

---

# RAPPORT DE TP

TP 6 : Apprentissage Profond

---

**Auteurs :**

Thomas DUCLOS  
Hugo VIDAL  
Sarah ABERGEL

**Enseignant :**

M. DERRAZ



Nous attestons que ce travail est original, qu'il est le fruit d'un travail commun au trinôme et qu'il a été rédigé de manière autonome.

Paris, le 05/04/2023

# Table des matières

<b>Introduction</b>	<b>1</b>
0.1 Liste des abréviations . . . . .	1
<b>1 RNN simple pour Arduino</b>	<b>2</b>
1.1 But de l'exercice . . . . .	2
1.2 Réponse à l'exercice . . . . .	2
<b>2 CNN sur Arduino Due</b>	<b>4</b>
2.1 But de l'exercice . . . . .	4
2.2 Réponse à l'exercice . . . . .	4
<b>A Annexes</b>	<b>9</b>
A.1 Code d'un RNN . . . . .	9
A.2 Code d'un CNN . . . . .	13

# Introduction

Le but de ce TP est de mettre en place un réseau de neurones récurrent (RNN) et convolutionnel (CNN) simple afin de se familiariser avec ces modèles d'apprentissage profond et la façon de les entraîner : la backpropagation du gradient.

## Liste des abréviations

- **RNN** : Réseau de Neurones Récurrents (Recurrent Neural Network en anglais)
- **CNN** : Réseau de Neurones Convolutifs (Convolutional Neural Network en anglais)
- **NN** : Réseau de Neurones (Neural Network en anglais)

# Exercice I - RNN simple pour Arduino

## A - But de l'exercice

Le but de l'exercice est de simuler un 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.

Le code de cet exercice est fourni en annexe. [Code de l'exercice](#).

## B - Réponse à l'exercice

### 1. Quel est le rôle de la fonction `initializeWeights` ?

Le rôle de la fonction `initializeWeights` est d'initialiser les valeurs des poids et des biais contenues dans les différentes matrices **weightsIH**, **weightsHH**, **weightsHO**, **biasH** et **biasO**. Les valeurs utilisées pour initialiser les différentes valeurs sont des valeurs aléatoires comprises entre 0 et 1.

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

La fonction `forwardPass` calcule l'état caché en effectuant une série d'opérations de calculs sur les entrées fournies dans le tableau d'entrées ainsi que sur les différents biais et poids du réseaux stockés globalement.

Plus précisément, la fonction calcule l'état caché en effectuant les opérations suivantes :

- (a) Initialiser les valeurs de l'état caché à 0.
- (b) Pour chaque neurone d'entrée, multiplier la valeur de l'entrée correspondante par le poids correspondant entre l'entrée et le neurone caché.
- (c) Pour chaque neurone caché, multiplier la valeur de son état actuel par le poids correspondant entre ce neurone et le neurone caché considéré.
- (d) Ajouter la somme de ces produits pondérés à la valeur du biais du neurone caché considéré.
- (e) Appliquer une fonction d'activation à la somme pondérée obtenue pour obtenir la valeur finale de l'état caché pour le neurone considéré.

Ces étapes sont effectuées pour chaque neurone caché dans le réseau, ce qui permet de calculer l'état caché complet du réseau de neurones.

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

Les avantages de l'utilisation de la fonction **sigmoid** en tant que fonction d'activation est qu'elle possède une plage de sortie entre 0 et 1, ce qui est particulièrement adapté pour des problèmes de classification binaire, notre cas d'étude nécessite une sortie binaire 0 ou 1 pour l'état de la LED.

4. **A quoi sert la fonction readPoten et comment est-elle utilisée pour ajuster le seuil de prédiction ?**

La fonction **readPoten** permet de lire la valeur du potentiomètre. Par la suite elle map cette valeur entre 0 et 1 et set le treshold à cette valeur.

Le treshold permet de considérer si un output est une valeur positive ou négative. Dans notre cas positive représente allumer la LED et négative représente éteindre la LED.

