TP6 : Apprentissage Profond (2h)

Le but du TP est de mettre en place un réseau de neurones reccurent (RNN) et convolutionnels (CNN) simple afin de se familiariser avec ces modèles simples d'apprentissage profond et la façon de les entrainer : la backpropagation (ou rétropropagation) du gradient. .

Exercice 1 –RNN simple pour Arduino

Sur un arduino, nous désirons utiliser une architecture de réseau de neurones récurrents (RNN) pour prédire une séquence de sortie en fonction d'une séquence d'entrée. Le RNN est composé de trois couches de neurones : une couche d'entrée, une couche cachée et une couche de sortie. Les poids et les biais de chaque couche sont initialisés aléatoirement à l'aide de la fonction d'initialisation des poids. Les sorties sont calculés par l'algorithme de propagation avant de l'entrée à travers le réseau pour calculer la sortie en utilisant la fonction d'activation sigmoid. Le seuil de prédiction de sortie est donné par la valeur lu du d'une fonction valpot et ajuste le seuil de prédiction en conséquence. La prédiction est initialisé par la séquence d'entrée avec des valeurs aléatoires et démarre la séquence de prédiction. Dans la boucle loop(), la séquence d'entrée est avancée à chaque itération et la séquence de sortie est prédite. La valeur prédite est affichée sur la LED interne de l'arduino Due en fonction du seuil de prédiction. Pour réaliser notre programme de prédiction, nous utilisons les variables et les fonctions ci-dessous :

- const int inputSize = 5; représente le nombre de neurones d'entrée dans le réseau de neurones.
- const int hiddenSize = 10 : représente le nombre de neurones cachés dans le réseau de neurones.
- const int outputSize = 1 : Cette ligne définit une constante entière appelée "outputSize" qui représente le nombre de neurones de sortie dans le réseau de neurones.
- const int sequenceLength = 20 : représente la longueur de la séquence à prédire dans le réseau de neurones.
- const int thresholdPin = A0 : représente le numéro de broche pour le potentiomètre utilisé pour régler le seuil de prédiction.
- const int buttonPin = 2 : représente le numéro de broche pour le bouton de démarrage de la prédiction.
-
- const int ledPin = 13 : représente le numéro de broche pour la LED qui affichera les résultats de la prédiction.

- float inputs[inputSize]; représente la séquence d'entrée pour le réseau de neurones.
- float outputs[outputSize] : représente la séquence de sortie pour le réseau de neurones.
- float hidden[hiddenSize] : un tableau de nombres flottants qui représente l'état caché pour le réseau de neurones.
- float weightsIH[hiddenSize][inputSize] : un tableau bidimensionnel de nombres flottants représente les poids entre les neurones d'entrée et les neurones cachés.
- float weightsHH[hiddenSize][hiddenSize] : un tableau bidimensionnel de nombres flottants appelé "weightsHH" qui représente les poids entre les neurones cachés et les autres neurones cachés.
- float weightsHO[outputSize][hiddenSize]: un tableau bidimensionnel de nombres flottants appelé "weightsHO" qui représente les poids entre les neurones cachés et les neurones de sortie.
-
- float biasH[hiddenSize]: Cette ligne déclare un tableau de nombres flottants appelé "biasH" qui représente les biais pour les neurones cachés.
- float biasO[outputSize] : Cette ligne déclare un tableau de nombres flottants appelé "biasO" qui représente les biais pour les neurones de sortie.
- float threshold = 0.5: Cette ligne déclare une variable flottante appelée "threshold" qui représente le seuil de prédiction initial.bool predict = false

