TP5 : Les Réseaux de neurones (2h)

Le but du TP est de mettre en place un réseau de neurones simple afin de se familiariser avec ces modèles et la façon de les entrainer : la backpropagation (ou rétropropagation) du gradient. Pour cela, nous commencerons par etudier le perceptron simple et sa procédure d'apprentissage. On implémentera ce réseau avec la librairie **NeuralNetwork** pour le tester sur l'arduino Due pour verfier son bon fonctionnemen, puis on calcule le temps et la précesion de l'algorithme backpropagation.

Exercice 1 –Perceptron simple

Un perceptron est un type de réseau neuronal artificiel (RNA) qui peut être implémenté sur une carte Arduino pour des tâches telles que la reconnaissance de motifs, la classification et le contrôle. Voici les étapes de base pour implémenter un perceptron sur une carte Arduino:

1. **Installation de la bibliothèque NeuralNetwork version : 1.7.9:**
   - Ouvrez l'IDE Arduino et cliquez sur Croquis > Inclure une bibliothèque > Gérer les bibliothèques.
   - Recherchez " **NeuralNetwork** " et cliquez sur Installer.

2. **Création de réseaux de neurones NeuralNetwork:**
   - Ouvrez l'IDE Arduino et cliquez sur Fichier > Nouveau .
   - Copiez et collez le code ci-dessous dans le nouvel onglet et complétez-le par suite:

```
#include<Arduino.h>
#define NumberOf(arg) ((unsigned int) (sizeof (arg) / sizeof
(arg [0])))
#define SELU// Uncomment this line to use SELU Activation
Function(faster!)
const unsigned int layers[] = {2,4,1};
//3 layers (1st)layer with 2 input neurons
//(2nd)layer with 4 hidden neurons
//(3rd)layer with 1 output neuron
```

3. **Définir les entrées et les sorties :**
   - Définir le nombre d'entrées (le nombre de capteurs ou de caractéristiques) et de sorties (le nombre de classes ou d'actions) pour votre tâche spécifique.

4. **Initialiser le perceptron :**
   - Définir le nombre de neurones dans la couche cachée (le cas échéant).

```
NeuralNetwork NN(layers,NumberOf(layers));
```

   - Définir les poids et les biais pour chaque connexion entre les neurones (de manière aléatoire ou en utilisant des valeurs pré-entraînées).
   -
   - Définir le taux d'apprentissage et la fonction d'activation (par exemple, sigmoïde ou ReLU).

```
#define SELU // Uncomment this line to use SELU Activation
Function     (faster!)
#define _1_OPTIMIZE B00010000
```

   - Par exemple, vous pouvez initialiser un perceptron avec un neurone dans la couche cachée, deux entrées, une sortie et des poids et des biais aléatoires comme suit :

5. **Former le perceptron :**
   - Alimenter les entrées dans le perceptron et calculer les sorties prédites.
   - Comparer les sorties prédites avec les sorties attendues et calculer l'erreur.
   - Rétropropager l'erreur à travers le réseau et ajuster les poids et les biais en utilisant le taux d'apprentissage.
   - Répéter le processus pendant plusieurs itérations ou jusqu'à ce que l'erreur soit minimisée.
   - Par exemple, vous pouvez former le perceptron pour classer les objets en fonction de leur couleur comme suit :

```
float input_data[4][2] = { {0.0, 0.0}, {0.0, 1.0}, {1.0, 0.0},
  {1.0, 1.0}};
const float input_data[4][2] = {
  {0, 0}, //0
  {0, 1}, //1
  {1, 0}, //1
  {1, 1}  //0
};
float output_data[4][1] = {{0.0},{1.0},{1.0},{0.0}};
.
.
.
 do{
```

```
    for (int j = 0; j < NumberOf(inputs); j++)
// 1 epoch
    {
       NN.FeedForward(inputs[j]);
// Feeds-Forward the inputs to the first layer of the NN and
Gets the output.
       NN.BackProp(expectedOutput[j]);
// Tells to the NN if the output was right/the-expectedOutput
and then, teaches it.
    }

    // Prints the Error.
    Serial.print("MSE: ");
    Serial.println(NN.MeanSqrdError,6);

    // loops through each epoch Until MSE goes < 0.001
  }while(NN.GetMeanSqrdError(NumberOf(inputs)) > 0.001);
```

