Rapport mi-projet :

Apprentissage automatique avec tinyML dans les réseaux logiciels

Avancement du projet:

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

- **Etude de la librairie Genann:**

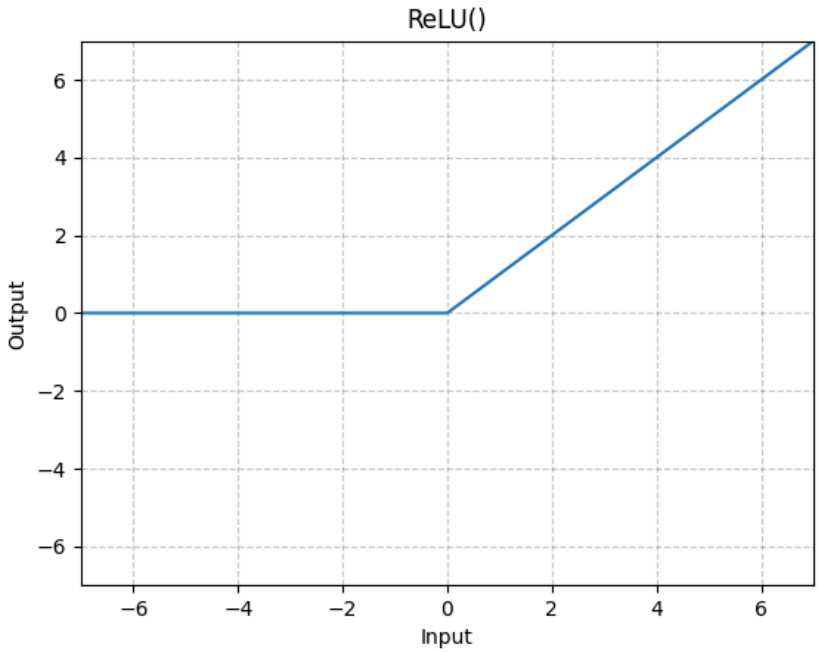
Genann est une bibliothèque logicielle open source conçue pour mettre en œuvre des réseaux de neurones artificiels (ANN), qui sont les réseaux de neurones les plus simples, dans des applications informatiques. Le fichier genann.c implémente une structure genann de ANN et l'utilisateur a le choix du nombre 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 un 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<=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 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.

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

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

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

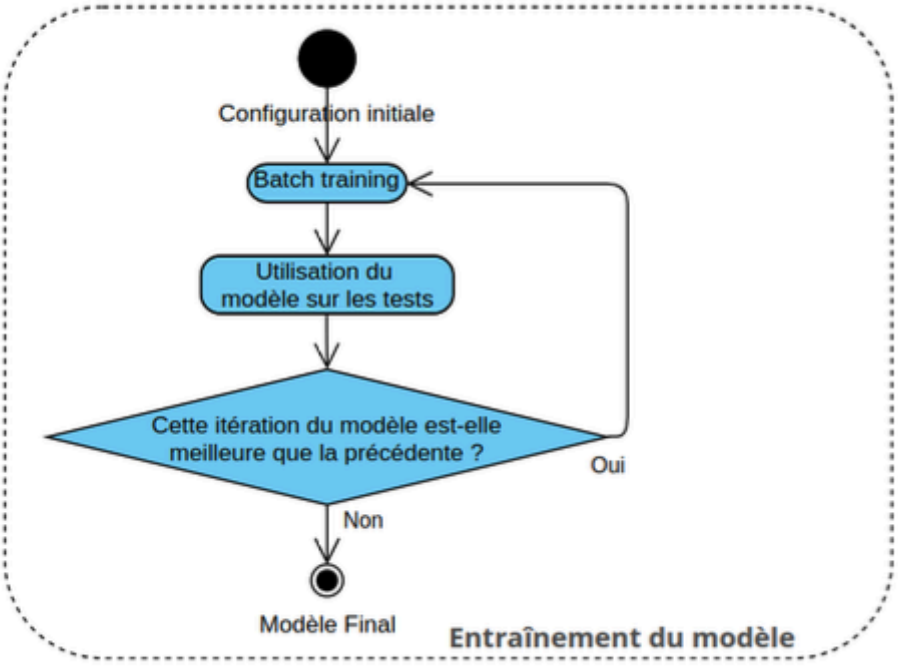
Structures et fonctions utilisées par notre dataloader