La fonction **initializeWeights()** est utilisée pour initialiser les poids et les biais du réseau de neurones. Elle utilise des nombres aléatoires pour initialiser les poids et les biais, qui sont stockés dans les tableaux weightsIH, weightsHH, weightsHO, biasH et biasO.

```
void initializeWeights() {
  // Initialize input-hidden weights
  for (int i = 0; i < hiddenSize; i++) {
    for (int j = 0; j < inputSize; j++) {weightsIH[i][j] = random(-100, 100) / 100.0;}
  }
  // Initialize hidden-hidden weights
  for (int i = 0; i < hiddenSize; i++) {
    for (int j = 0; j < hiddenSize; j++) {weightsHH[i][j] = random(-100, 100) / 100.0;}
  }
  // Initialize hidden-output weights
  for (int i = 0; i < outputSize; i++) {
    for (int j = 0; j < hiddenSize; j++) {weightsHO[i][j] = random(-100, 100) / 100.0;}
  }
  // Initialize biases
  for (int i = 0; i < hiddenSize; i++) {biasH[i] = random(-100, 100) / 100.0;}
for(int i = 0; i < outputSize; i++) {biasO[i] = random(-100, 100) / 100.0;}
}
```

La fonction **forwardPass( )** effectue une propagation avant à travers le réseau de neurones. Elle prend en entrée les valeurs d'entrée, de sortie et cachées, et calcule les valeurs cachées

et de sortie à partir des poids et des biais stockés. Elle utilise la fonction d'activation **activation()** pour calculer les valeurs cachées et de sortie. :

```
void forwardPass(float* input, float* output, float* hidden) {
// Compute hidden state
for (int i = 0; i < hiddenSize; i++) {
hidden[i] = 0;
for (int j = 0; j < inputSize; j++) {hidden[i] += weightsIH[i][j] * input[j];}
for (int j = 0; j < hiddenSize; j++) {hidden[i] += weightsHH[i][j] * hidden[j];}
hidden[i] += biasH[i];
hidden[i] = activation(hidden[i]);
}
// Compute output
for (int i = 0; i < outputSize; i++) {
output[i] = 0;
for (int j = 0; j < hiddenSize; j++) {output[i] += weightsHO[i][j] * hidden[j];}
output[i] += biasO[i];
output[i] = activation(output[i]);
}
}
```

La fonction **activation()** est une fonction d'activation sigmoid utilisée pour calculer les valeurs cachées et de sortie.

```
float activation(float x) {
// Sigmoid activation function
return 1.0 / (1.0 + exp(-x));
}
```

La fonction **readPoten()** lit la valeur fixe. Le seuil de prédiction est utilisé pour déterminer si la valeur de prédiction doit être considérée comme étant "activée" ou non.

```
void readPoten() {
// Read potentiometer value and map to prediction threshold
int potValue = 2.5;
}
```

La fonction startPrediction() initialise la séquence d'entrée avec des valeurs aléatoires et définit le drapeau de prédiction sur vrai pour indiquer que la séquence de prédiction doit être exécutée.

```
void startPrediction() {
// Initialize input sequence with random values
for (int i = 0; i < sequenceLength; i++) {
inputs[i] = random(0, 10) / 10.0;
}

// Set flag to start prediction sequence
predict = true;
}
```

La fonction setup()configure les broches pour le bouton, la LED et le port série. Elle initialise également les poids et les biais du réseau de neurones en appelant **initializeWeights()**.

```
void setup() {
  // Setup pins
  Serial.begin(115200);
  pinMode(buttonPin, INPUT_PULLUP);
  pinMode(ledPin, OUTPUT);

  // Initialize weights and biases
  initializeWeights();
}
```

La fonction loop() est la boucle principale du programme. Elle lit la valeur du poten et ajuste le seuil de prédiction en appelant readPoten(). Si la prédiction est activée, elle exécute une propagation avant en appelant forwardPass(), et utilise la valeur de sortie pour allumer ou éteindre la LED en fonction du seuil de prédiction. Enfin, elle décale la séquence d'entrée d'une position et met à jour la première valeur avec la valeur de sortie pour préparer la prochaine itération de la boucle.

```
void loop() {
  readPoten();
    startPrediction();
    // Perform forward pass if prediction sequence is started
  if (predict) {
    forwardPass(inputs, outputs, hidden);
    // Output predicted value on LED display
    if (outputs[0] > threshold) {digitalWrite(ledPin, HIGH);}
    else {
      digitalWrite(ledPin, LOW);
    }
    // Shift input sequence
    for (int i = sequenceLength - 1; i > 0; i--) {inputs[i] = inputs[i - 1]; }
    inputs[0] = outputs[0];
  }
}
```

Le code utilise la fonction d'activation sigmoïde pour les neurones, et il y a des poids et des biais générés aléatoirement lors de l'initialisation. Le réseau prend en entrée une séquence de longueur 20, et le programme utilise une boucle pour mettre à jour la séquence d'entrée avec les sorties du réseau lorsqu'il effectue une prédiction.