5. **A quoi sert la fonction startPrediction et comment est-elle utilisée pour calculer la prédiction ?**

La fonction **startPrediction** initialise les valeurs du tableau d'entrée entre 0 et 1. Par la suite, elle met la valeur de la variable **predict** à la valeur **true**. Cette valeur de la variable **predict** permet d'indiquer à la boucle **loop** du programme de commencer la prédiction du réseau sur le tableau d'entrée.

En résumé cette fonction possède à la fois un rôle d'initialisation mais également d'initiatrice du réseau de neurone.

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

Le **RNN** utilise la fonction **forwardPass** pour calculer la **sortie** du réseau en fonction de l'**état caché** et des **entrées**. Cette fonction prend en entrée un **tableau d'entrée**, un **tableau de sortie** et un **tableau d'état caché**. La fonction calcule l'**état caché** en utilisant les **entrées** et l'**état caché** de la couche précédente, en utilisant les **poids** et les **biais**, puis passe cette somme pondérée à une **fonction d'activation** pour obtenir l'état caché de la cellule. Ceci est répété pour chaque cellule de la couche cachée.

Ensuite la fonction **forwardPass** calcule la séquence de sortie en bouclant sur les **cellules de sortie** du **RNN**. Pour chaque cellule de sortie, la fonction calcule la **somme pondérée** des états cachés de toutes les cellules RNN en utilisant les **poids weightsHO** et le biais **biasO**. Cette somme pondérée est ensuite passée à la fonction d'activation pour **obtenir la sortie** de la cellule.

Pour étendre le nombre de **cellules RNN** dans le réseau, il faut ajouter des cellules RNN supplémentaires à la couche cachée. Cela implique d'augmenter la taille des tableaux **hidden**, **weightsHH**. Il faut également modifier la fonction **forwardPass** pour prendre en compte ces nouvelles cellules.

# Exercice II - CNN sur Arduino Due

## A - But de l'exercice

L'objectif de cette partie est d'entraîner un modèle de réseau de neurones convolutifs (CNN) pour la classification d'une matrice carrée de  $9 \times 9$ . Pour ce faire, vous allez suivre 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 des couches entièrement connectées.
4. Compiler le modèle sur l'Arduino Due.
5. Évaluer la performance du modèle en mesurant le temps d'entraînement.
6. Analyser les résultats et ajuster les hyperparamètres du modèle pour améliorer les performances si nécessaire.
7. Tester le modèle sur des matrices modifiées.

En suivant ces étapes, vous serez en mesure de construire un modèle CNN efficace pour la classification de matrices carrées de  $9 \times 9$ , ce qui peut avoir de nombreuses applications dans des domaines tels que la vision par ordinateur et le traitement de données.

Le code de cet exercice peut être retrouvé en annexe. [Code de l'exercice](#).

## B - Réponse à l'exercice

1. **A quoi servent les constantes `INPUT_SIZE`, `KERNEL_SIZE`, `PADDING_SIZE`, `STRIDE_SIZE` et `POOL_SIZE` ?**

Les constantes `INPUT_SIZE`, `KERNEL_SIZE`, `PADDING_SIZE`, `STRIDE_SIZE` et `POOL_SIZE` sont utilisées pour définir les paramètres du réseau de neurones CNN. Plus particulièrement les constantes représentent :

- **`INPUT_SIZE`** : représente la taille de la matrice d'entrée, ici  $9 \times 9$  pixels.
- **`KERNEL_SIZE`** : représente la taille du noyau de convolution utilisé pour filtrer la matrice, ici  $3 \times 3$  pixels.
- **`PADDING_SIZE`** : représente la taille de remplissage ajoutée autour de la matrice, ici 1 pixel.
- **`STRIDE_SIZE`** : représente la taille du pas de la fenêtre de convolution lors du glissement de l'image d'entrée, ici 1 pixel.
- **`POOL_SIZE`** : représente la taille la fenêtre utilisée pour effectuer le max-pooling sur la sortie de la convolution, ici  $2 \times 2$  pixels.

Ces paramètres sont utilisées dans les différentes fonctions afin de traiter la matrice d'entrée et produire une sortie.

