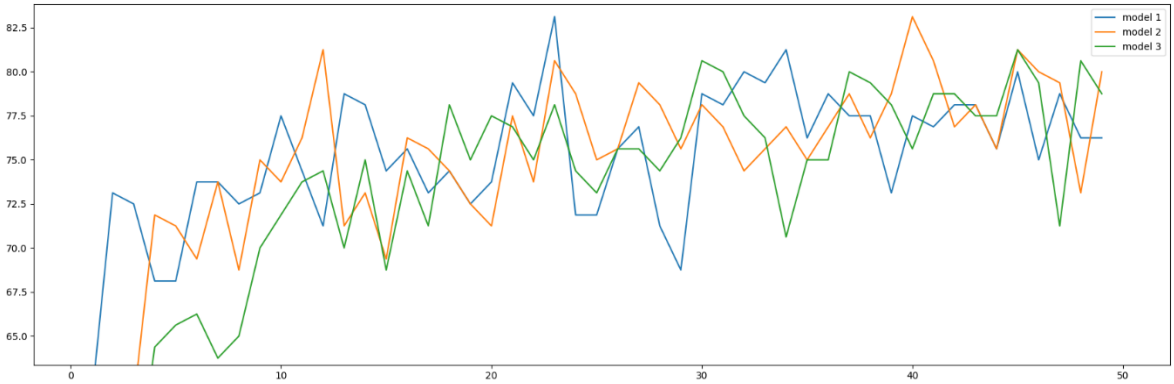


Figure 4 - Résultat test architecture

La lecture et l'interprétation de ces résultats est assez difficile. Il y a, pour chaque modèle, une grande variation dans la précision sur des epochs relativement proche. Néanmoins, on peut facilement observer que le model numéro 3, correspondant à l'architecture 3 présenté plus tôt, a une précision maximale d'environ 80% et est globalement le moins précis des deux autres architectures. Cette architecture peut donc être exclu du meilleur choix. La première et seconde architecture ont un maximum très proche aux alentours de 82.5%. L'une à 23 epochs et l'autre à 39 epochs. Etant donné la quantité de données relativement faible de la base, un maximum atteint avec le moins d'epochs possible est privilégié pour éviter le surapprentissage. L'architecture 1 ressort donc comme la plus précise.

Après avoir obtenu le meilleur modèle, il faut fixer le nombre de neurones la plus optimisé. Pour cela, un test à été réalisé avec un nombre de neurones à : 128, 256, 512, 1.5 * 784. Ces quantités ont été testées 50 fois puis une moyenne a été réalisée. Il en ressort que 256 neurones est la quantité la plus optimisée.

4.3 – Détermination de la meilleure base de données

Tous les tests réalisés jusqu'à maintenant ont été faits à partir de la base de données initiales composé de 100 images. Dans cet échantillon, 80 sont utilisées pour l'entraînement, 10 pour la validation pendant la phase d'entraînement et les 10 restantes pour les tests. Cela représente peu de données et cela impact forcément la qualité du modèle que l'on créer. C'est pour cette raison qu'il a été décidé de créer 2 autres bases de données, basé sur celle de base, afin de voir si l'on peut obtenir de meilleurs résultats.

La première base de données va contenir 2 fois plus de données car elle va prendre les images découpées en couleur et en noir et blanc à la fois. La deuxième sera utilisée pour tester la « data augmentation ». L'objectif de ce procédé est de prendre une base de données et d'y appliquer des changements pour augmenter grandement sa taille. Dans le cas présenté ici, les images découpées en couleur et en noir et blanc ont été pivotées de 15° vers la gauche et la droite. On a donc multiplié la base précédente par 3, ce qui fait un total de 600 images.

Ces 3 bases de données ont donc été testées avec l'architecture déterminée dans le chapitre précédent. Les données utilisées pour la validation seront, cette fois-ci, celle de Thomas Arthozoul. Les tests seront quant à eux réalisés à la fois sur la même base que les précédents tests, c'est-à-dire 20 images découpées en noir et blanc, mais aussi en second temps sur une base de 100 images appartenant à Gauthier Montagne afin de tester la robustesse des différentes bases de données face à des écritures différentes.