Q.1 Quel est le rôle de la fonction initializeWeights ?

Q.2 Comment est calculé l'état caché (hidden state) dans la fonction forwardPass ?

Q.3 Quel est l'avantage de l'utilisation de la fonction d'activation sigmoid pour calculer les sorties (output) du réseau de neurones ?

Q.4 À quoi sert la fonction readPoten et comment est-elle utilisée pour ajuster le seuil de prédiction (prediction threshold) ?

Q.5 À quoi sert la fonction startprediction et comment est-elle utilisée pour calculer la prédiction ?

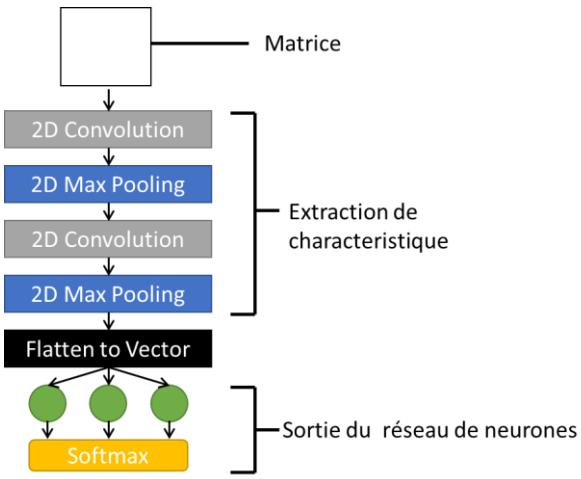
Q.6 Expliquer comment le RNN calcule la sortie avec la fonction forwardPass et comment on peut utiliser pour étendre le nombre des cellules RNN ?

Exercice 2 –CNN sur Arduino Due

L'objectif de cette partie est entraîner un modèle CNN pour la classification d'une matrice carrée de 9x9 en suivant les étapes suivantes :

1. Charger la matrice.
2. Prétraiter la matrice en la normalisant et en la redimensionnant si nécessaire.
3. Définir l'architecture du modèle CNN en utilisant des couches de convolution, de pooling et de couches entièrement connectées.
4. Compiler le modèle sur l'arduino Due.
5. Évaluer la performance du modèle en mesurant le temps d'entrainement.
6. Analyser les résultats et ajuster les hyperparamètres du modèle pour améliorer les performances si nécessaires.
7. Tester le modèle sur des matrices modifier.

Pour construire le réseau CNN nous suivons l'algorigramme sur la figure ci-dessous :



Sur l'arduino vous implémentez les fonctions nécessaires pour produire pour produire un réseau CNN.

la fonction de convolution 2D:

```
void convolution2D(float input[INPUT_SIZE][INPUT_SIZE],float kernel[KERNEL_SIZE][KERNEL_SIZE],float
output[OUTPUT_SIZE][OUTPUT_SIZE],float bias)
```

La fonction de convolution 2D qui prend quatre arguments : une matrice d'entrée input, une matrice de noyau kernel, une matrice de sortie output et un biais bias. Les trois premiers arguments sont des tableaux à deux dimensions de flottants (floats) et le dernier argument est un flottant.