2. **Que fait la fonction `convolution2D` et quels sont les arguments qu'elle prend en entrée ?**

La fonction `convolution2D` effectue une convolution 2D entre l'image d'entrée (définie par la matrice `input`), le noyau (défini par la matrice `kernel`) et un biais (défini par le scalaire `bias`), et stocke le résultat de la convolution dans la matrice de sortie `output`. Les arguments d'entrée de cette fonction sont :

- **`input`** : une matrice carrée de dimensions `INPUT_SIZE×INPUT_SIZE`, représentant l'image d'entrée.
- **`kernel`** : une matrice carrée de dimensions `IKERNEL_SIZE×KERNEL_SIZE`, représentant le noyau de convolution.
- **`output`** : une matrice carrée de dimensions `IUTPUT_SIZE×OUTPUT_SIZE`, représentant la sortie de la convolution.
- **`bias`** : un scalaire représentant le biais de la couche de convolution.

3. **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.**

La fonction **`maxPooling`** est une opération de sous-échantillonnage, c'est à dire une opération qui consiste à réduire la taille spatiale de la matrice d'entrée. Cela permet notamment de réduire la complexité du modèle et de diminuer la quantité de donnée traitées tout en conservant les données importantes.

La fonction prend en entrée deux paramètres qui sont :

- **`poolinput`** : Une matrice d'entrée de taille  $9 \times 9$
- **`pool`** : Une matrice de sortie de taille  $4 \times 4$

4. **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.**

La fonction `flatten2vector` prend en entrée une matrice multidimensionnelle et renvoie un vecteur à une dimension en "aplatissant" toutes les dimensions de la matrice d'entrée.

Plus précisément, la fonction prend en entrée une matrice de dimension  $4 \times 4$  et renvoie un vecteur de dimension  $16 \times 1$ .

5. **Ajouter une fonction `Printflatten2vector` en code Arduino pour afficher la taille du vecteur.**

On crée la fonction **`Printflatten2vector`** suivante, afin d'afficher la taille du vecteur de sortie de la fonction **`flatten2vector`** :

```
1 /**
2  * @brief Prints the flattened vector
3  */
4 void printflatten2vector()
5 {
6     Serial.print("Flattened Vector Size: ");
7     Serial.println(NumberOf(efflatted));
8     Serial.println();
9 }
```

Après exécution de ce code, on obtient la sortie suivante :

```
1 Flattened Vector Size: 16
```

On obtient donc bien la sortie attendue de 16.

6. Peut on définir le vecteur **expectedOutput**. Si oui comment vous pouvez le générer ?

On utilise une fonction **sigmoid** pour la fonction d'activation de la couche de sortie. Cela implique donc que les résultats attendus doivent être compris entre 0 et 1.

Pour générer un vecteur de sortie attendue on peut observer le contenu de notre vecteur **eflattened**. En effet, il est important que chaque entrée indentique possède la même sortie pour notre vecteur de sortie.

On procède aux modifications suivantes :

```
1 /**
2  * @brief Prints the flattened vector
3  */
4 void printflatten2vector()
5 {
6     Serial.print("Flattened Vector Size: ");
7     Serial.println(NumberOf(eflattened));
8     for (unsigned int i = 0; i < NumberOf(eflattened);
9         i++)
10    {
11        Serial.print(eflattened[i][0]);
12        Serial.print(" ");
13    }
14    Serial.println();
15 }
```

On obtient la sortie suivante :

```
1 Flattened Vector Size: 16
2 2.00 2.00 1.00 2.00 2.00 2.00 3.00 2.00 1.00 3.00 3.0
3 0 2.00 2.00 2.00 2.00 2.00
```

On remarque qu'on possède trois valeurs différentes dans notre vecteur **eflattened**. Il faut donc créer un vecteur de sortie contenant 3 valeurs différentes. On attribuera la valeur 0 à 1.0, la valeur 0.5 à 2.0 et la valeur 1.0 à 3.0. On crée donc le vecteur **expectedOutput** suivant :

```
1 float expectedOutput[NumberOf(eflattened)][1] = {
2     {0.5},
3     {0.5},
4     {0.0},
5     {0.5},
6     {0.5},
7     {0.5},
8     {1.0},
```



```

9    {0.5},
10   {0.0},
11   {1.0},
12   {1.0},
13   {0.5},
14   {0.5},
15   {0.5},
16   {0.5},
17   {0.5}};

```

### 7. Est-il toujours possible d'appliquer **NN.BackProp** ?

Il est toujours possible d'appliquer la fonction **NN.BackProp** car on a bien créé un vecteur de sortie attendue.

