

Roann CANTEL  
Daniel AKBARINIA  
Edouard POMPEE  
Antoine HEITZMANN

Encadrant :  
Leonardo LINGUAGLOSSA

## Rapport final :

# Apprentissage automatique avec tinyML dans les réseaux logiciels

## Sommaire :

<b>1. Introduction .....</b>	<b>2</b>
<b>2. Contexte .....</b>	<b>3</b>
<b>3. Travail réalisé .....</b>	<b>5</b>
<b>4. Conclusion et future work .....</b>	<b>22</b>
<b>5. Références .....</b>	<b>23</b>

## Introduction :

Le projet "Apprentissage automatique avec TinyML" vise à implémenter des algorithmes de machine learning, notamment de deep learning (réseaux de neurones), sur des microcontrôleurs à faible consommation énergétique. Nous développons nos modèles en langage C pour optimiser l'efficacité, la taille, la consommation, le temps d'entraînement et d'exécution. En outre, le tinyML soulève différents enjeux :

- Les enjeux sociaux incluent la confidentialité des données en traitant localement les informations et l'amélioration de l'accessibilité technologique pour les communautés marginalisées.
- Les enjeux environnementaux concernent la réduction de la consommation énergétique et l'application de TinyML dans des solutions écologiques et durables, comme la gestion optimisée de l'irrigation agricole.
- La possibilité de développer du machine learning à la latence très faible, ne nécessitant pas la communication avec un serveur distant.

Pour ce faire, nous avons fondé notre projet sur la librairie GenANN, implémentant en C les fonctions nécessaires à l'implémentation la plus simple d'un Dense Neural Network. Nous nous sommes fixé pour objectifs d'implémenter un test sur le dataset MNIST, et d'améliorer la librairie pour la mettre à jour, notamment avec l'implémentation des CNN, ou Convolutional Neural Network. A terme, nous souhaitons avoir une application capable de mettre en place le TinyML avec les programmes réalisés, et dotée d'une interface réseau pour pouvoir implémenter une situation réelle.

## Contexte :

Le machine learning est un domaine d'étude de l'intelligence artificielle, qui repose sur l'apprentissage semi-autonome d'un programme via un modèle mathématique. Grâce à de larges quantités de données pré-traitées (données d'entraînement), le logiciel apprend à reconnaître une corrélation entre ces données pour déterminer un résultat le plus juste possible sur des données qui n'ont pas encore été analysées (données de test ou réelles). Ce domaine a eu un grand gain de popularité dans les dernières années pour 3 raisons :

- Premièrement, la mise en circulation de nombreux appareils informatiques et capteurs, notamment avec l'IoT (IoT = Internet of Things, la connexion à internet de nombreux objets dotés de capteurs), est à l'origine d'une très grande quantité de données à analyser pour des nombreuses raisons : analyse du comportement, prévisions météo, etc...
- Des processeurs spécialisés aux calculs nécessaires au machine learning ont nettement progressé (GPU et TPU = Graphics/tensor processing unit).
- Des algorithmes de plus en plus performants et efficaces ont vu le jour.

L'importance croissante des appareils IoT dans divers domaines (santé, automobile et les dispositifs portables) rend essentiel le développement de modèles de Machine Learning efficaces sur des plateformes limitées par leur batterie et leur performance. Cependant, le gain de popularité du Machine Learning tel qu'il existe principalement (centralisé) présente ses limites :

Utiliser les capteurs de l'IoT pour analyser leurs données à distance implique une latence élevée, tandis que l'utilisation des processeurs spécialisés pour la performance nécessite une quantité d'énergie phénoménale, et entraîne un coût de production supplémentaire. Une solution à ces deux problèmes est l'*edge computing*: le traitement non centralisé des données, notamment en local sur les différents systèmes IoT. Cela implique de réaliser du machine learning sur des ordinateurs très peu puissants, rarement équipés de processeurs spécialisés. Cela représente le TinyML, une forme de Machine learning en plein essor et marquant une convergence entre l'IoT et le Machine Learning : effectuer des calculs de Machine Learning sur des microprocesseurs à la consommation extrêmement basse.

Le concept de TinyML a été inventé par des chercheurs cherchant à intégrer des modèles de Machine Learning sur des microcontrôleurs à faible puissance dès

le début des années 2010. Depuis son invention, des avancées significatives ont été réalisées dans l'optimisation des algorithmes, la réduction de la taille des modèles et l'adaptation aux contraintes matérielles spécifiques des microcontrôleurs. Le TinyML utilise notamment la compression de modèle, la quantification, et l'optimisation des hyperparamètres pour des performances optimales sur nos systèmes.

Différents modèles sont utilisés dans le TinyML :

- les réseaux de neurones profonds (DNN)
- les réseaux de neurones convolutionnels (CNN)
- les machines à vecteurs de support (SVM)
- les méthodes d'apprentissage non supervisées comme le clustering (a priori non abordé dans ce projet)

Pourtant, des lacunes dans ce domaine existent et comprennent la nécessité de développer des méthodes de validation et de débogage spécifiques au TinyML, ainsi que l'exploration de nouvelles architectures de modèle adaptées aux contraintes matérielles.

Le TinyML offre donc des opportunités prometteuses pour l'expansion du machine learning dans le cadre de l'IoT. L'enjeu du projet est ainsi de faire fonctionner un algorithme de machine learning le plus efficacement possible sur un micro contrôleur (a priori un raspberry pi), et de le faire interagir en réseau avec d'autres instances du modèle. Notre stratégie est donc assez claire : nous allons tester différents algorithmes de machine learning pour comprendre leur efficacité (définie ci-dessous dans les tâches), et en complexifiant peu à peu les exercices et les datas à analyser et classifier.