Le noyau parcours toutes les lignes et les colonnes de la matrice output. Le noyau est appliqué à la région centrée sur le pixel de la matrice de sortie, en prenant en compte le décalage (padding) et le pas (stride). La somme pondérée des éléments de la région d'entrée est calculée en multipliant chaque élément par son poids correspondant dans le noyau, puis en sommant les résultats. Enfin, le biais est ajouté à la sortie de chaque pixel pour obtenir la valeur finale de la matrice de sortie. La fonction Convolution D2 est écrite en code arduino comme suit :

```
void convolution2D(float input[INPUT_SIZE][INPUT_SIZE],float kernel[KERNEL_SIZE][KERNEL_SIZE],float
output[OUTPUT_SIZE][OUTPUT_SIZE],float bias)
{
 for(int i = 0; i < OUTPUT_SIZE; i++) {
  for(int j = 0; j < OUTPUT_SIZE; j++) {
   output[i][j] = 0;
   for(int k = 0; k < KERNEL_SIZE; k++) {
    for(int l = 0; l < KERNEL_SIZE; l++) {
     output[i][j] += input[i*STRIDE_SIZE + k - PADDING_SIZE][j*STRIDE_SIZE + l - PADDING_SIZE] * kernel[k][l];
    }
   }
   output[i][j] += bias;
  }
 }
}
```

- **Max-Pooling 2D**

La deuxième fonction appelé "maxPooling" effectue une opération de max-pooling sur une matrice d'entrée "poolinput" de taille OUTPUT_SIZE x OUTPUT_SIZE.

```
void maxPooling(float poolinput[OUTPUT_SIZE][OUTPUT_SIZE],float
pool[OUTPUT_SIZE/POOL_SIZE][OUTPUT_SIZE/POOL_SIZE])
```

L'opération de max-pooling est une opération de sous-échantillonnage souvent utilisée dans les réseaux de neurones convolutionnels (CNN) pour réduire la taille spatiale de la représentation de l'image.

. La fonction maxPooling est écrite en code arduino comme suit :

```
void maxPooling(float poolinput[OUTPUT_SIZE][OUTPUT_SIZE],float
pool[OUTPUT_SIZE/POOL_SIZE][OUTPUT_SIZE/POOL_SIZE])
{
 for(int i = 0; i < OUTPUT_SIZE/POOL_SIZE; i++) {
  for(int j = 0; j < OUTPUT_SIZE/POOL_SIZE; j++) {
   float maxVal = -INFINITY;
   for(int k = 0; k < POOL_SIZE; k++) {
    for(int l = 0; l < POOL_SIZE; l++) {
     maxVal = max(maxVal, poolinput[i*POOL_SIZE + k][j*POOL_SIZE + l]);
    }
   }
   pool[i][j] = maxVal;
  }
 }
}
```

La fonction prend également en entrée une matrice "pool" de taille OUTPUT_SIZE/POOL_SIZE x OUTPUT_SIZE/POOL_SIZE qui va contenir les valeurs maximales calculées pour chaque région de taille POOL_SIZE x POOL_SIZE de la matrice d'entrée. Le paramètre "-INFINITY" est utilisé comme valeur initiale de la variable "maxVal" pour s'assurer que la première valeur comparée sera toujours plus grande que "-INFINITY". Ensuite, la fonction "max" est appelée à chaque itération pour comparer la valeur actuelle avec la valeur maximale précédente, et stocker la plus grande valeur dans "maxVal". La fonction "maxPooling" retourne la matrice de sortie "pool" qui contient les valeurs maximales calculées pour chaque région de la matrice d'entrée.

- **Flatten To Vector**

La fonction "flatten2vector" pour transformer une matrice 2D de taille OUTPUT_SIZE/POOL_SIZE x OUTPUT_SIZE/POOL_SIZE en un vecteur 1D de taille (OUTPUT_SIZE/POOL_SIZE) * (OUTPUT_SIZE/POOL_SIZE) * 1.

```
void flatten2vector(float flattened[(OUTPUT_SIZE/POOL_SIZE)*(OUTPUT_SIZE/POOL_SIZE)*1],float
pool[OUTPUT_SIZE/POOL_SIZE][OUTPUT_SIZE/POOL_SIZE])
```

Le code arduino ci-dessous vous permet de réaliser cette fonction :

```
void flatten2vector(float flattened[(OUTPUT_SIZE/POOL_SIZE)*(OUTPUT_SIZE/POOL_SIZE)*1],float
pool[OUTPUT_SIZE/POOL_SIZE][OUTPUT_SIZE/POOL_SIZE])
{
 int idx = 0;
 for(int i = 0; i < OUTPUT_SIZE/POOL_SIZE; i++) {
  for(int j = 0; j < OUTPUT_SIZE/POOL_SIZE; j++) {
   flattened[idx++] = pool[i][j];
  }
 }
}
```