### 8. Exécuter le code arduino ci-dessous pour générer une sortie de CNN.

En exécutant le code Arduino fournit en Annexe, on obtient la sortie suivante de CNN :

```

1  ==OUTPUT==
2  0.5040258
3  0.5065975
4  0.0434024
5  0.4976205
6  0.4892119
7  0.4907302
8  0.9078354
9  0.4965506
10 0.1244029
11 0.9028047
12 0.8934928
13 0.4953395
14 0.4956358
15 0.4928572
16 0.5052462
17 0.4997209

```

On remarque immédiatement que l'on obtient des valeurs très proches de notre vecteur de sortie attendue. On peut donc conclure que notre CNN fonctionne correctement.

### 9. 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.

On ajoute une deuxième couche à notre CNN en ajoutant une convolution 2D et un max-pooling. Pour cela, on modifie la fonction **setup** du code Arduino afin d'ajouter cette deuxième couche de CNN.

```

1 void setup()
2 {
3   Serial.begin(115200);
4   /// First Convolutional Layer
5   convolution2D<INPUT_SIZE, INPUT_SIZE, KERNEL_SIZE,
        KERNEL_SIZE, OUTPUT_SIZE, OUTPUT_SIZE>(einput,
        ekernel, eoutput, ebias);
6   maxPooling<OUTPUT_SIZE, OUTPUT_SIZE>(eoutput, epool
        );
7   flatten2vector<(OUTPUT_SIZE/POOL_SIZE)*(OUTPUT_SIZE
        /POOL_SIZE), OUTPUT_SIZE/POOL_SIZE, OUTPUT_SIZE/
        POOL_SIZE>(eflattened, epool);
8   printflatten2vector<(OUTPUT_SIZE/POOL_SIZE)*(
        OUTPUT_SIZE/POOL_SIZE)>(eflattened);
9
10  /// Second Convolutional Layer
11  convolution2D(eoutput, ekernel2, eoutput2, ebias2);
12  maxPooling(eoutput2, epool2);
13  flatten2vector(eflattened2, epool2);
14  printflatten2vector<(OUTPUT_SIZE/POOL_SIZE)*(
        OUTPUT_SIZE/POOL_SIZE)>(eflattened2);
15
16  /// Training of the NN
17 }

```

On obtient la sortie suivante :

```

1 ==OUTPUT==
2 0.5158193
3 0.4966319
4 0.0200490
5 0.5263751
6 0.4992304
7 0.5321954
8 0.9571766
9 0.4746489
10 0.1688640
11 0.9574783
12 0.9575736
13 0.4916144
14 0.4915983
15 0.4887536
16 0.4916826
17 0.4915988

```

On peut remarquer que nos résultats sont encore plus proches de notre vecteur de sortie attendue que lors de la première couche. On peut donc conclure que notre CNN fonctionne correctement.