4.3.1 Test sur les données initial

Chacun de ces tests donne la moyenne des résultats avec un nombre d'epochs allant de 1 à 50 epochs, avec chaque epochs testé 8 fois sur les 20 images de la base de données initiale. Une courbe représentant l'écart-type est aussi affiché pour s'assurer de la stabilité des résultats. Nous observons les résultats suivants :

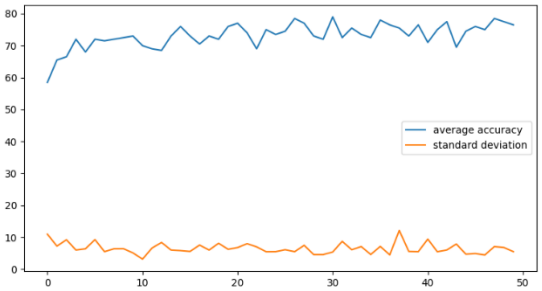


Figure 6 - base de données initiale

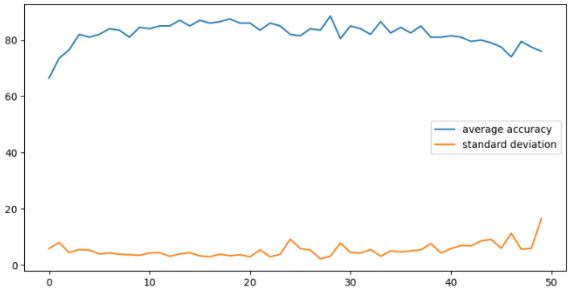


Figure 5 - base de données semi augmenté

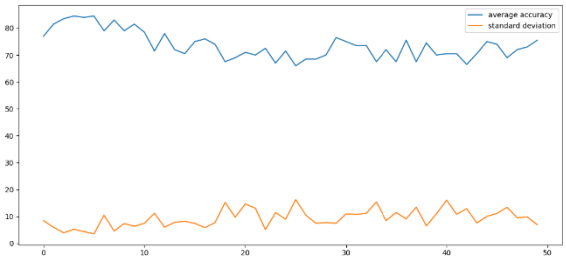


Figure 7 - base de données augmenté

La base de données initiales obtient des résultats plutôt faibles avec un maximum à environ 78% de précision. On observe un écart-type d'environ 4.

Pour la base « semi augmenté », les résultats sont très satisfaisants. Cette base de données atteint une précision jusqu'à environ 88% à 29 epochs avec un écart-type d'environ 8, ce qui est relativement élevé.

Enfin, la base de données augmentées atteint un maximum d'environ 88% avec un écart-type très faible d'environ 2.

On observe donc que la base de données initiales à de bien moins bons résultats que les bases augmentées et qu'on peut atteindre des résultats très satisfaisant avec celles-ci, notamment la base augmentée qui offre une stabilité des résultats très intéressante.

4.3.2 Test des données extérieurs

Les tests suivants ont été conduits dans les mêmes conditions que les précédents, à l'exception que les données testées proviennent de Gauthier Montagne. Ces tests permettent de tester la robustesse des 3 bases de données face à une base contenant une écriture différente. Voici les résultats obtenus :