## **Ce que nous avons réalisé**

### **Début mars -> Fin mars : Implémentation d'un modèle de classification d'image**

- **Etude de la librairie Genann:**

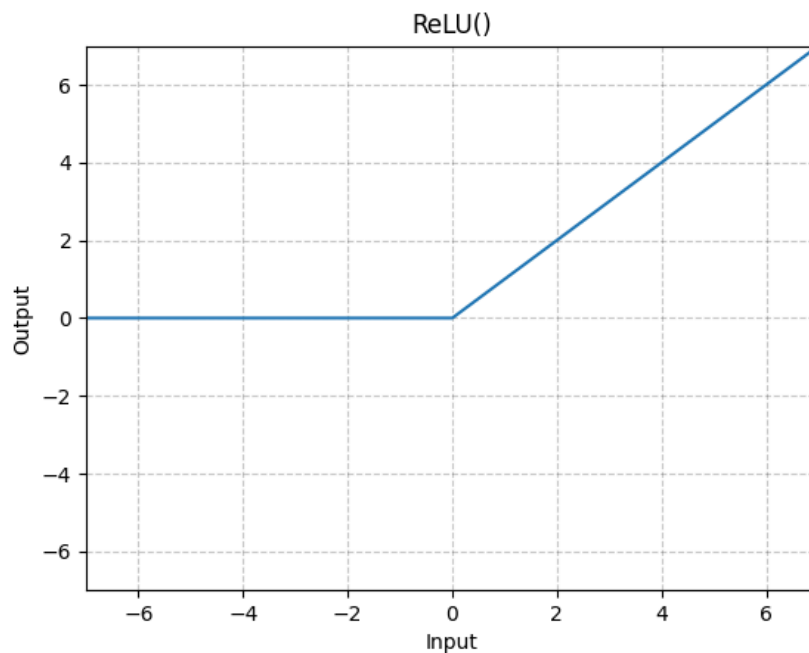
Genann est une bibliothèque logicielle open source conçue pour mettre en œuvre des réseaux de neurones artificiels (ANN), qui sont les réseaux de neurones les plus simples, dans des applications informatiques. Le fichier genann.c implémente une structure genann de ANN et l'utilisateur a le choix du nombre d'entrées, de couches de neurones cachées, du nombre de neurones par couche cachée et du nombre de sortie comme suit:

```
genann *genann_init(int inputs, int hidden_layers, int hidden, int outputs);
```

Cette bibliothèque implémente aussi genann\_train, une fonction qui permet d'entraîner un genann à partir d'une unique entrée, d'une sortie et d'un taux d'apprentissage (learning rate en anglais). Genann implémente donc une descente de gradient stochastique (SGD) plutôt qu'une descente de gradient mini-batch. La SGD est rapide car elle effectue des mises à jour fréquentes des poids, mais elle peut être moins stable et peut nécessiter plus d'itérations pour converger vers une solution optimale.

```
void genann_train(genann const *ann, double const *inputs, double const *desired_outputs, double learning_rate);
```

Genann propose une variété de fonctions d'activations pour les couches cachées et la sortie et la fonction genann\_train adapte ses équations de rétropropagation (algorithme qui met à jour les poids après passage dans le réseau d'une entrée) aux fonctions d'activation choisies. Cependant, un des défauts majeurs de genann est qu'il n'implémente pas la fonction d'activation ReLU (qui vaut 0 pour  $x \leq 0$  et  $x$  sinon), une fonction d'activation généralement utilisée dans les couches cachées qui donne les meilleurs résultats en terme de précision (accuracy en anglais) et de vitesse de convergence. Ce point va donner lieu à une amélioration de genann.



Graphe de la fonction ReLU

```
double genann_act_sigmoid(const genann *ann, double a);  
double genann_act_sigmoid_cached(const genann *ann, double a);  
double genann_act_threshold(const genann *ann, double a);  
double genann_act_linear(const genann *ann, double a);
```

#### Genann n'implémente pas ReLU

Genann nous paraît être la meilleure solution pour l'entraînement et le déploiement de modèles basés sur les ANN. Dans l'approche TinyML, qui est celle de notre projet, il existe TensorflowLite qui est une version allégée de la bibliothèque TensorFlow, spécialement conçue pour l'exécution de modèles d'apprentissage profond sur des appareils mobiles et des systèmes embarqués (comme des RaspberryPi) avec des ressources limitées. Toutefois, cette bibliothèque ne permet pas l'entraînement de modèles sur des systèmes embarqués et ne concerne que le déploiement de modèles.

- **Implémentation d'un test basé sur la base de donnée MNIST**

Afin d'entraîner notre modèle sur la base de données MNIST, nous avons développé un dataloader qui exploite la forme des fichiers binaires des images MNIST afin d'alimenter notre modèle ANN. Chaque image est représentée par une classe (0 à 9) donnant le numéro sur l'image et d'une suite de pixels matérialisé par double (8 octets) allant de 0 (noir) à 255 (blanc) qu'il nous a fallu normaliser à 1 en divisant par 255 car les ANN ont de meilleurs résultats sur des données peu étendues. Les ensemble de données

d'entraînement de test sont lues par notre dataloader et écrites à l'intérieur d'une structure de dataset comprenant la hauteur, la largeur, le nombre d'images, et les images elles-mêmes.

```
typedef struct image image;
struct image {
    int class;
    double *pixels;
};
```

```
typedef struct dataset dataset;
struct dataset {
    size_t nimages;
    unsigned int width;
    unsigned int height;
    image *images;
};
```

```
int dataset_read(dataset *output, char *images_file, char *labels_file);
void dataset_free(dataset *dt);
```

### Structures et fonctions utilisées par notre dataloader

Une fois les images exploitables par un genann (un ANN), nous avons mis en place un algorithme d'entraînement de modèle. Cet algorithme donne successivement des images à un genann initialisé au préalable afin qu'il s'entraîne en utilisant la fonction `genann_train`. Une boucle entraîne notre modèle sur l'ensemble du dataset d'entraînement et le nouveau modèle est comparé au modèle de l'itération précédente à partir de leur précision sur le dataset d'entraînement (bonnes prédictions/total des prédictions). Si le modèle précédent est moins bon, on continue d'entraîner notre modèle sinon, on s'arrête et on garde le modèle précédent: c'est l'algorithme de early stopping. Ainsi, on arrête notre entraînement à temps pour ne garder que le meilleur modèle et ainsi éviter l'overfitting, c'est-à-dire que notre modèle se base trop sur les données d'entraînement, ayant appris le bruit et les détails spécifiques à l'ensemble de données d'entraînement. L'overfitting conduit à une performance excellente sur les données d'entraînement mais à une mauvaise généralisation sur de nouvelles données ou sur l'ensemble de données de test.