# Annexes

## A - Code d'un RNN

```
1 /**
2  * @brief This code illustrate a small RNN
3  * @author M. DERRAZ
4  */
5 #include <Arduino.h>
6
7 // Constants
8 const int inputSize = 5;           // Number of input neurons
9 const int hiddenSize = 10;         // Number of hidden neurons
10 const int outputSize = 1;          // Number of output neurons
11 const int sequenceLength = 20;     // Length of the sequence
12                                     // to be predicted
13 const int ledPin = 13;              // Pin for the LED display
14 // Variables
15 float inputs[inputSize];            // Input sequence
16 float outputs[outputSize];          // Output
17                                     // sequence
18 float hidden[hiddenSize];           // Hidden state
19 float weightsIH[hiddenSize][inputSize]; // Input-hidden
20                                     // weights
21 float weightsHH[hiddenSize][hiddenSize]; // Hidden-hidden
22                                     // weights
23 float weightsHO[outputSize][hiddenSize]; // Hidden-output
24                                     // weights
25 float biasH[hiddenSize];             // Hidden biases
26 float biasO[outputSize];             // Output biases
27 float threshold = 0.5;               // Prediction
28                                     // threshold
29 bool predict = false;                // Flag for
30                                     // starting prediction sequence
31
32 /// Funtions declaration
33 void initializeWeights();
34 void forwardPass(float *input, float *output, float *
35                 hidden);
36 float activation(float x);
37 void readPoten();
38 void startPrediction();
```

Code A.1 – Constants for a simple RNN

```

1 void setup()
2 {
3     // Setup pins
4     Serial.begin(115200);
5     pinMode(ledPin, OUTPUT);
6     // Initialize weights and biases
7     initializeWeights();
8 }
9
10 void loop()
11 {
12     // Read potentiometer value and adjust threshold
13     readPoten();
14     // Check if start button is pressed
15
16     startPrediction();
17
18     if (predict)
19     {
20         forwardPass(inputs, outputs, hidden);
21
22         // Output predicted value on LED display
23         if (outputs[0] > threshold)
24         {
25             digitalWrite(ledPin, HIGH);
26         }
27         else
28         {
29             digitalWrite(ledPin, LOW);
30         }
31
32         Serial.print("Output: ");
33         Serial.println(outputs[0]);
34
35         // Shift input sequence
36         for (int i = sequenceLength - 1; i > 0; i--)
37         {
38             inputs[i] = inputs[i - 1];
39         }
40         inputs[0] = outputs[0];
41     }
42 }

```

Code A.2 – Loop and Setup functions for a simple RNN

```

1  /**
2  * @brief Initialize weights and biases
3  */
4  void initializeWeights()
5  {
6      // Initialize input-hidden weights
7      for (int i = 0; i < hiddenSize; i++)
8      {
9          for (int j = 0; j < inputSize; j++)
10         {
11             weightsIH[i][j] = random(-100, 100) / 100.0;
12         }
13     }
14     // Initialize hidden-hidden weights
15     for (int i = 0; i < hiddenSize; i++)
16     {
17         for (int j = 0; j < hiddenSize; j++)
18         {
19             weightsHH[i][j] = random(-100, 100) / 100.0;
20         }
21     }
22     // Initialize hidden-output weights
23     for (int i = 0; i < outputSize; i++)
24     {
25         for (int j = 0; j < hiddenSize; j++)
26         {
27             weightsHO[i][j] = random(-100, 100) / 100.0;
28         }
29     }
30     // Initialize biases
31     for (int i = 0; i < hiddenSize; i++)
32     {
33         biasH[i] = random(-100, 100) / 100.0;
34     }
35     for (int i = 0; i < outputSize; i++)
36     {
37         biasO[i] = random(-100, 100) / 100.0;
38     }
39 }
40
41 /**
42 * @brief Forward pass of the RNN to compute the output
43     sequence
44 */
45 void forwardPass(float *input, float *output, float *
46     hidden)
47 {

```

```

46 // Compute hidden state
47 for (int i = 0; i < hiddenSize; i++)
48 {
49     hidden[i] = 0;
50     for (int j = 0; j < inputSize; j++)
51     {
52         hidden[i] += weightsIH[i][j] * input[j];
53     }
54     for (int j = 0; j < hiddenSize; j++)
55     {
56         hidden[i] += weightsHH[i][j] * hidden[j];
57     }
58     hidden[i] += biasH[i];
59     hidden[i] = activation(hidden[i]);
60 }
61 // Compute output
62 for (int i = 0; i < outputSize; i++)
63 {
64     output[i] = 0;
65     for (int j = 0; j < hiddenSize; j++)
66     {
67         output[i] += weightsHO[i][j] * hidden[j];
68     }
69     output[i] += biasO[i];
70     output[i] = activation(output[i]);
71 }
72 }
73
74 /**
75  * @brief Sigmoid activation function
76  */
77 float activation(float x)
78 {
79     // Sigmoid activation function
80     return 1.0 / (1.0 + exp(-x));
81 }
82
83 /**
84  * @brief Read potentiometer value and adjust prediction
85         threshold
86  */
87 void readPoten()
88 {
89     // Read poten value and map to prediction threshold
90     int potValue = 0.5; // exemple
91 }
92

