RAPPORT DE TP

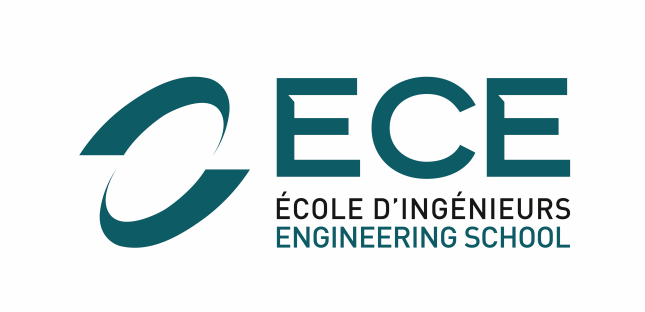
TP 6 : Apprentissage Profond

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

Introduction

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

Liste des abréviations

— **RNN** : Réseau de Neurones Récurrents (Recurrent Neural Network en anglais)
— **CNN** : Réseau de Neurones Convolutifs (Convolutional Neura 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 calculées 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 lue 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'intialiser les valeurs des **poids** et des **biais** contenus dans les différentes matrices **weightsIH**, **weightsHH**, **weightsHO**, **biasH** et **biasO**. Les différentes valeurs sont initialisées aléatoirement 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 réseau.
   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 **sigmoïd** en tant que fonction d'activation sont 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 nécéssite 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 fixe le treshold sur cette dernière.
   Le treshold permet de **considérer** si une sortie 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**. Ensuite, elle 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 9×9. 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 9×9, 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 correspondent à :
   — `INPUT_SIZE` : la taille de la matrice d'entrée, ici 9×9 pixels.
   — `KERNEL_SIZE` : la taille du noyau de convolution utilisé pour filtrer la matrice, ici 3×3 pixels.
   — `PADDING_SIZE` : la taille de remplissage ajoutée autour de la matrice, ici 1 pixel.
   — `STRIDE_SIZE` : la taille du pas de la fenêtre de convolution lors du glissement de l'image d'entrée, ici 1 pixel.
   — `POOL_SIZE` : la taille de la fenêtre utilisée pour effectuer le max-pooling sur la sortie de la convolution, ici 2×2 pixels.
   Ces paramètres sont utilisés 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-échantillonage, 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 données traitées tout en conservant les données importantes.
   La fonction prend en entrée deux paramètres qui sont :
   — **poolinput** : Une matrice d'entrée de taille 9×9
   — **pool** : Une matrice de sortie de taille 4×4

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 4×4 et renvoie un vecteur de dimension 16×1.

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** :

```
/**
* @brief Prints the flattened vector
*/
void printflatten2vector()
{
  Serial.print("Flattened Vector Size: ");
  Serial.println(NumberOf(eflattened));
  Serial.println();
```

```
}
```

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

```
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 **sïgmoid** 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 le vecteur de sortie attendu 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 :

```
/**
* @brief Prints the flattened vector
*/
void printflatten2vector()
{
  Serial.print("Flattened Vector Size: ");
  Serial.println(NumberOf(eflattened));
  for (unsigned int i = 0; i < NumberOf(eflattened); i++)
  {
    Serial.print(eflattened[i][0]);
    Serial.print(" ");
  }
  Serial.println();
}
```

   On obtient la sortie suivante :

```
Flattened Vector Size: 16
2.00 2.00 1.00 2.00 2.00 2.00 3.00 2.00 1.00 3.00 3.00 2.
   00 2.00 2.00 2.00 2.00
```

   On remarque que l'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 :

```
float expectedOutput[NumberOf(eflattened)][1] = {
  {0.5},
  {0.5},
  {0.0},
  {0.5},
  {0.5},
```

```
  {0.5},
  {1.0},
  {0.5},
  {0.0},
  {1.0},
  {1.0},
  {0.5},
  {0.5},
  {0.5},
  {0.5},
  {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 créer le vecteur de sortie attendue.