Avant le passage dans la boucle de early stopping, output est une liste avec dix 0.0 à la suite.

```
// Implémentation du "early stopping"
for(j = 0; is_new_ai_better(old_ann, ann, tests); j++) {

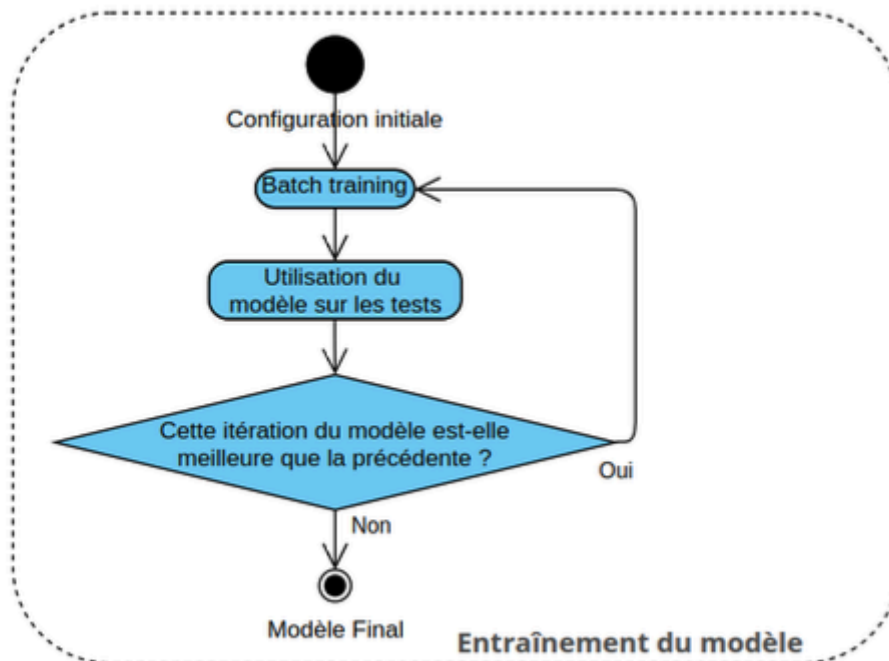
    // Remplacement de l'ancien modèle
    if(old_ann)
        genann_free(old_ann);
    old_ann = genann_copy(ann);

    // On entraîne l'IA sur toute la base de données
    for (i = 0; i < training.nimages; ++i) {
        printf("[Entraînement numero %d]: %zd%\r",
            j+1,
            (100 * (i+1)) / training.nimages
        );

        output[training.images[i].class] = 1;
        genann_train(ann, training.images[i].pixels, output, 0.1);
        output[training.images[i].class] = 0;
    }
    printf("\n");
}
```

Algorithme d'entraînement de nos modèles basé sur le schéma dessous





- **Mise en place du Docker**

Directement dans le projet, nous avons décidé d'utiliser docker afin d'entraîner et de déployer nos modèles de deep learning en raison de ses avantages de portabilité, de facilité de déploiement et de gestion de ressource. En effet, docker nous permet d'exécuter nos programmes à partir d'une couche minimale de Linux sans avoir recours à une machine virtuelle lourde en taille de données. Finalement, notre image docker avec notre modèle ANN peut être exécutée comme container sur n'importe quelle machine (comme un Raspberry Pi) avec docker et se présente comme suit pour le docker d'entraînement par exemple :

```

# Utilise une image de base Alpine Linux version 3.8
FROM alpine:3.8

# Déclare un volume pour persister des données et les rendre accessibles en dehors du conteneur
VOLUME /vol1/

# Définit le répertoire de travail dans le conteneur
WORKDIR /vol1/

# Installe le compilateur GCC et les bibliothèques de développement essentielles sans garder de cache
RUN set -ex && \
    apk add --no-cache gcc musl-dev

# Supprime certains fichiers exécutables de GCC qui ne sont pas nécessaires pour alléger l'image
RUN set -ex && \
    rm -rf /usr/libexec/gcc/x86_64-alpine-linux-musl/6.4.0/cc1obj && \
    rm -rf /usr/libexec/gcc/x86_64-alpine-linux-musl/6.4.0/lto1 && \
    rm -rf /usr/libexec/gcc/x86_64-alpine-linux-musl/6.4.0/lto-wrapper && \
    rm -rf /usr/bin/x86_64-alpine-linux-musl-gcj

# Copie les fichiers sources du répertoire local 'src' vers le répertoire de travail dans le conteneur
COPY src/ /vol1/

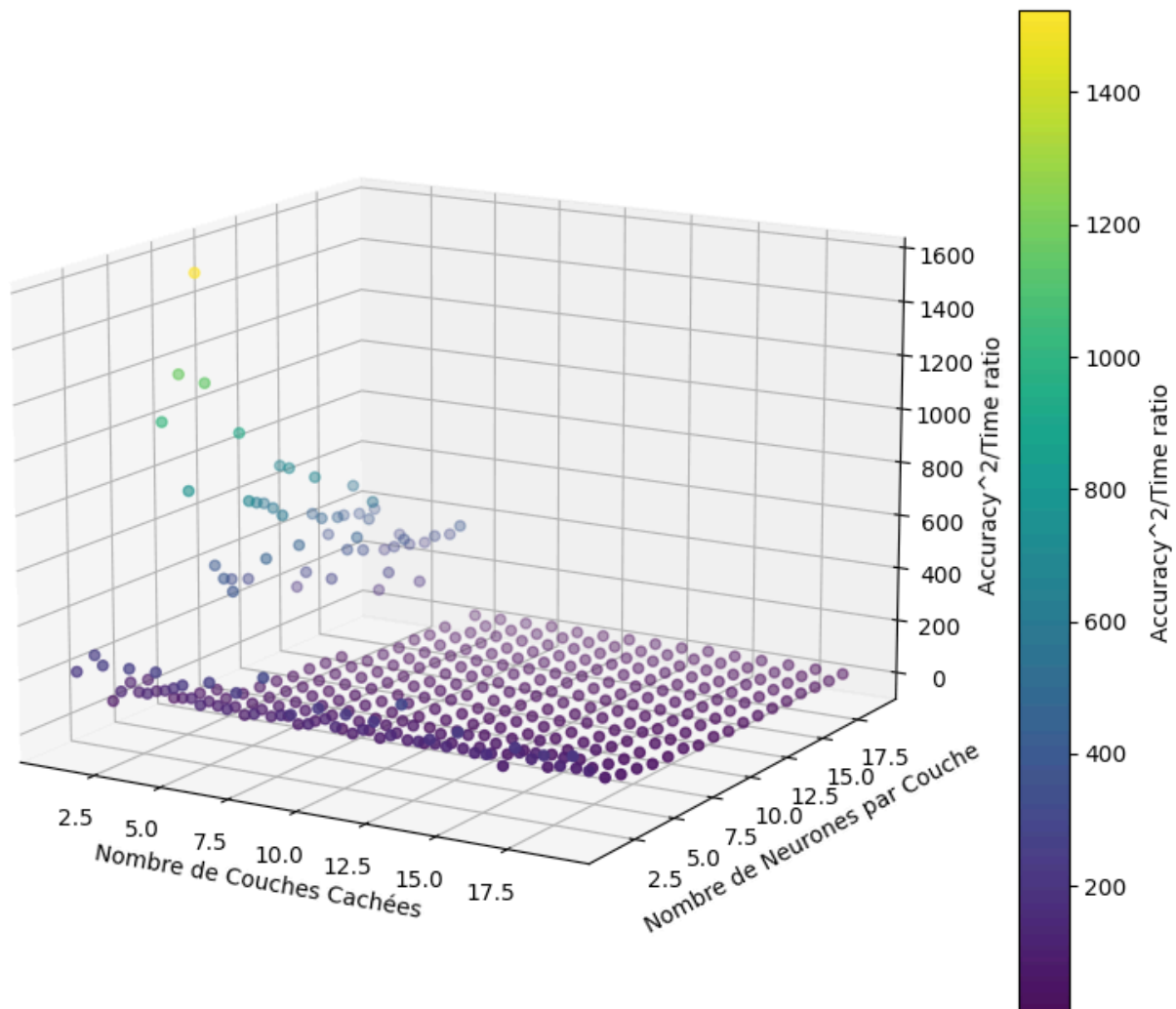
# Commandes pour compiler et exécuter le programme spécifié
CMD gcc -c mnist.c && \
    # Compile le fichier source mnist.c
    gcc -c genann.c && \
    # Compile le fichier source genann.c
    gcc -c mnist_db.c && \
    # Compile le fichier source mnist_db.c pour le dataloader
    gcc -o mnist mnist.o matrix.o genann.o mnist_db.o -lm && \
    # Lie les fichiers objet pour créer l'exécutable 'mnist'
    ./mnist 2 16
    # Exécute le programme 'mnist' avec les arguments spécifiés (2 couches cachées et 16 neurones dans chaque)