```

```

93 /**
94  * @brief Start prediction sequence by initializing input
    sequence with random values
95  * and by setting the predict flag to true
96  */
97 void startPrediction()
98 {
99     // Initialize input sequence with random values
100    for (int i = 0; i < sequenceLength; i++)
101    {
102        inputs[i] = random(0, 10) / 10.0;
103    }
104    // Set flag to start prediction sequence
105    predict = true;
106 }

```

Code A.3 – RNN Functions to activate, predict and train the NN

## B - Code d'un CNN

```

1  #include <Arduino.h>
2  #include <NeuralNetwork.h>
3  #include <math.h>
4
5  #define INPUT_SIZE 9    // input image size
6  #define KERNEL_SIZE 3  // kernel size
7  #define PADDING_SIZE 1 // padding size
8  #define STRIDE_SIZE 1  // stride size
9  #define POOL_SIZE 2    // max-pooling size
10 #define OUTPUT_SIZE ((INPUT_SIZE - KERNEL_SIZE + (2 *
    PADDING_SIZE)) / STRIDE_SIZE + 1)
11 #define NumberOf(arg) ((unsigned int)(sizeof(arg) / sizeof
    (arg[0])))
12 #define _1_OPTIMIZE B00010000
13 #define ACTIVATION__PER_LAYER
14 #define SIGMOID 0
15 #define TANH 1
16
17 float eoutput[OUTPUT_SIZE][OUTPUT_SIZE];
18 float epool[OUTPUT_SIZE / POOL_SIZE][OUTPUT_SIZE /
    POOL_SIZE];
19 float eflattened[(OUTPUT_SIZE / POOL_SIZE) * (OUTPUT_SIZE
    / POOL_SIZE)][1];
20 float ebias(0.);
21
22 // initialize input, kernel, and bias
23 float einput[INPUT_SIZE][INPUT_SIZE] = {

```

```

24     {0, 0, 0, 0, 0, 0, 0, 0, 0},
25     {0, 1, 0, 0, 0, 0, 0, 1, 0},
26     {0, 0, 1, 0, 0, 0, 1, 0, 0},
27     {0, 0, 0, 1, 0, 1, 0, 0, 0},
28     {0, 0, 0, 0, 1, 0, 0, 0, 0},
29     {0, 0, 0, 1, 0, 1, 0, 0, 0},
30     {0, 0, 1, 0, 0, 0, 1, 0, 0},
31     {0, 1, 0, 0, 0, 0, 0, 1, 0},
32     {0, 0, 0, 0, 0, 0, 0, 0, 0}};
33
34 float ekernel[KERNEL_SIZE][KERNEL_SIZE] =
35     {{0, 1, 0},
36      {1, 1, 1},
37      {0, 1, 1}};
38
39 float expectedOutput[NumberOf(eflattened)][1] = {
40     {0.5},
41     {0.5},
42     {0.0},
43     {0.5},
44     {0.5},
45     {0.5},
46     {1.0},
47     {0.5},
48     {0.0},
49     {1.0},
50     {1.0},
51     {0.5},
52     {0.5},
53     {0.5},
54     {0.5},
55     {0.5}};
56
57 float ekernel2[KERNEL_SIZE][KERNEL_SIZE] =
58     {{0, 1, 0},
59      {1, 1, 1},
60      {0, 1, 1}};
61 float ebias2(0.);
62
63 // initialize output, pool, and flattened arrays for
64 // second layer
65 float eoutput2[OUTPUT_SIZE][OUTPUT_SIZE];
66 float epool2[OUTPUT_SIZE / POOL_SIZE][OUTPUT_SIZE /
67 POOL_SIZE];
68 float eflattened2[(OUTPUT_SIZE / POOL_SIZE) * (OUTPUT_SIZE
69 / POOL_SIZE)][1];
70
71 // Neural Network Parameters