6. Test du perceptron :

- Cliquez sur Croquis > Vérifier/Compiler pour compiler le code.

```
#include<Arduino.h>
#include <NeuralNetwork.h>
#define NumberOf(arg) ((unsigned int) (sizeof (arg) / sizeof
(arg [0]))) //calculates the amount of layers (in this case 3)
#define _1_OPTIMIZE B00010000 // REDUCING RAM By using the same
pointer for every layer's weights.
#define SELU                  // Uncomment this line to use
SELU Activation Function     (faster!)
const unsigned int layers[] = {2,4,1}; //3 layers (1st)layer
with 2 input neurons (2nd)layer with 4 hidden neurons and
(3rd)layer with 1 output neuron
float *output_data; // 3rd layer's outputs (in this case
output)
```

```
//Default Inputs/Training-Data
const float input_data[4][2] = {
  {0, 0}, //0
  {0, 1}, //1
  {1, 0}, //1
  {1, 1}  //0
};

const float expectedOutput[4][1] = {{0},{1},{1},{0}}; // values that we were expecting to get f²rom the 3rd/(output)layer of Neural-network, in other words something like a feedback to the Neural-network.
NeuralNetwork NN(layers,NumberOf(layers));
void setup()
{
  Serial.begin(115200);
  // Creating a NeuralNetwork with default learning-rates
  do{
    for (int j = 0; j < NumberOf(input_data); j++) // 1 epoch
    {
       NN.FeedForward(input_data[j]);      // Feeds-Forward the inputs to the first layer of the NN and Gets the output.
       NN.BackProp(expectedOutput[j]); // Tells to the NN if the output was right/the-expectedOutput and then, teaches it.
    }
    // Prints the Error.
    Serial.print("MSE: ");
    Serial.println(NN.MeanSqrdError,6);
    // loops through each epoch Until MSE goes < 0.001
  }while(NN.GetMeanSqrdError(NumberOf(input_data)) > 0.001);
Serial.println("\n =-[OUTPUTS]-=");
}
void loop() {
  //Goes through all inputs
```

```
  //NeuralNetwork NN(layers,NumberOf(layers));
  for (int i = 0; i < NumberOf(input_data); i++)
  {
    output_data = NN.FeedForward(input_data[i]); // Feeds-
Forward the inputs[i] to the first layer of the NN and Gets the
output
    Serial.println(output_data[0], 7);       // prints the
first 7 digits after the comma.
  }

   NN.print(); // prints the weights and biases of each layer
}
```

- Cliquez sur Croquis > Téléverser pour téléverser le code sur la carte Arduino.
- Ouvrez la console série en cliquant sur Outils > Moniteur Série.
- Ouvrez la console série en cliquant sur Outils > Moniteur Série.
- Le perceptron effectuera une série d'itérations pour entraîner le réseau neuronal à résoudre le problème XOR.
- Les entrées et les sorties calculées pour chaque combinaison de sortie XOR seront affichées dans le Moniteur Série.

Q.1 Comment sont initialisés les poids du réseau de neurones ?

Q.2 Comment la méthode backPropagate ajuste-t-elle les poids et les biais du réseau ?

Q.3 Pourquoi est-il important d'utiliser une fonction d'activation dans un réseau de neurones ?

Q.4 Comment peut-on étendre cet exemple pour résoudre des problèmes plus complexes ?

Q.5 Comment le réseau de neurones est-il configuré pour résoudre le problème XOR ?

Q.6 Comment les entrées et les sorties attendues sont-elles stockées ?

Q.7 Comment le réseau de neurones est entraîné pour résoudre le problème XOR ?