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 entraine l'IA sur toute la base de données
    for (i = 0; i < training.nimages; ++i) {
        printf("[Entrainement 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:

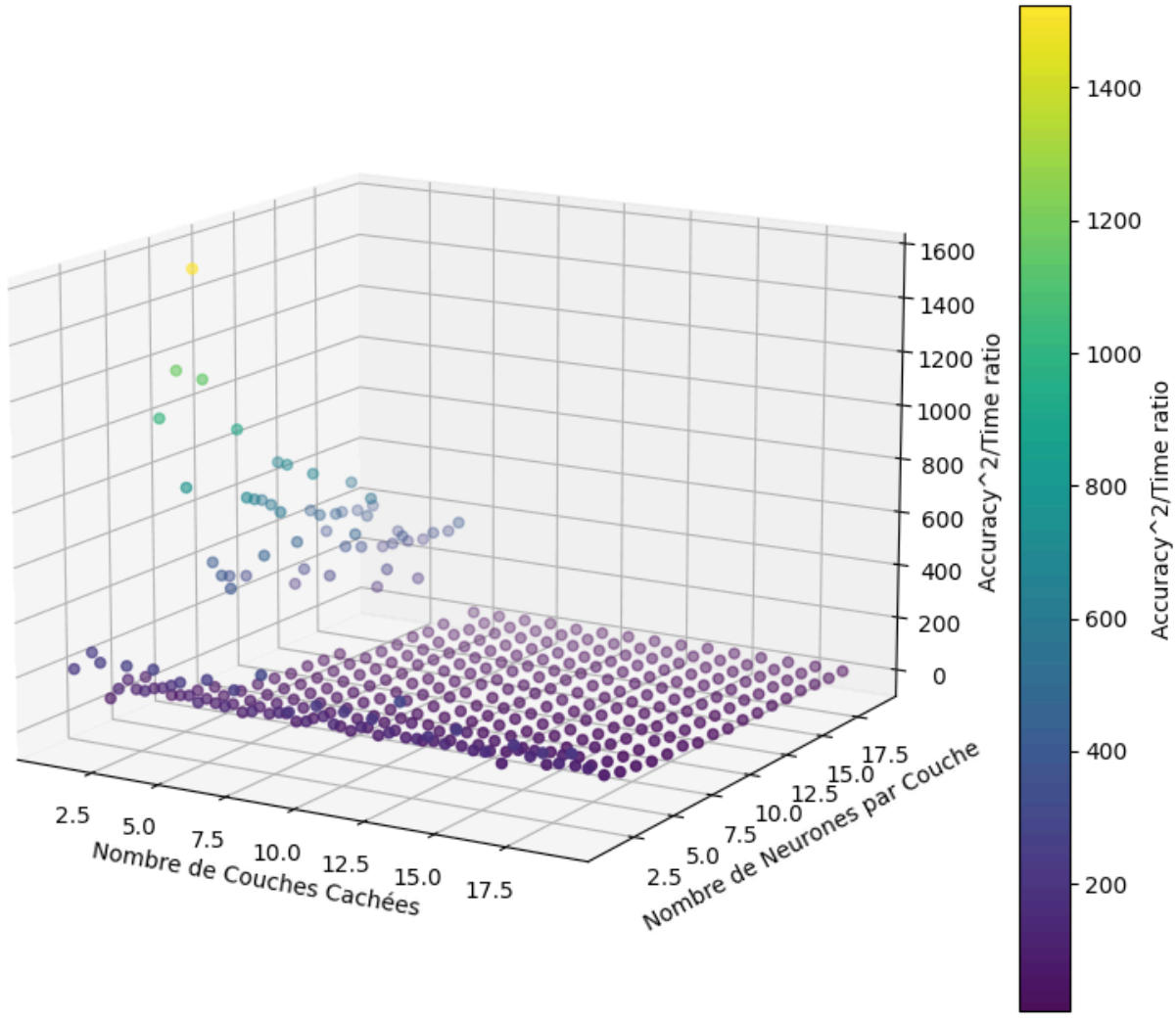
```
# Utilise une image de base Alpine Linux version 3.8
FROM alpine:3.8

# 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 Accuracy^2/Time plutôt que Accuracy/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

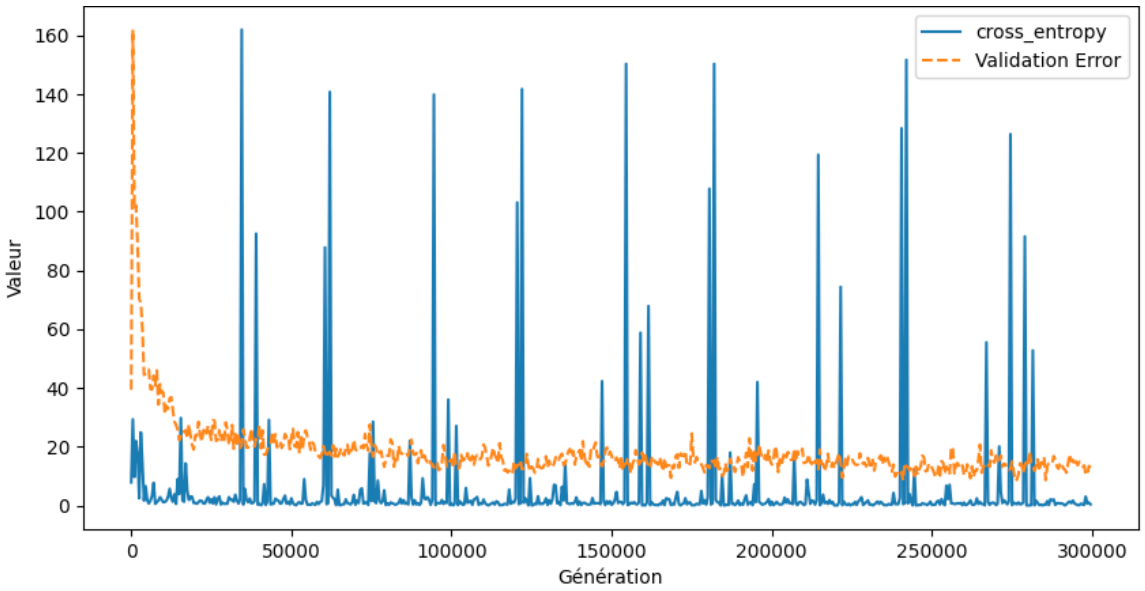
3 Meilleures Précisions	Input Layer	Hidden Layer	Neuron per hidden	Output	RAM (kb)	Flash (kb)	Training Time (s)	Prediction (ms)	Accuracy