```

Fichier dockerfile d'entraînement de nos modèles. Pour le transformer en image, on fait `docker build -t (nom_de_l'image)`. Pour exécuter l'image en tant que container, on fait `docker run (id_de_l'image)`

- **Implémentation du calcul des indicateurs de performances**

Initialement, nous avons utilisé la trace de la matrice de confusion c'est-à-dire la précision sur les données test afin de comparer deux modèles dans l'early stopping et cela donne d'assez bon résultats. Cependant, quand on choisit parmi 2 modèles, généralement on ne garde que celui qui minimise la fonction de coût au sens de la log-vraisemblance négative (negative log-likelihood en anglais). Pour comparer nos modèles, il faut donc garder celui qui a la plus petite entropie croisée catégorique (*categorical cross entropy* en anglais). Nous avons intégré cette fonction à l'early stopping et n'avons constaté aucune différence de modèle produit si ce n'est un temps d'entraînement 5% plus long à cause du calcul de cette fonction de coût. Nous avons entraîné 361 modèles avec 1 à 19 couches cachées et de 1 à 19 neurones dans chaque couches cachées, utilisant partout sigmoïde comme fonction d'activation et voici les résultats :



Graphique en 3D donnant le ratio précision au carré divisé par le temps d'entraînement en fonction de hidden layer et de hidden. Les modèles utilisent partout la fonction sigmoïde comme fonction d'activation avec un learning rate de 0.3. On utilise  $\text{Accuracy}^2/\text{Time}$  plutôt que  $\text{Accuracy}/\text{Time}$  pour éviter de donner trop d'importance aux modèles qui ont une mauvaise précision (10%) en s'entraînant en une seule génération. Les modèles avec peu de couches cachées (1 à 2) sont favorisés

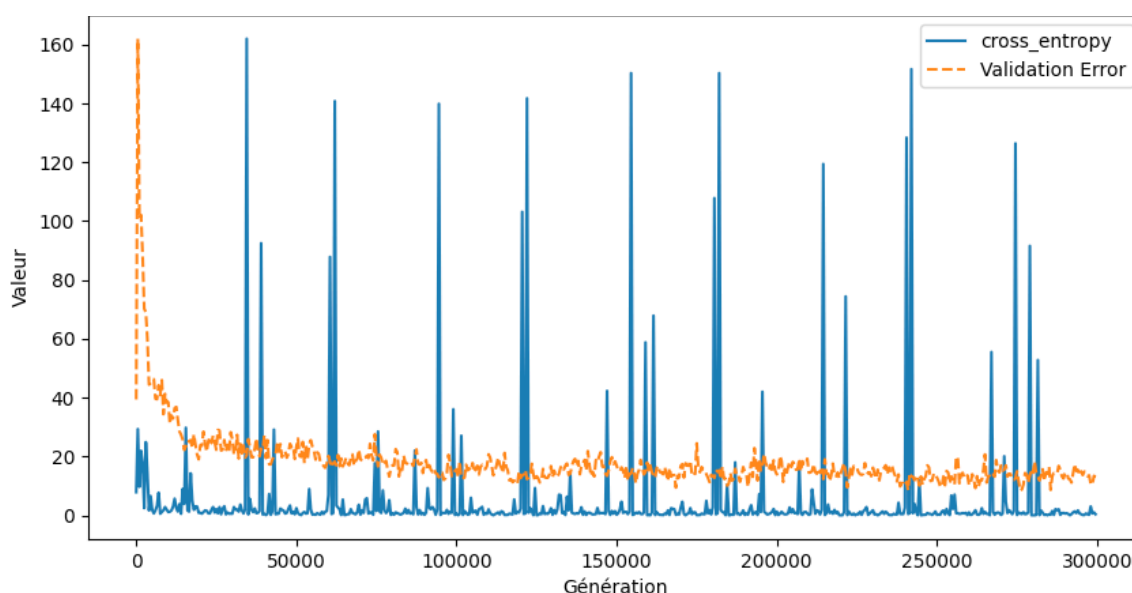
3 Meilleures Précisions	Input Layer	Hidden Layer	Neuron per hidden	Output	RAM (kb)	Flash (kb)	Training Time (s)	Prediction (ms)	Accuracy
	784	2	19	10	443.00	134.00	30.39	571.15	94.26
3 Meilleures précisions en MNIST ANN	784	2	18	10	442.00	127.00	34.48	520.57	94.02
	784	1	19	10	431.00	130.00	23.30	536.69	93.95

Tableau des 3 meilleurs modèles en terme de précision en fonction de hidden layer et hidden. On dépasse les 94% de précision. Les prédiction se font sur 10000 images en même temps

3 Meilleures Précisions*2/time	Input Layer	Hidden Layer	Neuron per hidden	Output	RAM (kb)	Flash (kb)	Training Time (s)	Prediction (ms)	Accuracy	Accuracy*2/Time
	784	1	8	10	427.00	62.00	5.07	225.42	87.90	1524.25
3 Meilleures Précisions*2/time en MNIST ANN	784	1	7	10	427.00	56.00	6.69	195.07	88.47	1170.12
	784	2	7	10	435.00	56.00	6.74	203.58	88.06	1150.53

Tableau des 3 meilleurs modèles en terme de précision au carré divisé par le temps d'entraînement en fonction de hidden layer et hidden. On dépasse les 87% de précision