Q.8 Comment la sortie pour de nouvelles entrées peut-elle être calculée après l'entraînement ?

Exercice 2 – Perceptron multicouche

A) Additionneur 1bit à 1bit :

Le principe de fonctionnement de l'additionneur bit à bit par perceptron multicouche est similaire à celui d'un additionneur classique, à la différence près qu'il utilise un réseau de neurones pour effectuer les calculs.

Le réseau de neurones est constitué de plusieurs couches de neurones interconnectés. Les neurones dans la couche d'entrée reçoivent les bits des nombres à ajouter en entrée. Les neurones dans la couche cachée effectuent des calculs sur ces bits en utilisant des poids spécifiques pour chaque connexion entre les neurones. Les neurones dans la couche de sortie produisent les bits de la somme résultante.

Le calcul effectué par chaque neurone est basé sur la somme pondérée des entrées, suivie d'une fonction d'activation qui détermine la sortie du neurone. La fonction d'activation la plus couramment utilisée pour les réseaux de neurones est la fonction sigmoïde, qui produit une sortie comprise entre 0 et 1. Pour réaliser cet additionneur, nous suivons les étapes suivantes:

1. **Conception du perceptron :**

- La couche d'entrée est composée de trois neurones qui représentent les bits d'entrée A et B et la retenue Cin

```
unsigned int layers[] = {3,5,3,2};
// 3 layers (1st)layer with 3 input neurons
//(2nd)layer 5 hidden neurons each
//(3th)layer 3 hidden neurons each
//(4th)layer with 2 output neuron
```

- La couche cachée est composée de plusieurs neurones qui utilisent une fonction d'activation non linéaire pour calculer des fonctions complexes des entrées.

```
#define ACTIVATION__PER_LAYER
```

- La couche de sortie est composée de deux neurones qui représentent les bits de sortie S et de retenue Cout.

```
float *outputs[2]; // 4th layer's outputs (in this case output)
*output[0]-> addition results, *output[1]-> carry
```

- Chaque neurone est connecté à tous les neurones de la couche suivante.
- Les poids de chaque connexion sont choisis pour que le réseau calcule la somme des bits d'entrée A et B ainsi que le bit de retenue Cin, et que les neurones de sortie calculent les bits de sortie S et Cout.
- Les biais de chaque neurone sont choisis pour qu'ils produisent les valeurs binaires 0 ou 1 en fonction de leurs entrées pondérées.
- La fonction d'activation de la couche cachée est une fonction sigmoïde ou tangent hyperbolique qui permet au réseau de modéliser des fonctions non linéaires. Vous pouvez prédéfinir les fonctions d'activation pour chaque couche du MLP par l'ajout du numéro de la fonction d'activation :

```
byte Actv_Functions[] = {   1, 1, 0};
// 1 = Tanh and 0 = Sigmoid
```

- La fonction d'activation de la couche de sortie est aussi une fonction sigmoïde qui renvoie des probabilités normalisées pour chaque bit de sortie.
- L'erreur de sortie est calculée à partir de la différence entre la sortie réelle et la sortie attendue pour chaque exemple d'entraînement, et les poids sont ajustés en fonction de cette erreur en utilisant un algorithme d'apprentissage tel que la rétropropagation du gradient.

```
const float inputs[8][3] = {
  {0, 0, 0}, //0
  {0, 0, 1}, //1
  {0, 1, 0}, //1
  {0, 1, 1}, //0
  {1, 0, 0}, //1
  {1, 0, 1}, //0
  {1, 1, 0}, //0
```

```
  {1, 1, 1}  //1
};
const float expectedOutput[8][2] = {{0,0}, {1,0}, {1,0}, {0,1}, {1,0},
{0,1}, {0,1}, {1,1}};
) *expectedOutput[0]-> addition results, **expectedOutput[1]-> carry
```

Une fois que le MLP est entraîné, il peut être utilisé pour effectuer des additions bit à bit en alimentant des bits d'entrée A et B et en obtenant les bits de sortie S et Cout. Cette architecture de réseau de neurones est plus complexe que celle du réseau de neurones binaire, mais elle permet une plus grande flexibilité et une meilleure précision dans la modélisation des fonctions d'entrée-sortie complexes.