8. **Executer le code arduino ci-dessous pour générer une sortie de CNN.**

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

```
==OUTPUT==
0.5040258
0.5065975
0.0434024
0.4976205
0.4892119
0.4907302
0.9078354
0.4965506
0.1244029
0.9028047
0.8934928
0.4953395
0.4956358
0.4928572
0.5052462
0.4997209
```

   On remarque imédiatement que l'on obtient des valeurs très proches de notre vecteur de sortie attendu. 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 deuxieme couche de **CNN**.

```
void setup()
{
  Serial.begin(115200);
  /// First Convolutional Layer
  convolution2D<INPUT_SIZE, INPUT_SIZE, KERNEL_SIZE,
     KERNEL_SIZE, OUTPUT_SIZE, OUTPUT_SIZE>(einput,
     ekernel, eoutput, ebias);
  maxPooling<OUTPUT_SIZE, OUTPUT_SIZE>(eoutput, epool);
  flatten2vector<(OUTPUT_SIZE/POOL_SIZE)*(OUTPUT_SIZE/
     POOL_SIZE), OUTPUT_SIZE/POOL_SIZE, OUTPUT_SIZE/
     POOL_SIZE>(eflattened, epool);
  printflatten2vector<(OUTPUT_SIZE/POOL_SIZE)*(
     OUTPUT_SIZE/POOL_SIZE)>(eflattened);

  /// Second Convolutional Layer
  convolution2D(eoutput, ekernel2, eoutput2, ebias2);
  maxPooling(eoutput2, epool2);
  flatten2vector(eflattened2, epool2);
  printflatten2vector<(OUTPUT_SIZE/POOL_SIZE)*(
     OUTPUT_SIZE/POOL_SIZE)>(eflattened2);

  /// Training of the NN
}
```

On obtient la sortie suivante :

```
==OUTPUT==
0.5158193
0.4966319
0.0200490
0.5263751
0.4992304
0.5321954
0.9571766
0.4746489
0.1688640
0.9574783
0.9575736
0.4916144
0.4915983
0.4887536
0.4916826
0.4915988
```

On peut remarquer que nos résultats sont encore plus proches de notre vecteur de sortie

attendu que lors de la première couche. On peut donc conclure que notre CNN fonctionne correctement.

Annexes

A - Code d'un RNN

```
/**
* @brief This code illustrate a small RNN
* @author M. DERRAZ
*/
#include <Arduino.h>

// Constants
const int inputSize = 5;        // Number of input neurons
const int hiddenSize = 10;      // Number of hidden neurons
const int outputSize = 1;       // Number of output neurons
const int sequenceLength = 20; // Length of the sequence to
    be predicted
const int ledPin = 13;          // Pin for the LED display
// Variables
float inputs[inputSize];                    // Input sequence
float outputs[outputSize];                  // Output sequence
float hidden[hiddenSize];                   // Hidden state
float weightsIH[hiddenSize][inputSize];   // Input-hidden
    weights
float weightsHH[hiddenSize][hiddenSize]; // Hidden-hidden
    weights
float weightsHO[outputSize][hiddenSize]; // Hidden-output
    weights
float biasH[hiddenSize];                    // Hidden biases
float biasO[outputSize];                    // Output biases
float threshold = 0.5;                      // Prediction
    threshold
bool predict = false;                       // Flag for starting
     prediction sequence

/// Funtions declaration
void initializeWeights();
void forwardPass(float *input, float *output, float *hidden);
float activation(float x);
void readPoten();
void startPrediction();
```

Code A.1 – Constants for a simple RNN

```
void setup()
{
    // Setup pins
    Serial.begin(115200);
    pinMode(ledPin, OUTPUT);
    // Initialize weights and biases
    initializeWeights();
}

void loop()
{
    // Read potentiometer value and adjust threshold
    readPoten();
    // Check if start button is pressed

    startPrediction();

    if (predict)
    {
    forwardPass(inputs, outputs, hidden);

    // Output predicted value on LED display
    if (outputs[0] > threshold)
    {
        digitalWrite(ledPin, HIGH);
    }
    else
    {
        digitalWrite(ledPin, LOW);
    }

    Serial.print("Output: ");
    Serial.println(outputs[0]);

    // Shift input sequence
    for (int i = sequenceLength - 1; i > 0; i--)
    {
        inputs[i] = inputs[i - 1];
    }
    inputs[0] = outputs[0];
    }
}
```

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

```
/**
* @brief Initialize weights and biases
*/
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;
    }
}

/**
* @brief Forward pass of the RNN to compute the output
    sequence
*/
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]);
    }
}

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

/**
* @brief Read potentiometer value and adjust prediction
    threshold
*/
void readPoten()
{
    // Read poten value and map to prediction threshold
    int potValue = 0.5; // exemple
```

```
90 }
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
     ];
```

```
float eflattened[(OUTPUT_SIZE / POOL_SIZE) * (OUTPUT_SIZE / POOL_SIZE)][1];
float ebias(0.);

// 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 expectedOutput[NumberOf(eflattened)][1] = {
    {0.5},
    {0.5},
    {0.0},
    {0.5},
    {0.5},
    {0.5},
    {1.0},
    {0.5},
    {0.0},
    {1.0},
    {1.0},
    {0.5},
    {0.5},
    {0.5},
    {0.5},
    {0.5}};

float ekernel2[KERNEL_SIZE][KERNEL_SIZE] =
    {{0, 1, 0},
        {1, 1, 1},
        {0, 1, 1}};
float ebias2(0.);