Nous voulons un modèle qui a une bonne précision tout en gardant un temps d'entraînement plutôt faible. Le deuxième tableau nous incite à ne garder qu'un modèle avec 1 couche cachée et 8 neurones par couches cachées. C'est typiquement celui-ci qui sera utilisé sur le Raspberry Pi aussi bien en entraînement qu'en déploiement. La courbe d'entraînement pour ce modèle ressemble à cela:



Courbe d'apprentissage du modèle avec avec 1 couche cachée et 8 neurones par couches cachées, fonctions d'activation sigmoïde et learning rate de 0.3

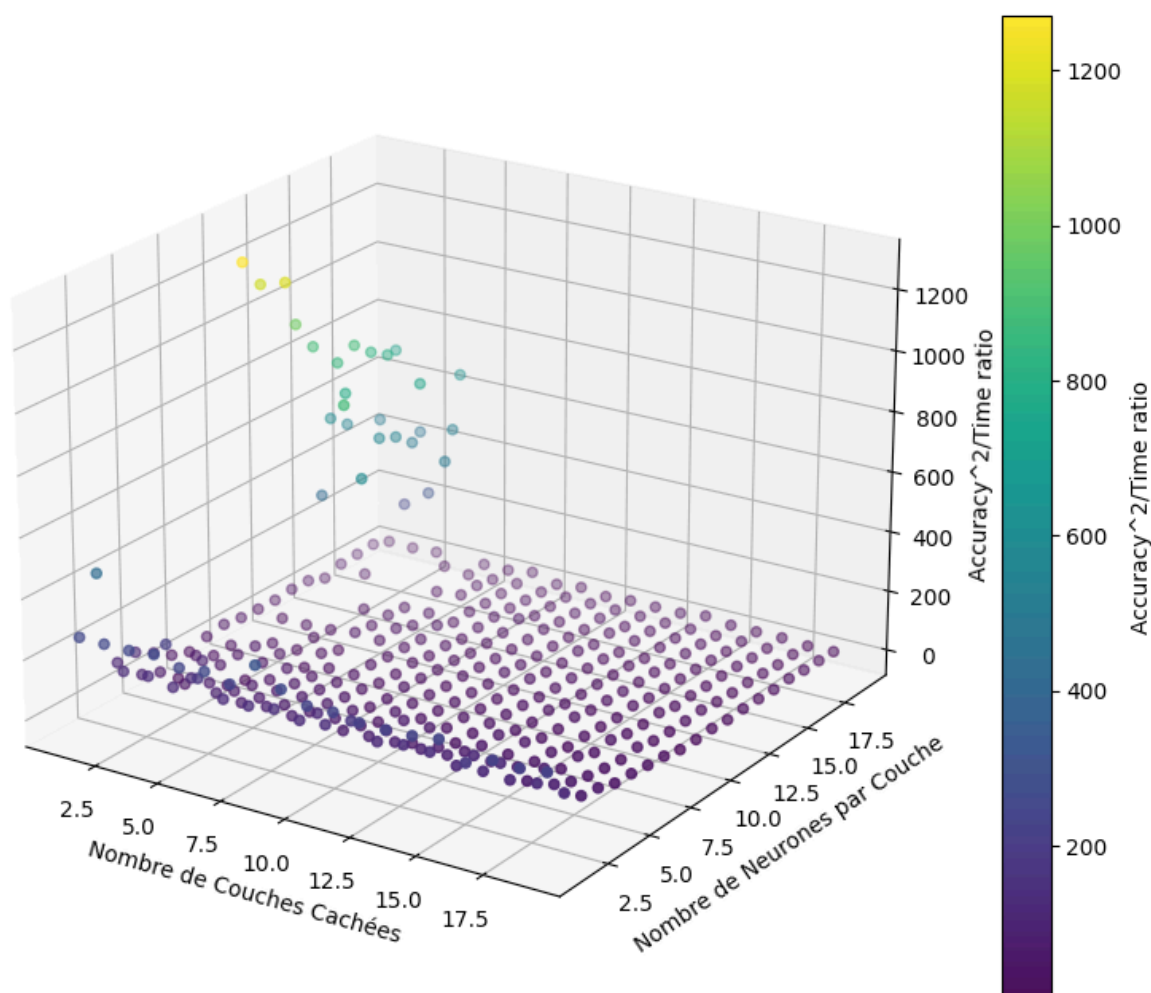
Cette courbe affiche deux variables en fonction du nombre d'images passées au modèle. Ici, le modèle a analysé 300 000 images, ce qui donne 6 générations car il y a 60 000 images par générations. Les courbes représentent:

- L'entropie croisée (catégorique ici) sur le dataset d'entraînement. Cette valeur ne fait que diminuer au cours de notre entraînement jusqu'à une valeur appelée erreur de Bayes pour laquelle les poids du réseau minimisent l'entropie croisée sur les données d'entraînement. Ici, quelques artefacts apparaissent aux multiples de 30 000 images à cause de la redondance des images.

- L'erreur de validation. C'est l'entropie croisée sur le dataset de test. Cette valeur diminue lors de l'underfitting et augmente lors de l'overfitting. L'algorithme d'early stopping sélectionne le modèle à la limite de l'augmentation brusque de l'erreur de validation.

- **Amélioration de genann**

Pour améliorer Genann, nous avons modifié la bibliothèque afin qu'elle puisse prendre en compte les fonctions d'activations ReLU dans les couches cachées, essentielles pour un entraînement rapide et efficace du modèle. En entraînant des modèles avec des ReLU dans les couches cachées, nous avons noté une amélioration notable du temps de training (50% plus rapide sur les meilleurs modèles) avec cependant un plus grand risque d'overfitting. 361 modèles comprenant ReLU avec 1 à 19 couches cachées et de 1 à 19 neurones dans chaque couches cachées ont été entraînés et voici les résultats :



Graphique en 3D donnant le ratio précision au carré divisé par le temps d'entraînement en fonction de hidden layer et de hidden. Les modèles utilisent la fonction ReLU dans les couches cachées comme fonction d'activation avec un learning rate de 0.1

3 Meilleures Précisions	Input Layer	Hidden Layer	Neuron per hidden	Output	RAM (kb)	Flash (kb)	Training Time (s)	Prediction (ms)	Accuracy
	784	4	17	10	465.00	126.00	28.36	509.94	93.10
3 Meilleures précisions en MNIST ANN	784	3	17	10	453.00	123.00	34.47	530.26	92.20
	784	4	15	10	463.00	112.00	15.06	443.48	91.66

Tableau des 3 meilleurs modèles en terme de précision en fonction de hidden\_layer et hidden. On dépasse les 93% de précision (les prédictions se font sur 10000 images test)