La fonction prend en entrée une matrice "pool" de taille OUTPUT_SIZE/POOL_SIZE x OUTPUT_SIZE/POOL_SIZE qui contient les valeurs maximales calculées pour chaque région de

taille POOL_SIZE x POOL_SIZE de la matrice d'entrée. Elle retourne un vecteur 1D "flattened" qui contient les valeurs de la matrice "pool" dans un ordre séquentiel.

Pour chaque élément de la matrice "pool", la fonction "flatten" stocke la valeur correspondante dans le vecteur "flattened" à l'indice "idx", puis incrémente "idx" pour pointer vers l'indice suivant dans le vecteur. La boucle interne est répétée pour chaque élément de la matrice, jusqu'à ce que toutes les valeurs de la matrice aient été stockées dans le vecteur "flattened".

La fonction "flatten" retourne le vecteur "flattened" qui contient toutes les valeurs de la matrice "pool" dans un ordre séquentiel. Cette opération de mise à plat est généralement utilisée pour préparer les données d'entrée pour une couche de neurones entièrement connectée dans un réseau de neurones.

Le code arduino ci-dessous donné ci-dessous, explique le fonctionnement de cette première partie :

```
#include<Arduino.h>
#include <math.h>
#define INPUT_SIZE 9 // input image size
#define KERNEL_SIZE 3 // kernel size
#define PADDING_SIZE 1 // padding size
#define STRIDE_SIZE 1 // stride size
#define POOL_SIZE 2 // max-pooling size
#define OUTPUT_SIZE ((INPUT_SIZE - KERNEL_SIZE + (2 * PADDING_SIZE)) / STRIDE_SIZE + 1)
//
void convolution2D(float input[INPUT_SIZE][INPUT_SIZE],float kernel[KERNEL_SIZE][KERNEL_SIZE],float
output[OUTPUT_SIZE][OUTPUT_SIZE],float bias)
{
 for(int i = 0; i < OUTPUT_SIZE; i++) {
  for(int j = 0; j < OUTPUT_SIZE; j++) {
   output[i][j] = 0;
   for(int k = 0; k < KERNEL_SIZE; k++) {
    for(int l = 0; l < KERNEL_SIZE; l++) {
     output[i][j] += input[i*STRIDE_SIZE + k - PADDING_SIZE][j*STRIDE_SIZE + l - PADDING_SIZE] * kernel[k][l];
    }
   }
   output[i][j] += bias;
  }
 }
}
//
void maxPooling(float poolinput[OUTPUT_SIZE][OUTPUT_SIZE],float
pool[OUTPUT_SIZE/POOL_SIZE][OUTPUT_SIZE/POOL_SIZE])
{
 for(int i = 0; i < OUTPUT_SIZE/POOL_SIZE; i++) {
  for(int j = 0; j < OUTPUT_SIZE/POOL_SIZE; j++) {
   float maxVal = -INFINITY;
   for(int k = 0; k < POOL_SIZE; k++) {
    for(int l = 0; l < POOL_SIZE; l++) {
```

```
        maxVal = max(maxVal, poolinput[i*POOL_SIZE + k][j*POOL_SIZE + l]);
       }
      }
     pool[i][j] = maxVal;
    }
  }
}
void flatten2vector(float flattened[(OUTPUT_SIZE/POOL_SIZE)*(OUTPUT_SIZE/POOL_SIZE)*1],float
pool[OUTPUT_SIZE/POOL_SIZE][OUTPUT_SIZE/POOL_SIZE])
{
 int idx = 0;
 for(int i = 0; i < OUTPUT_SIZE/POOL_SIZE; i++) {
  for(int j = 0; j < OUTPUT_SIZE/POOL_SIZE; j++) {
   flattened[idx++] = pool[i][j];
  }
 }
}
//
void setup() {
// initialize input, kernel, and bias
float einput[INPUT_SIZE][INPUT_SIZE]={
 {0,0,0,0,0,0,0,0,0},
 {0,1,0,0,0,0,0,1,0},
 {0,0,1,0,0,0,1,0,0},
 {0,0,0,1,0,1,0,0,0},
 {0,0,0,0,1,0,0,0,0},
 {0,0,0,1,0,1,0,0,0},
 {0,0,1,0,0,0,1,0,0},
 {0,1,0,0,0,0,0,1,0},
 {0,0,0,0,0,0,0,0,0}
};

float ekernel[KERNEL_SIZE][KERNEL_SIZE]=
{{0,1,0},
 {1,1,1},
 {0,1,1}};

float eoutput[OUTPUT_SIZE][OUTPUT_SIZE];
float epool[OUTPUT_SIZE/POOL_SIZE][OUTPUT_SIZE/POOL_SIZE];
float eflattened[(OUTPUT_SIZE/POOL_SIZE)*(OUTPUT_SIZE/POOL_SIZE)*1];
float ebias(0.);
convolution2D(einput,ekernel,eoutput,ebias);
maxPooling(eoutput,epool);
flatten2vector(eflattened,epool) ;
}
void loop() {
}
```