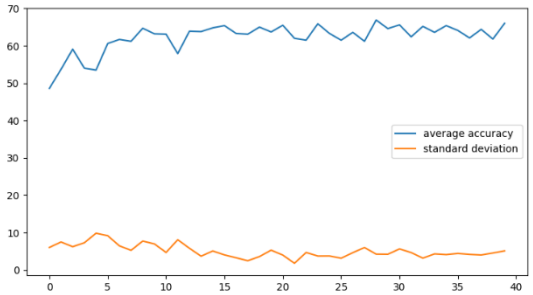


Figure 10 - test 2 base de données initial

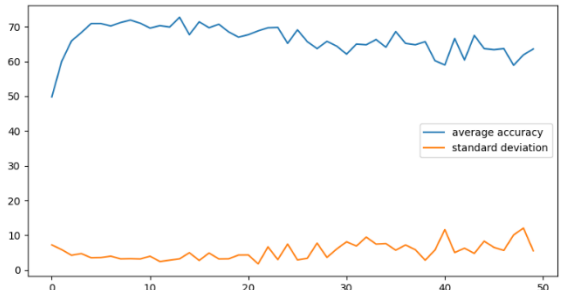


Figure 9 - test 2 base de données semi augmenté

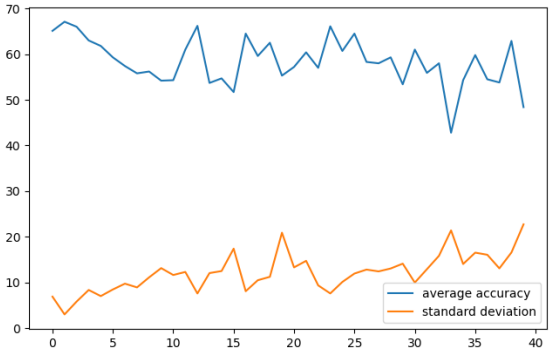


Figure 8 - test 2 base de données augmenté

On observe globalement que le profil des courbes des 3 bases sont similaires aux précédents tests. Néanmoins, cette fois-ci, les résultats sont globalement moins bons. Ce qui est prévisible au vu des données de cette base de test.

La base de données initiales offre encore une fois les moins bons résultats avec une précision maximale à environ 66% et un écart-type aux alentours de 6 à 27 epochs. On peut remarquer qu'à partir de 13 epochs, les résultats sont très stables et de précision correcte.

La base semi-augmenté offre cette fois-ci les meilleurs résultats avec une précision maximale à environ 73% et un écart-type d'environ 4 ce que l'on peut considérer comme bon. Cette base de données à de bons résultats avec une bonne stabilité jusqu'à environ 22 epochs.

La base augmentée offre aussi un résultat maximal élevé à environ 68% avec un écart-type à 3,5 à 3 epochs. On remarque tout de fois une grosse chute de précision et de stabilité une fois passé 4 epochs.

Nous pouvons donc conclure que la base de données semi-augmentée (contenant les images découpées en couleur et en noir et blanc à la fois) la plus optimisée. Elle obtient de très bons résultats sur des données « proches » de ses données d'entraînement, mais aussi avec des données « extérieures ». La base de données augmenté offre les meilleurs résultats sur les données proches, mais on a pu observer que sur une écriture différente elle ne n'arrive pas à atteindre la semi augmentée. L'hypothèse retenue est que l'augmentation des données ayant été faite uniquement sur des données « proches », cela a rendu le modèle très précis sur ces mêmes données mais moins adaptable à d'autres. L'augmentation des données semble donc être une très technique très efficace pour améliorer la précision de son modèle, cependant, pour le rendre le plus robuste possible face à différentes écritures, il aurait aussi fallu utiliser des données extérieures.

5 - Développement de la phase d'inférence

L'objectif du projet est d'avoir une intelligence artificielle pouvant être utilisé sur un système embarqué. Pour cette raison, le développement de la phase d'inférence sera fait en langage C.

Les fonctions de chargement et de mise en niveau de gris des images ont été fournies. Nous allons ici voir les développements réalisés nécessaire à l'inférence d'une architecture d'un perceptron multicouche et d'un réseau neuronal convolutif.

5.1 – Inférence d'un perceptron multicouche

Le développement de la phase d'inférence consiste à utiliser les poids et biais extrait du modèle précédemment entraîné pour réaliser le calcul suivant pour chaque couche :

$$y_{neural} = f\left(\left(\sum_{i \in \mathbb{N}} w_i * x_i\right) + b\right)$$