3 Meilleures Précisions*2/time	Input Layer	Hidden Layer	Neuron per hidden	Output	RAM (kb)	Flash (kb)	Training Time (s)	Prediction (ms)	Accuracy	Accuracy*2/Time
	784	2	9	10	437.00	69.00	5.74	259.51	85.37	1269.91
3 Meilleures Précisions*2/time en MNIST ANN	784	3	10	10	449.00	76.00	6.58	317.16	88.66	1194.80
	784	2	10	10	437.00	75.00	6.38	286.57	86.40	1170.79

Tableau des 3 meilleurs modèles en terme de précision au carré divisé par le temps d'entraînement en fonction de hidden\_layer et hidden. On dépasse les 94% de précision

Nous voulons un modèle qui a une bonne précision tout en gardant un temps d'entraînement plutôt faible. Le deuxième tableau nous incite à ne garder qu'un modèle avec 2 couches cachées et 9 neurones par couches cachées. C'est typiquement celui-ci qui sera utilisé sur le Rpi aussi bien en entraînement qu'en déploiement. Voici sa matrice confusion:

Matrice de confusion avec 2 couches cachées et 9 neurones pour chacune

	0	1	2	3	4	5	6	7	8	9
0	953	0	24	10	4	23	26	4	9	12
1	1	1111	11	3	4	7	4	15	16	7
2	0	3	866	25	9	5	3	26	9	2
3	3	4	56	868	0	39	0	11	38	4
4	7	0	8	0	874	10	20	6	13	36
5	2	2	1	56	0	698	13	0	38	15
6	9	1	17	5	19	28	879	0	15	1
7	0	4	16	18	5	15	1	922	7	26
8	2	10	31	15	7	49	12	3	822	7
9	3	0	2	10	60	18	0	41	7	899
	0	1	2	3	4	5	6	7	8	9

Vrais

Avec les ANN, nos résultats montrent qu'on ne peut pas dépasser 87% de précision pour un modèle s'entraînant assez vite. Dans la suite, on développe les réseaux de neurones convolutionnels (CNN). Cette nouvelle architecture est spécialement adaptée à la classification d'images et peut dépasser 98% de précision pour des modèles s'entraînant rapidement.

### **Début Avril -> Mi-mai : Implémentation d'un add-on pour les CNN pour GenANN**

- **Implémentation des couches de convolutions**

Avant d'implémenter les couches de convolutions, nous avons commencé par implémenter une série de fonctions permettant d'effectuer des opérations matricielles, comme la création d'une matrice, le calcul de sa trace, ... Ces opérations sont nécessaires à l'implémentation des convolutions. Pour l'implémentation des convolutions : Le code s'occupant de la propagation en avant a été implémenté, et relu. Il ne reste que le code s'occupant de la rétro-propagation.

```
int matrix_new(Matrix* out, size_t width, size_t height);
int matrix_create_output_convolution(Matrix *out, Matrix filter, unsigned int stride, Matrix input);
void matrix_convolution(Matrix *out, const Matrix filter, unsigned int stride, const Matrix input);
double matrix_trace(const Matrix m);
void rotate_matrix_180(const Matrix* mat);
void matrix_free(Matrix);
Matrix* padding_matrix(const Matrix* mat, int pad);
void matrix_scale(const Matrix* mat, double c);
void matrix_add(const Matrix* out, const Matrix rhs);
```

```
Matrix convolution_run(ConvolutionLayer *layer, Matrix input);
void convolution_train(ConvolutionLayer *layer, Matrix *out_bt, Matrix input, Matrix in_bt, double learning_rate);
```

```
Pooling* pooling_init(PoolingType type, size_t pool_size, size_t input_size);
double const* pooling_run(Pooling *pool, double const *inputs);
double* pooling_backpropagate(Pooling *pool, double const *inputs);
```

- **Implémentation des CNN en sequential**

Pour tester les couches de convolutions, nous avons décidé - en nous inspirant de la bibliothèque TensorFlow - de proposer un objet permettant de relier plusieurs couches de réseaux de neurones les un à la suite des autres : Le *Sequential*. Cet objet, en plus de simplifier la création de modèles, nous permettra la répartition sur plusieurs machines des couches d'un modèle donné.

```
model = models.Sequential([
    layers.Flatten(input_shape=(28, 28)),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

### Sequential sur Tensorflow

```
Sequential* sequential_create(size_t input_width, size_t input_height);
void sequential_add_pooling_2d(Sequential *, Pooling *);
void sequential_add_dense(Sequential *, genann const *);