Q.7 À quoi servent les constantes INPUT_SIZE, KERNEL_SIZE, PADDING_SIZE, STRIDE_SIZE et POOL_SIZE ?

Q8. Que fait la fonction **convolution2D()** et quels sont les arguments qu'elle prend en entrée ?

Q.9 Que fait la fonction **maxPooling()** et quels sont les arguments qu'elle prend en entrée ?. Préciser les dimensions des matrices d'entrée et de sortie.

Q.10 Que fait la fonction **flatten2vector()** et quels sont les arguments qu'elle prend en entrée ? Préciser les dimensions de la matrice d'entrée et la taille du vecteur de sortie

Q.11 Ajouter une fonction **Printflatten2vector()** en en code arduino pour afficher la taille du vecteur.

Nous avons modifié le code pour y inclure un modèle de réseau de neurones. Pour cela, nous avons ajouté les lignes de code ci-dessous dans le setup():

```
unsigned int layers[] = {NumberOf(eflattened), 6, 3, 1};
byte Actv_Functions[] = {   1, 1, 0};
float expectedOutput[NumberOf(eflattened)][1];
 NeuralNetwork NN(layers, NumberOf(layers), Actv_Functions);
 do{
   for (int j=0; j < NumberOf(eflattened); j++)
   {
    NN.FeedForward(eflattened[j]);
    NN.BackProp(expectedOutput[j]);
   }
   Serial.print("J cretirion Error: ");// Prints the Error.
   Serial.println(NN.MeanSqrdError,4);
 }while(NN.GetMeanSqrdError(NumberOf(eflattened)) > 0.0001);
```

Q.12 Peut-on définir le vecteur **expectedOutput** . Si oui comment vous pouvez le générer ?.

Q.13 Est-il toujours possible d'appliquer **NN.BackProp** ?.