Figure 1 - Calcul d'une couche d'un perceptron

On peut traduire cette formule par un calcul matriciel entre un vecteur d'entrée (x) et une matrice de poids (w), puis, une addition du vecteur issue du résultat précédent avec un vecteur de biais (b). Cette opération est à réaliser à chaque « Dense » présent dans le modèle entraîné. L'ensemble des opérations effectué s'appelle une « séquence ». La séquence complète se déroule comme présenté dans la figure ci-dessous :

```
float *flattenPixels = (float *)malloc(28 * 28 * sizeof(float));
flatten(mPixelsGray, flattenPixels);
```

```
getBiases(bias, pfichierBiases, sizeBias);
getWeights(weights, pfichierWeights, sizeWeights, sizeBias);

dense(layer, flattenPixels, weights, bias, sizeWeights, sizeBias);

for(int i = 0; i < sizeBias; i++)
{
    flattenPixels[i] = layer[i];
}
```

Figure 2 - Code résumé d'une période d'une séquence

Ce code est résumé dans un souci de lisibilité. Le code présenté dans le second cadre est placé dans une boucle « while » qui va boucler tant qu'il y a des couches à lire.

La première étape de la séquence consiste à aplatir notre matrice d'image d'entrée en un vecteur. En effet, pour effectuer le calcul matriciel, il faut que « x » soit un vecteur, or, pour l'instant notre image chargée est une matrice 28 * 28. La fonction présentée dans la figure ci-dessous va effectuer cette opération :

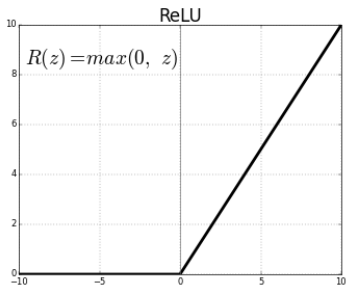
```
void flatten(unsigned char **mPixelsGray, float *flattenPixels)
{
    for (int i = 0; i < 28; i++)
    {
        for (int j = 0; j < 28; j++)
        {
            flattenPixels[i * 28 + j] = (float)mPixelsGray[i][j];
        }
    }
}
```

Figure 3 - Fonction flatten 2 dimensions

Les prochaines étapes de la séquence se déroulent dans la boucle while. En premier lieu, on charge les données de poids et de biais et on récupère leurs tailles pour les prochains calculs. La condition de fin de la boucle intervient lorsqu'il n'y a plus de fichier de poids ou de biais à lire. Après avoir chargé ces informations, le calcul du Dense (le produit et l'addition matricielle), peut avoir lieu. On remarque, dans le code présenté dans la figure 2, que le premier argument de la fonction est « layer ». Cette variable est en fait un tableau temporaire de la taille du vecteur de sortie. Ce tableau va prendre les résultats des calculs, puis, le vecteur flattenPixels sera vider de sa mémoire pour être remplacé par les valeurs de layer. De cette manière, pour la prochaine itération de la boucle, le vecteur flattenPixels aura comme valeurs le vecteur de sortie du précédent Dense. De cette manière, peu importe le nombre de couches du modèle que l'on infère, le résultat sera bon.

Pendant l'entraînement d'une intelligence artificielle, une fonction d'activation est spécifiée pour chaque couche. Ces fonctions permettent d'introduire de la non-linéarité et ainsi à apprendre et modéliser des relations complexes entre les données. Dans le cas présenté ici, la fonction « ReLu » est utilisée pour les couches cachées intermédiaires et la fonction « Softmax » pour la couche de sortie. Ces fonctions sont définies de la manière suivante :

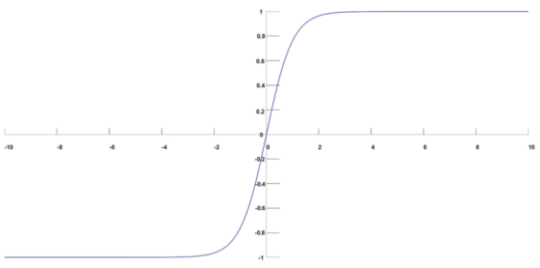
ReLu :



ReLu permet d'éviter le problème de saturation, car pour chaque entrée positive, la dérivée est constante. Cela facilite notamment la descente de gradient lors de la rétropropagation.