```



```

69 unsigned int layers[] = {NumberOf(eflattened), 6, 3, 1};
70 byte Actv_Functions[] = {TANH, TANH, SIGMOID};
71 NeuralNetwork NN(layers, NumberOf(layers), Actv_Functions)
    ;
72 float *outputs; // output of the neural network
73
74 template <size_t flattenedSize1D>
75 void printflatten2vector(float (&eflattened)[flattenedSize
    1D][1]);
76
77 template <int InputSize1D, int InputSize2D, int KernelSize
    1D, int KernelSize2D, int OutputSize1D, int OutputSize2
    D>
78 void convolution2D(float (&input)[InputSize1D][InputSize2D
    ], float (&kernel)[KernelSize1D][KernelSize2D], float
    (&output)[OutputSize1D][OutputSize2D], float bias);
79
80 template <size_t rows, size_t cols>
81 void maxPooling(float (&poolinput)[rows][cols], float (&
    pool)[rows / 2][cols / 2]);
82
83 template <size_t flattenedSize1D, size_t poolSize1D,
    size_t poolSize2D>
84 void flatten2vector(float (&flattened)[flattenedSize1D][1
    ], float (&pool)[poolSize1D][poolSize2D]);

```

Code A.4 – Constants for a simple CNN

```

1 void setup()
2 {
3     Serial.begin(115200);
4     /// First Convolutional Layer
5     convolution2D<INPUT_SIZE, INPUT_SIZE, KERNEL_SIZE,
        KERNEL_SIZE, OUTPUT_SIZE, OUTPUT_SIZE>(einput,
        ekernel, eoutput, ebias);
6     maxPooling<OUTPUT_SIZE, OUTPUT_SIZE>(eoutput, epool);
7     flatten2vector<(OUTPUT_SIZE/POOL_SIZE)*(OUTPUT_SIZE/
        POOL_SIZE), OUTPUT_SIZE/POOL_SIZE, OUTPUT_SIZE/
        POOL_SIZE>(eflattened, epool);
8     printflatten2vector<(OUTPUT_SIZE/POOL_SIZE)*(OUTPUT_SIZE
        /POOL_SIZE)>(eflattened);
9     /// Second Convolutional Layer
10    convolution2D(eoutput, ekernel2, eoutput2, ebias2);
11    maxPooling(eoutput2, epool2);
12    flatten2vector(eflattened2, epool2);
13    printflatten2vector<(OUTPUT_SIZE/POOL_SIZE)*(OUTPUT_SIZE
        /POOL_SIZE)>(eflattened2);
14    /// Feed Forward and Back Propagation
15    do
16    {
17        for (unsigned int j = 0; j < NumberOf(eflattened); j
            ++))
18        {
19            NN.FeedForward(eflattened[j]);
20            NN.BackProp(expectedOutput[j]);
21        }
22        Serial.print("J curretirion Error: "); // Prints the
            Error.
23        Serial.println(NN.MeanSqrdError, 4);
24    } while (NN.getMeanSqrdError(NumberOf(eflattened)) > 0.0
        03);
25    Serial.print("==OUTPUT==");
26    for(unsigned int i = 0; i < NumberOf(eflattened); i++)
27    {
28        outputs = NN.FeedForward(eflattened[i]);
29        Serial.println(outputs[0], 7);
30    }
31    NN.print();
32 }
33 void loop()
34 {}