Q.14 Executer le code arduino ci-dessous pour générer une sortie de CNN

```
#include<Arduino.h>
#include <NeuralNetwork.h>
#include <math.h>
#define INPUT_SIZE 9 // input image size
#define KERNEL_SIZE 3 // kernel size
#define PADDING_SIZE 1 // padding size
#define STRIDE_SIZE 1 // stride size
#define POOL_SIZE 2 // max-pooling size
#define OUTPUT_SIZE ((INPUT_SIZE - KERNEL_SIZE + (2 * PADDING_SIZE)) / STRIDE_SIZE + 1)
#define NumberOf(arg) ((unsigned int) (sizeof (arg) / sizeof (arg [0])))
#define _1_OPTIMIZE B00010000
#define ACTIVATION__PER_LAYER
float *outputs;
```

```
void convolution2D(float input[INPUT_SIZE][INPUT_SIZE],float
kernel[KERNEL_SIZE][KERNEL_SIZE],float output[OUTPUT_SIZE][OUTPUT_SIZE],float bias)
{
 for(int i = 0; i < OUTPUT_SIZE; i++) {
  for(int j = 0; j < OUTPUT_SIZE; j++) {
   output[i][j] = 0;
   for(int k = 0; k < KERNEL_SIZE; k++) {
    for(int l = 0; l < KERNEL_SIZE; l++) {
     output[i][j] += input[i*STRIDE_SIZE + k - PADDING_SIZE][j*STRIDE_SIZE + l -
PADDING_SIZE] * kernel[k][l];
    }
   }
   output[i][j] += bias;
  }
 }
}
void maxPooling(float poolinput[OUTPUT_SIZE][OUTPUT_SIZE],float
pool[OUTPUT_SIZE/POOL_SIZE][OUTPUT_SIZE/POOL_SIZE])
{
 for(int i = 0; i < OUTPUT_SIZE/POOL_SIZE; i++) {
  for(int j = 0; j < OUTPUT_SIZE/POOL_SIZE; j++) {
   float maxVal = -INFINITY;
   for(int k = 0; k < POOL_SIZE; k++) {
    for(int l = 0; l < POOL_SIZE; l++) {
     maxVal = max(maxVal, poolinput[i*POOL_SIZE + k][j*POOL_SIZE + l]);
    }
   }
   pool[i][j] = maxVal;
  }
 }
}
void flatten2vector(float
flattened[(OUTPUT_SIZE/POOL_SIZE)*(OUTPUT_SIZE/POOL_SIZE)][1],float
pool[OUTPUT_SIZE/POOL_SIZE][OUTPUT_SIZE/POOL_SIZE])
{
 int idx = 0;
 for(int i = 0; i < OUTPUT_SIZE/POOL_SIZE; i++) {
  for(int j = 0; j < OUTPUT_SIZE/POOL_SIZE; j++) {
   flattened[idx++][1] = pool[i][j];
  }
 }
}
//
void setup() {

// initialize input, kernel, and bias
float einput[INPUT_SIZE][INPUT_SIZE]={
 {0,0,0,0,0,0,0,0,0,0},
 {0,1,0,0,0,0,0,1,0},
 {0,0,1,0,0,0,1,0,0},
 {0,0,0,1,0,1,0,0,0},
 {0,0,0,0,1,0,0,0,0},
 {0,0,0,1,0,1,0,0,0},
 {0,0,1,0,0,0,1,0,0},
 {0,1,0,0,0,0,0,1,0},
```

```
 {0,0,0,0,0,0,0,0,0}
};

float ekernel[KERNEL_SIZE][KERNEL_SIZE]=
{{0,1,0},
 {1,1,1},
 {0,1,1}};
float eoutput[OUTPUT_SIZE][OUTPUT_SIZE];
float epool[OUTPUT_SIZE/POOL_SIZE][OUTPUT_SIZE/POOL_SIZE];
float eflattened[(OUTPUT_SIZE/POOL_SIZE)*(OUTPUT_SIZE/POOL_SIZE)][1];
float ebias(0.);
convolution2D(einput,ekernel,eoutput,ebias);
maxPooling(eoutput,epool);
flatten2vector(eflattened,epool) ;

unsigned int layers[] = {NumberOf(eflattened), 6, 3, 1};
byte Actv_Functions[] = {   1, 1, 0};
float expectedOutput[NumberOf(eflattened)][1];


 NeuralNetwork NN(layers, NumberOf(layers), Actv_Functions);
 do{
   for (int j=0; j < NumberOf(eflattened); j++)
   {
     NN.FeedForward(eflattened[j]);
     //NN.BackProp(expectedOutput[j]);
   }
   Serial.print("J cretirion Error: ");// Prints the Error.
   Serial.println(NN.MeanSqrdError,4);
 }while(NN.GetMeanSqrdError(NumberOf(eflattened)) > 0.003);

}

void loop() {
}
```

Q.14 Ajoutez une deuxième couche à votre CNN (Convolution 2D et Max-pooling) et exécutez à nouveau le code Arduino. N'oubliez pas d'ajouter des matrices de taille appropriée pour la deuxième couche et assurez-vous que le nouveau CNN génère un vecteur "flatten" de taille plus petit.