Softmax :

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{ pour tout } j \in \{1, \dots, K\},$$



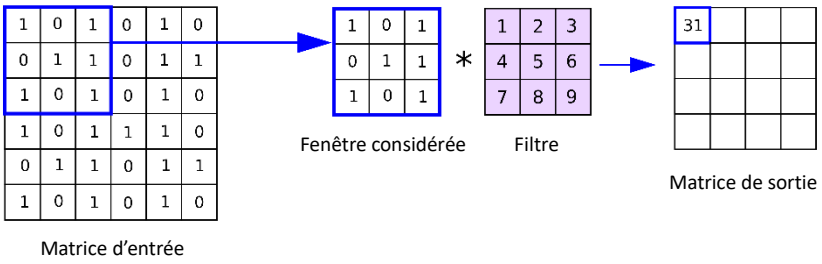
La fonction softmax est très utilisée en classification, car elle permet de normaliser les sorties de la dernière couche.

Ainsi, après avoir parcouru tous les fichiers de poids et biais, le programme nous fournit une prédiction pour chaque classe.

5.2 - Inférence d'un CNN

Le développement de la phase d'inférence d'un réseau de neurones convolutif est très différent d'un MLP (multi layer perceptron). L'architecture du modèle d'entraînement est simple pour faciliter le développement de l'inférence. Le modèle est composé d'une couche de convolution, d'une couche flatten et d'un Dense de sortie.

La couche de convolution prend en argument le nombre de « feature map » et le taille des filtres appliqué. L'objectif d'une convolution est de multiplier une « fenêtre » de données de la matrice que l'on d'entrée par un filtre. La figure ci-dessous montre comment se déroule l'opération :



Ici, le filtre est de taille 3 * 3 et une seule convolution est effectué. On multiplie chaque case de la fenêtre par la valeur au même indice du filtre. Dans cet exemple, on fait donc : 1*1 + 0*2 + 1*3 + 0*4 + 1*5 + 1*6 + 1*7 + 0*8 + 1*9 = 31. Le nombre de « feature map » désigne donc le nombre de matrices de sortie que l'on va avoir à la suite des convolutions. Il y aura ainsi autant de filtres que de feature map, et leurs valeurs seront fixée pendant l'entraînement par le modèle. Durant l'extraction des biais, il y a aussi une valeur de biais pour chaque filtre qui sera additionné à chaque résultat de convolution. La matrice de sortie de ces convolutions forme donc une matrice trois dimensions.

Une convolution se déroule de la manière suivante :

```
void conv2D(float **image, float ***filters, float ***convOutput, float *bias, int nFilters)
{
    for(int i = 0; i < nFilters; i++)
    {
        for(int j = 0; j < 26; j++)
        {
            for(int k = 0; k < 26; k++)
            {
                for(int l = 0; l < 3; l++)
                {
                    for(int m = 0; m < 3; m++)
                    {
                        convOutput[i][j][k] += image[j + l][k + m] * filters[i][l][m];
                    }
                }
                convOutput[i][j][k] += bias[i];
                convOutput[i][j][k] = relu(convOutput[i][j][k]);
            }
        }
    }
}
```

Figure 4 - Code d'une convolution

Cette fonction commence par une boucle « for » qui va itérer sur chaque filtre. Ensuite, une double boucle parcourt la matrice d'entrée, donc l'image. On remarque que cette double boucle fait à chaque fois 26 itérations et non pas 28 comme la taille de l'image. En effet, lors de la convolution, on applique un filtre 3*3 à chaque pixel. A partir du 26e pixel, si l'on cherche a créer une fenêtre 3*3, on va dépasser la taille réelle de l'image. Pour éviter cela, on limite la taille de l'image sur laquelle on itère. Ainsi, la feature map créer à partir de l'image de base aura pour dimensions : 26 * 26 * le nombre de feature map. Enfin, on itère sur une autre double boucle for qui va parcourir l'image selon la taille du filtre