	784	2	19	10	443.00	134.00	30.39	571.15	94.26
3 Meilleures précisions en MNIST ANN	784	2	18	10	442.00	127.00	34.48	520.57	94.02
	784	1	19	10	431.00	130.00	23.30	536.69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

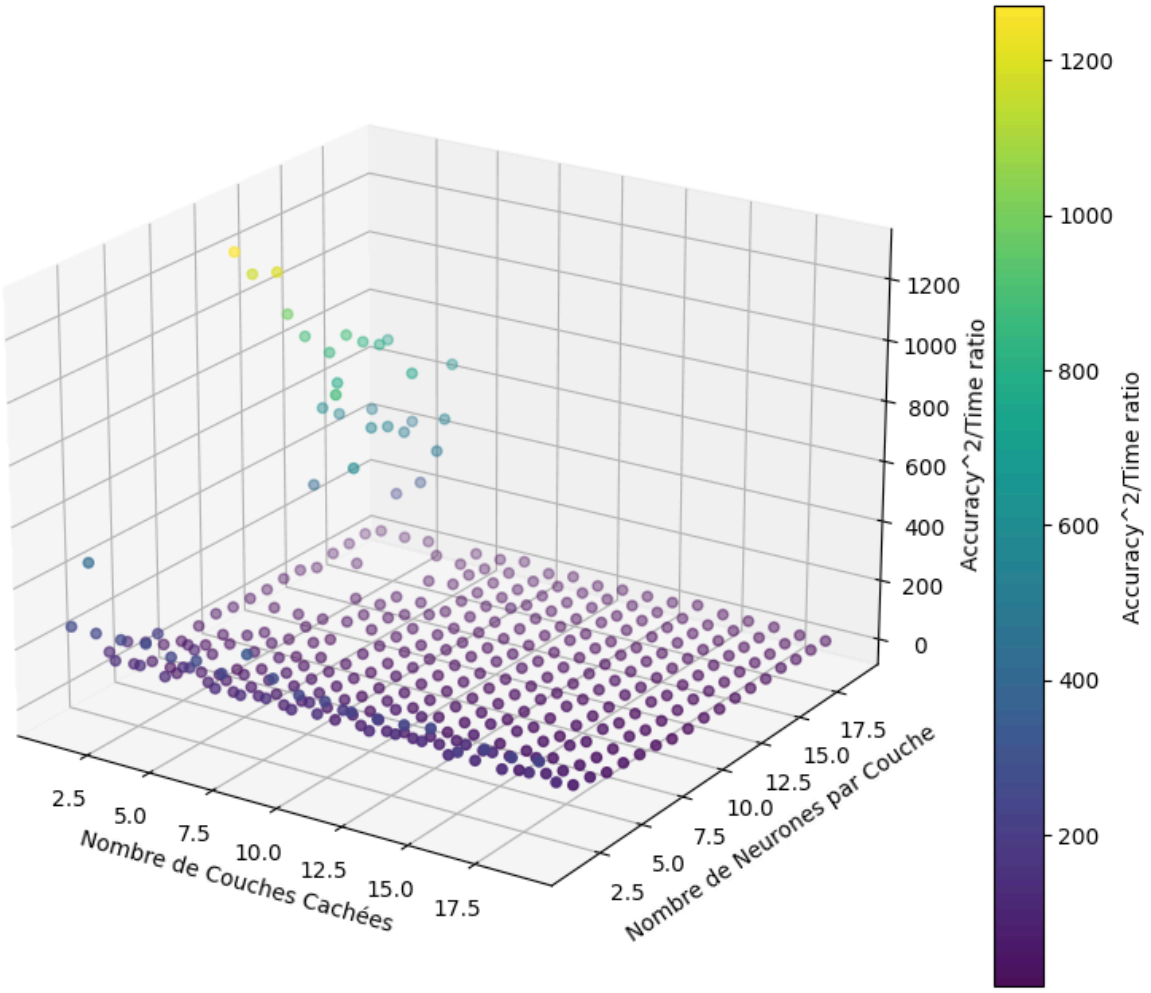
Cette courbe affiche deux variables en fonction du nombre d'images passées au modèle. Ici, le modèle a analysé 300000 images, ce qui donne 6 générations car il y a 60000 images par générations. Les courbes représentent:

-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 30000 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 cachés, 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

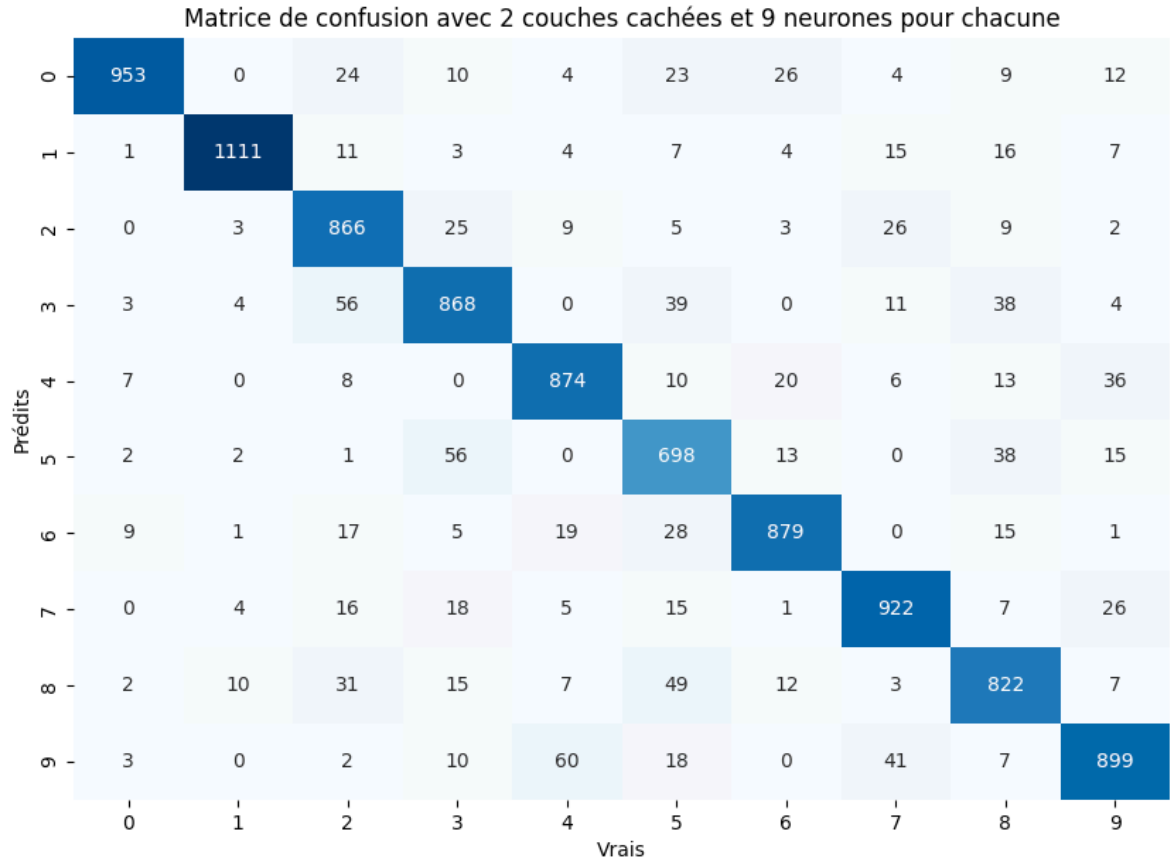
3 Meilleures Précisions	Input Layer	Hidden Layer	Neuron per hidden	Output	RAM (kb)	Flash (kb)	Training Time (s)	Prediction (ms)	Accuracy
	784	4	17	10	465.00	126.00	28.36	509.94	93.10
3 Meilleures précisions en MNIST ANN	784	3	17	10	453.00	123.00	34.47	530.26	92.20
	784	4	15	10	463.00	112.00	15.06	443.48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

Tableau des 3 meilleurs modèles en terme de précision en fonction de hidden_layer et hidden. On dépasse les 93% de précision

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:



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);
```

- **Premiers tests de réseaux de neurones CNN**

Ces tests se baseront sur les bases de données MNIST et CIDAR. 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

```c
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

Pour le moment, les tests n'ont pas encore été conduits, car il reste à :

- Relire le code lié aux couches de *Pooling*
- Relire le code lié aux couches de convolution
- Relire le code lié aux *Sequentials*