```

```
63 // initialize output, pool, and flattened arrays for second
       layer
64 float eoutput2[OUTPUT_SIZE][OUTPUT_SIZE];
65 float epool2[OUTPUT_SIZE / POOL_SIZE][OUTPUT_SIZE / POOL_SIZE
       ];
66 float eflattened2[(OUTPUT_SIZE / POOL_SIZE) * (OUTPUT_SIZE /
       POOL_SIZE)][1];
67
68 // 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)[flattenedSize1D
       ][1]);
76
77 template <int InputSize1D, int InputSize2D, int KernelSize1D,
        int KernelSize2D, int OutputSize1D, int OutputSize2D>
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

```
void setup()
{
  Serial.begin(115200);
  /// First Convolutional Layer
  convolution2D<INPUT_SIZE, INPUT_SIZE, KERNEL_SIZE,
     KERNEL_SIZE, OUTPUT_SIZE, OUTPUT_SIZE>(einput, ekernel,
     eoutput, ebias);
  maxPooling<OUTPUT_SIZE, OUTPUT_SIZE>(eoutput, epool);
  flatten2vector<(OUTPUT_SIZE/POOL_SIZE)*(OUTPUT_SIZE/
     POOL_SIZE), OUTPUT_SIZE/POOL_SIZE, OUTPUT_SIZE/POOL_SIZE
     >(eflattened, epool);
  printflatten2vector<(OUTPUT_SIZE/POOL_SIZE)*(OUTPUT_SIZE/
     POOL_SIZE)>(eflattened);
  /// Second Convolutional Layer
  convolution2D(eoutput, ekernel2, eoutput2, ebias2);
  maxPooling(eoutput2, epool2);
  flatten2vector(eflattened2, epool2);
  printflatten2vector<(OUTPUT_SIZE/POOL_SIZE)*(OUTPUT_SIZE/
     POOL_SIZE)>(eflattened2);
  /// Feed Forward and Back Propagation
  do
  {
    for (unsigned 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)
     ;
  Serial.print("==OUTPUT==");
  for(unsigned int i = 0; i < NumberOf(eflattened); i++)
  {
    outputs = NN.FeedForward(eflattened[i]);
    Serial.println(outputs[0], 7);
  }
  NN.print();
}
void loop()
{}
```

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

```
/**
* @brief Convolution 2D function for a 2D array of floats with a 2D kernel of floats and a bias of float type.
* @param input 2D array of floats
* @param kernel 2D array of floats
* @param output 2D array of floats
* @param bias float type
* @tparam InputSize1D size of the first dimension of the input array
* @tparam InputSize2D size of the second dimension of the input array
* @tparam KernelSize1D size of the first dimension of the kernel array
* @tparam KernelSize2D size of the second dimension of the kernel array
* @tparam OutputSize1D size of the first dimension of the output array
* @tparam OutputSize2D size of the second dimension of the output array
*/
template <int InputSize1D, int InputSize2D, int KernelSize1D, int KernelSize2D, int OutputSize1D, int OutputSize2D>
void convolution2D(float (&input)[InputSize1D][InputSize2D], float (&kernel)[KernelSize1D][KernelSize2D], float (&output)[OutputSize1D][OutputSize2D], float bias)
{
    for (int i = 0; i < OutputSize1D; i++)
    {
    for (int j = 0; j < OutputSize2D; j++)
    {
        output[i][j] = 0;
        for (int k = 0; k < KernelSize1D; k++)
        {
        for (int l = 0; l < KernelSize2D; l++)
        {
            int input_row = i * STRIDE_SIZE + k - PADDING_SIZE;
            int input_col = j * STRIDE_SIZE + l - PADDING_SIZE;
            if (input_row >= 0 && input_row < InputSize1D && input_col >= 0 && input_col < InputSize2D)
            {
            output[i][j] += input[input_row][input_col] * kernel[k][l];
            }
```

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

```
73 */
74 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)[flattenedSize1D
       ][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