La dernière étape consiste à faire un Dense comme pour un MLP. Pour faire ce calcul, il est nécessaire d'avoir en entrée un vecteur. Or, pour l'instant, les informations sont contenues dans une matrice trois dimensions. Il faut donc effectuer la même opération que dans la fonction flatten du MLP, mais cette fois sur une dimension supplémentaire. Pour cela, il faut aligner chaque donnée de la feature map par la profondeur, comme montrer dans la figure ci-dessous :

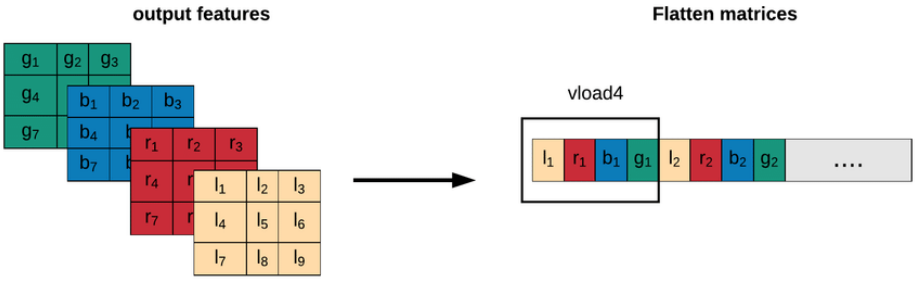


Figure 5 - Flatten 3 dimensions

Le code de cette fonction est en annexe. Une fois cette opération effectuée, on peut alors faire le calcul matriciel classique d'un Dense et avoir la prédiction du modèle.

6 – Conclusion

Pour conclure, nous avons été capable de développer une intelligence artificielle capable d'être déployées sur un système embarqué. Les différents tests menés on surtout mis en évidence l'importance de la base de données d'apprentissage. Une base de données riche et variée offrira les meilleurs résultats, comme nous avons pu le constater.

Une piste d'amélioration pour le système actuelle serait donc de récupérer les données des autres personnes travaillant sur le même projet afin d'avoir un grand nombre d'écriture différente dans la base de données.

7 – Annexes

Code complet d'une séquence d'un perceptron multicouche :

```
int sequence(unsigned char **mPixelsGray)
{
    int i = 1;
    int prediction = 0;

    float *flattenPixels = (float *)malloc(28 * 28 * sizeof(float));
    flatten(mPixelsGray, flattenPixels);

    while (i != 0)
    {
        char weightsPath[100];
        char biasesPath[100];

        sprintf(weightsPath, "./../poids_et_biais/weights_%d.txt", i);
        FILE *pfichierWeights = fopen(weightsPath, "rb");
        if (pfichierWeights == NULL)
        {
            printf("Lecture terminée\n");
            prediction = predict(flattenPixels);
            break;
        }

        sprintf(biasesPath, "./../poids_et_biais/biases_%d.txt", i);
        FILE *pfichierBiases = fopen(biasesPath, "rb");
        if (pfichierBiases == NULL)
        {
            printf("Lecture terminée\n");
            break;
        }

        int sizeBias = countLines(pfichierBiases);
        int sizeWeights = countLines(pfichierWeights);

        float *bias = (float *)malloc(sizeBias * sizeof(float));
        getBiases(bias, pfichierBiases, sizeBias);

        float *weights = (float *)malloc(sizeWeights * sizeBias * sizeof(float *));
        getWeights(weights, pfichierWeights, sizeWeights, sizeBias);

        fclose(pfichierWeights);
        fclose(pfichierBiases);

        float *layer = (float *)malloc(sizeBias * sizeof(float)); // vecteur temporaire pour stocker les valeurs d'entrées de la nouvelle couche

        dense(layer, flattenPixels, weights, bias, sizeWeights, sizeBias);

        free(flattenPixels); // On free le vecteur d'entrée pour l'adapter a la prochaine couche
        flattenPixels = (float *)malloc(sizeBias * sizeof(float));

        for(int i = 0; i < sizeBias; i++)
        {
            flattenPixels[i] = layer[i];
        }
        printf("\n");
        free(bias);
        free(weights);
        free(layer);

        i++;
    }
    free(flattenPixels);
    return prediction;
}
```

