Aymane ICHOU
M1 BIG DATA

Rapport de projet : Réseaux de neurones artificiels (RNA)

Sommaire

Aymane ICHOU
M1 BIG DATA
2100 7668

Rapport de projet : Réseaux de neurones artificiels (RNA)

1.Introduction

L'objectif de ce programme est de simuler un réseau de neurones artificiel (RNA) simple, capable d'apprendre à reproduire le comportement des portes logiques AND, OR et XOR.
Le tout est implémenté en C, sans bibliothèque d'intelligence artificielle, afin de comprendre en détail les mécanismes d'un réseau neuronal :

-propagation avant, aussi appelé forward propagation (calcul de la sortie),
-rétropropagation, également appelé backpropagation (ajustement des poids),
-l'entraînement sur plusieurs epochs, appelé train,
-et en améliorant le traitement grâce aux threads (bibliothèque : pthread).

2.Code

2.1 Paramètres et constantes globales

```
// --------- Paramètres du réseau ---------
#define INPUTS 2
#define HIDDEN 3
#define OUTPUTS 1
#define EPOCHS 10000
```

Ces constantes définissent l'architecture du réseau :

INPUTS = 2 : chaque porte logique prend deux entrées (couche entrée).
HIDDEN = 3 : le réseau contient 3 neurones cachés (couche caché).
OUTPUTS = 1 : une seule sortie (couche sortie).
EPOCHS = 10000 : le réseau s'entraîne sur 10 000 itérations pour converger.

Ces valeurs contrôlent la taille et la durée de l'apprentissage.

2.2 Mutex

```
// --------- Mutex pour affichage ---------
pthread_mutex_t print_mutex;
```

Plusieurs RNA sont entrainés en parallèle, cela permet de garder une concordance sur l'accès à la ressource, ainsi qu'à l'affichage. Deux thread ou plus, n'écrivent pas en même temps dans le terminal.

2.3 Fonctions utilitaires.

```c
double sigmoid(double x) {
    return 1.0 / (1.0 + exp(-x));
}
```

La sigmoïde transforme une valeur entre 0 et 1, simulant l'activation d'un neurone.
Elle permet d'introduire une non-linéarité, indispensable pour l'apprentissage.

```c
double d_sigmoid(double y) {
    return y * (1.0 - y);
}
```

La dérivée de la sigmoïde, utilisée dans la rétropropagation/backpropagation, elle permet de calculer l'impact d'une erreur sur chaque neurone.

```c
double random_weight() {
    return ((double)rand() / RAND_MAX) * 2.0 - 1.0;
}
```

Random_weight, permet la génération d'un flottant aléatoire compris entre -1 et 1. Cela est nécessaire pour l'initialisation du réseau et des poids. Sans cela, les neurones apprendraient tous la même chose.

```c
void line() {
    printf("------------------------------\n");
}
```

Fonction d'affichage dans le terminal, permet d'améliorer le rendu dans le terminal.

2.4 Structures.

2.4.1. Structure du réseau de neurones.

```c
// --------- Structure du réseau ---------
typedef struct {
    char name[20];   // Nom du réseau (ex: "AND", "XOR")
    double weight_input_hidden[INPUTS][HIDDEN];
    double bias_hidden[HIDDEN];
    double weight_hidden_output[HIDDEN];
    double bias_output;
    double learning_rate;
} NeuralNetwork;
```

Chaque réseau est une instance de cette structure, contenant :

name : identifiant du réseau (AND, OR ou XOR).
weight_input_hidden : poids entre la couche d'entrée et la couche cachée.
bias_hidden : biais pour chaque neurone caché.
weight_hidden_output : poids entre la couche cachée et la sortie.
bias_output : biais de la sortie.
learning_rate : taux d'apprentissage (ici, arbitrairement, 0.1).

2.4.2. Structure pour les threads.

```c
// --------- Structure pour threads ---------
typedef struct {
    NeuralNetwork *nn;
    double (*inputs)[2];
    double *targets;
    double *epoch_errors; // Tableau pour stocker les erreurs par epoch
} ThreadData;
```

Cette structure permet de passer tous les paramètres nécessaires à un thread :
le réseau, les entrées, les sorties attendues et un tableau pour enregistrer les erreurs à chaque époch.

2.5.Fonctions

2.5.1 Initialisation du réseau.

```
// --------- Initialisation du réseau ---------
void init_network(NeuralNetwork *nn, const char *name, double learning_rate)
```

Cette fonction :

copie le nom du réseau,
initialise tous les poids et biais de manière aléatoire,
fixe le taux d'apprentissage.

Chaque réseau part d'un état aléatoire différent.

2.5.2 Propagation avant / Forward propagation.

```
double forward_propagation(NeuralNetwork *nn, double input1, double input2, double hidden[HIDDEN])
```

Cette étape calcule la sortie du réseau pour une paire d'entrées :

Chaque neurone caché combine les entrées pondérées et applique la sigmoïde.
La sortie combine les activations cachées avec leurs poids et biais, puis applique à nouveau la sigmoïde.

Cela simule le passage de l'information de gauche à droite dans le réseau.

2.5.3 Rétro-propagation / BackPropagation.

```
void backpropagation(NeuralNetwork *nn, double input1, double input2,
                     double hidden[HIDDEN], double output, double target)
```

C'est la phase d'apprentissage :

Erreur globale : error = target - output

Erreur de sortie : dérivée de la sigmoïde × erreur

Erreur cachée : propage l'erreur de la sortie vers chaque neurone caché

Mise à jour des poids et biais proportionnelle à l'erreur et au taux d'apprentissage.

Cette boucle interne ajuste petit à petit le réseau pour minimiser l'erreur quadratique moyenne.

2.5.4. Entrainement / Train parallèle

```
// --------- Entraînement ---------
void *train(void *arg) {
```

Chaque thread exécute cette fonction :

Pour chaque epoch :

- Il calcule la sortie du réseau sur les 4 exemples d'apprentissage (0,0), (0,1), (1,0), (1,1).
- Il rétropropage l'erreur pour corriger les poids.
- Il enregistre l'erreur totale de l'époque.

Chaque porte logique est donc entraînée indépendamment, en parallèle avec les autres.

2.5.4. Test et affichage des résultats

```
// --------- Test et affichage ---------
void test_and_print(NeuralNetwork *nn, double inputs[4][2], double *epoch_errors)
```

Une fois l'entraînement terminé :

- Le programme affiche les résultats finaux pour chaque combinaison d'entrée.
- Puis il affiche l'évolution de l'erreur toutes les 2000 epochs.
- Le mutex est utilisé ici pour que les sorties des threads ne se mélangent pas à l'écran.

2.5.4. Main / Fonction principale.

C'est le cœur du programme, qui orchestre toutes les étapes :
-Initialisation du générateur aléatoire et du mutex.
-Définition des ensembles d'apprentissage pour AND, OR et XOR.
-Création d'un tableau de 3 réseaux de neurones (nb_gate = 3).
-Initialisation des structures de données et des threads.
-Lancement de trois threads d'entraînement en parallèle.
-Synchronisation avec pthread_join.
-Affichage des performances finales pour chaque porte logique.
-Destruction du mutex et fin du programme.

3. Conclusion

Ce programme illustre les fondements d’un réseau de neurones artificiel :

- la propagation d’une information à travers plusieurs couches,
- l’apprentissage supervisé par rétropropagation,
- et l’exécution parallèle grâce aux threads.

Les trois portes logiques sont apprises indépendamment, mais à partir du même code général.