```

Code A.5 – Loop and Setup functions for a simple CNN

```

1  /**
2  * @brief Convolution 2D function for a 2D array of floats
   * with a 2D kernel of floats and a bias of float type.
3  * @param input 2D array of floats
4  * @param kernel 2D array of floats
5  * @param output 2D array of floats
6  * @param bias float type
7  * @tparam InputSize1D size of the first dimension of the
   * input array
8  * @tparam InputSize2D size of the second dimension of the
   * input array
9  * @tparam KernelSize1D size of the first dimension of the
   * kernel array
10 * @tparam KernelSize2D size of the second dimension of the
   * kernel array
11 * @tparam OutputSize1D size of the first dimension of the
   * output array
12 * @tparam OutputSize2D size of the second dimension of the
   * output array
13 */
14 template <int InputSize1D, int InputSize2D, int KernelSize
   1D, int KernelSize2D, int OutputSize1D, int OutputSize2
   D>
15 void convolution2D(float (&input)[InputSize1D][InputSize2D
   ], float (&kernel)[KernelSize1D][KernelSize2D], float
   (&output)[OutputSize1D][OutputSize2D], float bias)
16 {
17     for (int i = 0; i < OutputSize1D; i++)
18     {
19         for (int j = 0; j < OutputSize2D; j++)
20         {
21             output[i][j] = 0;
22             for (int k = 0; k < KernelSize1D; k++)
23             {
24                 for (int l = 0; l < KernelSize2D; l++)
25                 {
26                     int input_row = i * STRIDE_SIZE + k -
                       PADDING_SIZE;
27                     int input_col = j * STRIDE_SIZE + l -
                       PADDING_SIZE;
28                     if (input_row >= 0 && input_row < InputSize1D
                       && input_col >= 0 && input_col < InputSize2
                       D)
29                     {
30                         output[i][j] += input[input_row][input_col] *
                           kernel[k][l];
31                     }

```

```

32     }
33     }
34     output[i][j] += bias;
35 }
36 }
37 }
38
39 /**
40  * @brief Max pooling function for 2D vectors which create
41  *        a new 2D vector with half the size of the input
42  * @param poolinput The input 2D vector
43  * @param pool The output 2D vector
44  * @tparam rows The number of rows in the input vector
45  * @tparam cols The number of columns in the input vector
46  */
47 template <size_t rows, size_t cols>
48 void maxPooling(float (&poolinput)[rows][cols], float (&
49 pool)[rows / 2][cols / 2])
50 {
51     for (int i = 0; i < rows / 2; i++)
52     {
53         for (int j = 0; j < cols / 2; j++)
54         {
55             float maxVal = -INFINITY;
56             for (int k = 0; k < 2; k++)
57             {
58                 for (int l = 0; l < 2; l++)
59                 {
60                     maxVal = max(maxVal, poolinput[i * 2 + k][j *
61 2 + l]);
62                 }
63             }
64             pool[i][j] = maxVal;
65         }
66     }
67 }
68
69 /**
70  * @brief Flattens a 2D vector into a 1D vector
71  * @param flattened The flattened 1D vector
72  * @param pool The 2D vector to be flattened
73  * @tparam flattenedSize1D The size of the flattened vector
74  * @tparam poolSize1D The size of the 2D vector in the
75  *        first dimension
76  * @tparam poolSize2D The size of the 2D vector in the
77  *        second dimension
78  */
79 template <size_t flattenedSize1D, size_t poolSize1D,

```

```

    size_t poolSize2D>
75 void flatten2vector(float (&flattened)[flattenedSize1D][1
    ], float (&pool)[poolSize1D][poolSize2D]) {
76     int idx = 0;
77     for (int i = 0; i < poolSize1D; i++) {
78         for (int j = 0; j < poolSize2D; j++) {
79             flattened[idx++][0] = pool[i][j];
80         }
81     }
82 }
83
84 /**
85  * @brief Prints the flattened 1D vector
86  * @param eflattened The flattened 1D vector
87  * @tparam flattenedSize1D The size of the flattened vector
88  */
89 template <size_t flattenedSize1D>
90 void printflatten2vector(float (&eflattened)[flattenedSize
    1D][1])
91 {
92     Serial.print("Flattened Vector Size: ");
93     Serial.println(flattenedSize1D);
94     for (unsigned int i = 0; i < flattenedSize1D; i++)
95     {
96         Serial.print(eflattened[i][0]);
97         Serial.print(" ");
98     }
99     Serial.println();
100 }

```

Code A.6 – Convolution functions