double const *sequential_run(Sequential const *seq, double const *inputs);
void sequential_train(Sequential const *seq, double const *inputs, double const *desired_outputs, double learning_rate);
```

### Nos sequential en C

## **Mi-Mai -> Mi-Juin : Premiers test de CNN et mise en place d'optimisations**

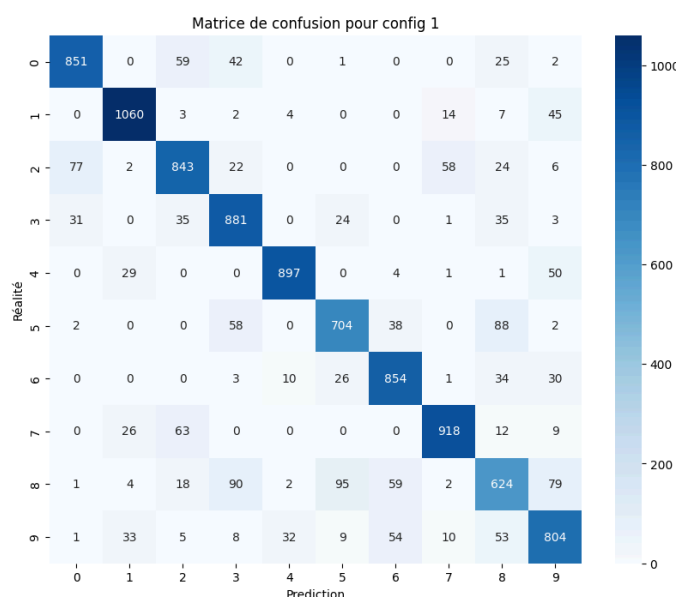
- **Premiers tests des CNN**

Initialement, nos CNN n'avaient qu'un seul filtre par couche de convolution. Voici les résultats:

Config	Conv1 (WxH)	Conv2 (WxH)	Pool1	Pool2	RAM (kb)	flash (kb)	Training Time	Prediction (m)	Accuracy
1	5x5	5x5	2	2	473	393	29.29	193.86	91.83
3	7x7	7x7	2	2	473	456	47.67	407.54	89.27
2	3x3	3x3	2	2	473	296	17.41	138.90	89.04
4	9x9	9x9	2	2	473	582	72.04	413.38	83.95

Tableau de 4 modèles CNN. On dépasse les 91% de précision avec 1 seul filtre par couche de convolution (plus le filtre est grand, plus le modèle est lourd)





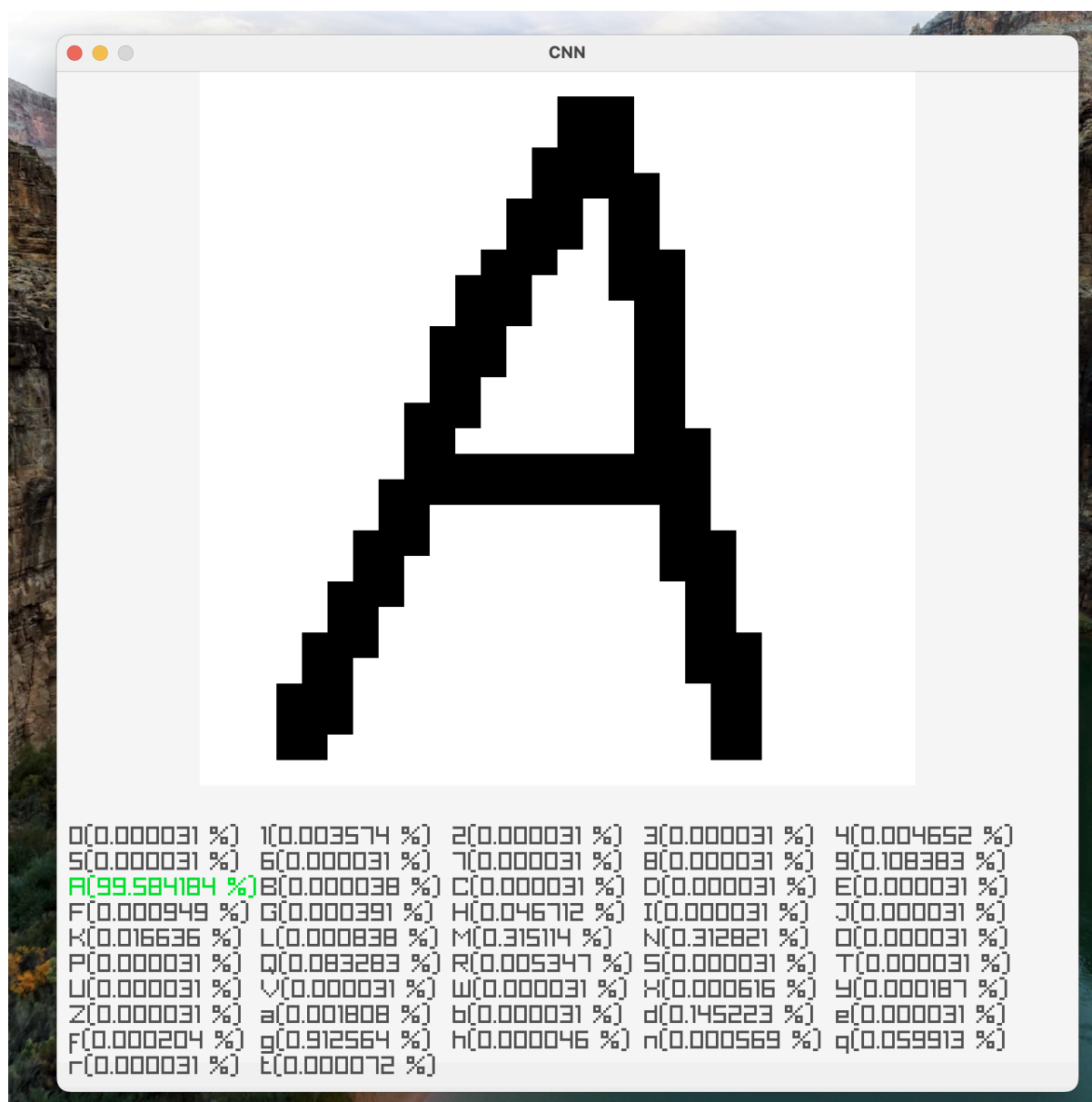
Une première amélioration a été de rajouter la possibilité d'avoir plusieurs filtres par couche de convolution. Cela a rendu le modèle plus lourd, avec un impact très négatif sur le temps d'entraînement. Nous avons ensuite ajouté une option afin d'inclure du bruit sur les données de d'entraînement afin d'avoir de meilleurs résultats en déploiement.

```
c1 = convolution_init(28, 28, 1, 5, 5, /*filters=*/2, 1, 2, 2);
a1 = activation_init((char*)"sigmoid", sigmoid, sigmoid_derivate, 28, 28, 2);
p1 = pooling_init(AVERAGE_POOLING, 2, 28, 2);
c2 = convolution_init(14, 14, 2, 5, 5, /*filters=*/5, 1, 2, 2);
a2 = activation_init((char*)"sigmoid", sigmoid, sigmoid_derivate, 14, 14, 5);
p2 = pooling_init(AVERAGE_POOLING, 2, 14, 5);
fl = flatten_init(7, 7, 5);
ann = genann_init(7*7*5, 2, 600, CLASS_COUNT);
```

### Nos sequential en C (inspiré de LeNet, une architecture CNN)

- **Extension à la base de données EMNIST(extended MNIST)**

Maintenant que nous avons la possibilité de créer de grands modèles avec plusieurs filtres de convolution par couche de convolution, nous avons décidé d'entraîner nos modèles sur EMNIST qui inclut, en plus des chiffres manuscrits, les lettres de l'alphabet manuscrites ce qui fait 112000 images d'entraînement. Très vite, nous avons été limités par la faible capacité de calcul de notre RaspberryPi, ce qui nous a forcé à développer de petits modèles. Pour 23 minutes d'entraînement et 10 générations, notre modèle atteint 85% de bonnes réponses sur les données test.



Capture d'écran de notre démo sur CNN. Ici notre modèle prédit 'A' avec 99,58% de certitude (softmax)

- **Mesure des performances :**

En parallèle de l'implémentation de ces différents programmes, nous avons mis en place des dockers afin de pouvoir mesurer les performances du modèle. Nous ne sommes pas parvenus à avoir une mesure instantanée des performances du système comme prévu au départ (données de RAM, utilisation processeur etc...), mais nous avons obtenu des valeurs moyennées sur le temps d'exécution du modèle. Toutefois, nous avons sans problème pu mesurer les performances de nos modèles, en calculant les matrices de confusions, le temps d'entraînement et les fréquences de prédictions et en les comparant entre modèles.

**Semaine de piscine : 24 -> 27 juin : Finalisation et rendu final**

- **Contribution : Pull request**

Notre test de la librairie sur le dataset MNIST nous a donné des résultats très concluants, permettant de se rendre compte de l'efficacité de la librairie, et de son fonctionnement. Nous avons donc décidé de faire une pull request sur le github de GenANN, afin de contribuer au projet avec une démonstration plus poussée que celle initialement fournie par la librairie : une simple implémentation sur le dataset IRIS.



Capture d'écran de notre pull request à Genann


- **Démonstration finale :**

Nous avons réalisé une page web permettant de reconnaître les signes manuscrits dessinés sur un canevas. Héberger un serveur sur une raspberry pi ayant pour but de mettre en pratique du TinyML semble a priori contre-intuitif : nous l'avons fait dans le but de mettre en évidence un cas d'usage éventuel de notre projet, où cette fois tous les échanges se feraient en local, avec une raspberry connectée à une tablette graphique par exemple. Le TinyML prendrait alors tout son sens, permettant une reconnaissance de l'écriture instantanée, sans avoir besoin de dépendre d'un serveur plus puissant.

De plus, nous aurions souhaité en premier lieu héberger un serveur sur les PC, et que la raspberry s'y connecte. Cela aurait permis de se coller davantage à l'attitude du TinyML, c'est-à-dire d'occuper le moins de place possible sur le module à faible puissance, ici la raspberry, et de superviser la raspberry avec un serveur central. Cependant, après avoir contacté la DSI et pour des raisons de sécurité fixées par Télécom Paris, l'idée est difficilement réalisable sans avoir des contraintes techniques fortes (liées au pare-feu déployé), et ce n'était pas le cœur de notre projet. Finalement, après réflexion, nous avons décidé d'abandonner cette possibilité et d'héberger le serveur sur la raspberry.

Pour mettre en place une application plus réaliste, il suffirait de remplacer le docker réalisant le serveur HTTP par un docker récupérant l'input d'une tablette graphique par

exemple, et communiquant de la même manière avec le serveur faisant tourner le modèle, localement.