2. **Entraînement des perceptrons :**

- Une boucle est utilisée pour exécuter le processus d'entraînement pendant un certain nombre d'itérations. Dans cette boucle, les données d'entrée sont présentées à chaque réseau de neurones, et les poids et les biais du réseau de neurones sont ajustés en fonction de la différence entre la sortie calculée et la sortie attendue. La précision est contrôlée par :

```
NN.GetMeanSqrdError(NumberOf(inputs)) > 0.01
```

- Pour chaque combinaison d'entrée, la méthode FeedForward est utilisée pour calculer la sortie. La méthode backProp est ensuite utilisée pour ajuster les poids et les biais en fonction de la différence entre la sortie calculée et la sortie attendue.

```
for (int j = 0; j < NumberOf(inputs); j++)
   {
     NN.FeedForward(inputs[j]);
    NN.BackProp(expectedOutput[j]);
   }
NN.print();
```

```
Sortie   Moniteur série ×
Message (Enter to send message to 'Arduino Due (Programming Port)' on 'COM8')
2  W: 1.4888716  W: 1.2349588  W: 1.4048699
3  W:-1.7464341  W:-1.4147676  W: 0.0908575
4  W:-0.4876901  W:-0.6316606  W:-2.0208418
5  W: 1.1934689  W: 0.5883750  W: 0.9151610
----------------------
5 3| bias:1.00
1  W:-3.9742181  W:-3.2772853  W: 10.9015017  W: 9.7702332  W:-3.1901855
2  W:-1.5881205  W:-0.7863392  W: 5.9952884  W: 5.4659424  W:-1.4865783
3  W:-2.4401910  W:-3.7204137  W: 4.6846142  W: 3.8973672  W:-2.4536710
----------------------
3 2| bias:2.29
1  W: 8.7951422  W:-7.0956531  W:-8.8719816
2  W:-9.1529112  W: 2.2958915  W:-3.4927156
----------------------
```

3. Test des perceptrons :

Une fois l'entraînement terminé, Les résultats sont affichés dans la console série, en affichant l'entrée, la sortie calculée pour le MLP :

```
for (int i = 0; i < NumberOf(inputs); i++)
  {
   outputs[i] = NN.FeedForward(inputs[i]);
   Serial.println(outputs[i][0], 7);
   Serial.println(outputs[i][1], 7);
  }
```

```
Sortie   Moniteur série ×
Message (Enter to send message to 'Arduino Due (Programming Port)' on 'COM8')
1 = 0.9654479
1 ≅ 0.9358917

 =-[OUTPUTS]-=
0 ≅ 0.0008191
0 ≅ 0.0165206
0 ≅ 0.0274237
1 ≅ 0.9650390
0 ≅ 0.0147105
1 ≅ 0.9654226
1 ≅ 0.9654479
1 ≅ 0.9358917
```