Code complet d'un Dense :

```
void dense(float *layer, float *flattenPixels, float *weights, float *bias, int weightsSize, int biasSize)
{
    for(int i = 0; i < biasSize; i++)
    {
        layer[i] = 0;
    }

    for (int i = 0; i < weightsSize; i++)
    {
        for (int j = 0; j < biasSize; j++)
        {
            layer[j] += flattenPixels[i] * weights[i * biasSize + j];
        }
    }
    for(int i = 0; i < biasSize; i++)
    {
        layer[i] += bias[i];
    }

    if(biasSize == 10)
    {
        softmax(layer, biasSize);
    }
    else
    {
        for (int i = 0; i < biasSize; i++)
        {
            layer[i] = relu(layer[i]);
        }
    }
}
```

Code de la fonction flatten 3 dimensions :

```
void flatten3d(float ***input, float *flattenPixels, int nFilters)
{
    for(int i = 0; i < 26; i++)
    {
        for(int j = 0; j < 26; j++)
        {
            for(int k = 0; k < nFilters; k++)
            {
                flattenPixels[nFilters * ((26 *i) + j) + k] = input[k][i][j];
            }
        }
    }
}
```

Code complet d'une séquence CNN :

```
int sequenceCnn(unsigned char **mPixelsGray, int nFilters)
{
    int filterSize = 3;
    int prediction = 0;

    float ***filtersWeights = (float ***)malloc(nFilters * sizeof(float**));
    for (int i = 0; i < nFilters; i++)
    {
        filtersWeights[i] = (float **)malloc(filterSize * sizeof(float*));
        for (int j = 0; j < 3; j++)
        {
            filtersWeights[i][j] = (float *)malloc(filterSize * sizeof(float));
        }
    }
    getCnnWeights(filtersWeights, nFilters);

    float *bias = (float *)malloc(nFilters * sizeof(float));
    getCnnBiases(bias, nFilters);

    float ***convOutput = (float ***)malloc(nFilters * sizeof(float**));
    for (int i = 0; i < nFilters; i++)
    {
        convOutput[i] = (float **)malloc(26 * sizeof(float*));
        for (int j = 0; j < 26; j++)
        {
            convOutput[i][j] = (float *)malloc(26 * sizeof(float));
        }
    }

    float **image = (float **)malloc(28 * sizeof(float*)); // matrice qui cast l'image en float
    for (int i = 0; i < 28; i++)
    {
        image[i] = (float *)malloc(28 * sizeof(float));
    }
    for(int i = 0; i < 28; i++)
    {
        for(int j = 0; j < 28; j++)
        {
            image[i][j] = (float)mPixelsGray[i][j];
        }
    }

    conv2D(image, filtersWeights, convOutput, bias, nFilters);

    float *flattenConvOutput = (float *)malloc(26 * 26 * nFilters * sizeof(float));

    flatten3d(convOutput, flattenConvOutput, nFilters);

    FILE *pfichierWeights = fopen("./../poids_et_biais/cnn_weight_dense.txt", "rb");
    FILE *pfichierBiases = fopen("./../poids_et_biais/cnn_bias_dense.txt", "rb");

    int sizeBias = countLines(pfichierBiases);
    int sizeWeights = countLines(pfichierWeights);

    float *biasDense = (float *)malloc(sizeBias * sizeof(float));
    getBiases(biasDense, pfichierBiases, sizeBias);

    float *weightsDense = (float *)malloc(sizeWeights * sizeBias * sizeof(float *));
    getWeights(weightsDense, pfichierWeights, sizeWeights, sizeBias);

    fclose(pfichierWeights);
    fclose(pfichierBiases);

    float *output = (float *)malloc(sizeBias * sizeof(float));

    dense(output, flattenConvOutput, weightsDense, biasDense, sizeWeights, sizeBias);

    prediction = predict(output);

    free(flattenConvOutput);
    free(bias);
    free(biasDense);
    free(weightsDense);
    free(output);
    free(filtersWeights);
    free(convOutput);

    return prediction;
}
```