```
Docker > Demo finale >  docker-compose.yml
1  version: '3'
2  services:
3    web_server:
4      build: ./web_server
5      ports:
6        - "5000:5000"
7      depends_on:
8        - ml_server
9
10   ml_server:
11     build: ./ml_server
12     ports:
13       - "55565:55565"
```

Le fichier docker-compose.yml permet de build et run facilement les 2 dockers avec la commande *docker compose* tout en routant les ports dockers vers les ports de la raspberry

La démonstration est donc composée de 2 containers, un gérant la page HTTP, les requêtes clients etc, et le second s'occupant de l'exécution du modèle. Ils sont articulés entre eux avec un fichier docker compose, qui permet l'exécution simple et fiable de la démo.

```
Docker > Demo finale > ML > M Makefile
1  # Variables de compilation
2  CC=gcc
3  LD=gcc
4  CFLAGS=-Wall -Wextra -Wno-unused-parameter -Wpedantic -Wshadow \
5      -Wformat=2 -Wwrite-strings -Wstrict-prototypes -Wold-style-definition \
6      -Wredundant-decls -Wnested-externs -Wmissing-include-dirs \
7      -O3 -Isrc -c -std=c11
8  LDFLAGS=-O3 -lm
9  # Dossiers des fichiers sources et objets
10 SRC_DIR = src
11 OBJ_DIR = objets
12 BIN_DIR = bin
13
14 # Fichiers sources explicitement listés
15 SRCS = $(SRC_DIR)/server.c $(SRC_DIR)/sequential.c $(SRC_DIR)/genann.c \
16      $(SRC_DIR)/pooling.c $(SRC_DIR)/convolution.c $(SRC_DIR)/activation.c \
17      $(SRC_DIR)/flatten.c $(SRC_DIR)/tensor.c $(SRC_DIR)/mnist_db.c $(SRC_DIR)/utils.c
18
19 # Fichiers objets
20 OBJS = $(subst $(SRC_DIR)/%.c, $(OBJ_DIR)/%.o, $(SRCS))
21
22 # Nom de l'exécutable
23 EXEC = $(BIN_DIR)/demo_server
24
25 # Règle par défaut
26 all: $(EXEC)
27
28 # Règle pour créer l'exécutable
29 $(EXEC): $(OBJS)
30     @mkdir -p $(BIN_DIR)
31     $(CC) $(OBJS) -o $@ $(LDFLAGS)
32
33 # Règle pour créer les fichiers objets
34 $(OBJ_DIR)/%.o: $(SRC_DIR)/%.c
35     @mkdir -p $(OBJ_DIR)
36     $(CC) $(CFLAGS) -c $< -o $@
37
38 # Règle de nettoyage
39 clean:
40     rm -rf $(OBJ_DIR) $(BIN_DIR)
41
42 .PHONY: all clean
```

Makefile assez complexe nécessaire à la compilation de notre code pour la démonstration finale, il met en valeur la portabilité et la facilité du docker

## Conclusion et Future work

Nous avons donc produit au cours de ce projet un add on pour la librairie de machine learning très légère GenANN, permettant d'utiliser des réseaux de neurones convolutionnels, ainsi qu'un programme implémentant un test de la librairie sur le dataset MNIST. Nous avons fait une **pull request** sur cette implémentation, car nous pensons que cet exemple met en valeur et permet de montrer l'efficacité de GenANN.

Nous avons également produit des images **dockers** permettant d'exécuter nos programmes facilement sur n'importe quelle machine possédant docker, afin de garantir une portabilité avancée tout en évitant la charge liée à une machine virtuelle. Nous avons également implémenté dans le docker de notre démonstration finale une interface réseau, qui nous permet via un serveur HTTP de faire des prédictions de caractères manuscrits en temps réel.

Par la suite, nous espérons pouvoir tirer de notre production des résultats intéressants permettant de publier **un article scientifique** avec notre encadrant, M. Linguaglossa.

Des pistes d'améliorations pourraient être de pouvoir rajouter des classes à la volée en entraînant le modèle en parallèle de faire des prédictions, ainsi que de finir d'implémenter des tests plus demandant tels que le dataset CIFAR. Sur un plan plus théorique nous pourrions analyser notre code C pour déterminer les instructions nécessaires à son exécution et déterminer la complexité d'un microcontrôleur sur-mesure capable de le faire tourner.

## Références :

1. Partha Pratim Ray,  
A review on TinyML: State-of-the-art and prospects,  
Journal of King Saud University - Computer and Information Sciences,  
Volume 34, Issue 4,  
2022,  
Pages 1595-1623,  
ISSN 1319-1578,  
<https://doi.org/10.1016/j.jksuci.2021.11.019>.
2. Norah N. Alajlan and Dina M. Ibrahim  
TinyML: Enabling of Inference Deep Learning Models on Ultra-Low-Power IoT Edge  
Devices for AI Applications  
<https://doi.org/10.3390/mi13060851>
3. Autres : des blogs technologiques, des forums de discussion sur l'IoT et le TinyML...