Le code du cette exemple est fourni ci-dessous et vous pouvez le telecharger de boostcamp:

```
#include<Arduino.h>
#include <NeuralNetwork.h>
```

```
#define NumberOf(arg) ((unsigned int) (sizeof (arg) / sizeof (arg
[0])))
#define _1_OPTIMIZE B00010000
#define ACTIVATION__PER_LAYER
unsigned int layers[] = {3, 5, 3, 2};
byte Actv_Functions[] = {   1, 1, 0};
// 1 = Tanh and 0 = Sigmoid
//  Create the new Neural Network
NeuralNetwork NN(layers, NumberOf(layers), Actv_Functions);
float *outputs[2];
//Default Inputs/Training-Data
const float inputs[8][3] = {
  {0, 0, 0}, //0
  {0, 0, 1}, //1
  {0, 1, 0}, //1
  {0, 1, 1}, //0
  {1, 0, 0}, //1
  {1, 0, 1}, //0
  {1, 1, 0}, //0
  {1, 1, 1}  //1
};
const float expectedOutput[8][2] = {
  {0,0},{1,0},{1,0},{0,1},{1,0},{0,1},{0,1},{1,1}};
 void setup()
{
  Serial.begin(115200);
   do{ for (int j=0; j < NumberOf(inputs); j++)
       { NN.FeedForward(inputs[j]);
         NN.BackProp(expectedOutput[j]);
       }
    Serial.print("J cretirion Error: ");// Prints the Error.
    Serial.println(NN.MeanSqrdError,2);
  }while(NN.GetMeanSqrdError(NumberOf(inputs)) > 0.01);
   NN.print();
```

```
}
void loop() {
  Serial.println("\n =-[OUTPUTS]-=");
  for (int i=0; i < NumberOf(inputs); i++)
  { outputs[i] = NN.FeedForward(inputs[i]);
    Serial.print(round(outputs[i][1]));
    Serial.print(" ≅ ");
    Serial.println(outputs[i][1], 7);
  }
}
```

Q.8 Copier et enregistrer le fichier avec un nom de votre choix sur votre PC. Vérifier et télé verser le code sur la carte Arduino due. Executer le code et copier les valeurs afficher dans le moniteur série dans un fichier texte.

Q.9 Augmenter le nombre de réseaux dans les 2 couches caches 7, 5, entrainer de nouveau le MLP et afficher les poids de la couche de sortie du MLP. Comparer les poids du nouveau MLP avec les poids de l'ancien MLP

Q.10 Calculer la sortie de votre MLP pour l'additionneur bit à bit et comparer aux valeur de l'ancien réseaux avec les couches `{3, 5, 3, 2}`.

Q.11 Augmenter la précision de calcul de votre réseau dans le cas des couches intermédiaires 7,5 à 0.001. Entrainer le MLP et calculer de nouveau les valeurs des poids et des sorties. Commenter les résultats obtenus ?.

Q.12 Modifier les fonctions d'activation pour le MLP dans le cas des couches intermédiaires 7, 5 à 0.001. Considérer que les fonctions d'activation sont des sigmoïdes. Entrainer le MLP et calculer de nouveau les valeurs des poids et des sorties. Quel est l'impact du choix de la fonction d'activation sur les valeurs de sortie du réseau.

B) Multiplexeur 4x1 neuronal :

Un multiplexeur 4x1 est un circuit logique qui permet de sélectionner l'un des quatre signaux d'entrée pour être transmis à la sortie en fonction des signaux de commande. Il peut être

implémenté à l'aide d'un MLP en utilisant une architecture avec une couche d'entrée, une ou deux couche cachée et une couche de sortie. Pour Réaliser le Multiplexeur, nous vous demandons de suivre les mêmes étapes que l'additionneur à un bit et répondre aux questions suivantes

Q.13 Copier et enregistrer le dernier fichier avec un autre nom de votre choix sur votre PC. Vérifier le code sur la carte Arduino due. Pour réaliser la fonction de multiplexage, modifier les anciennes valeurs des variables et remplir les variables suivantes avec les nouvelles valeurs :

```
float …outputs[…];
const float inputs[…][…] = {……
};
const float expectedOutput[…][…] = {……};
```

Q.14 Fixer le nombre de couches à 3 avec 4 entrées 2 entrées select et une sortie, entrainer le MLP et afficher les poids de la couche de sortie du MLP.

Q.15 Calculer la sortie de votre MLP pour le Multiplexeur dans cas de la configuration des couches suivante :`{6, 7, 3, 1}`.

Q.16 Augmenter la précision de calcul de votre réseau dans le cas des couches intermédiaires 9,2 à 0.001. Entrainer le MLP et calculer de nouveau les valeurs des poids et des sorties. Commenter les résultats obtenus ?.