Projeto 2 - EP

Universidade de Aveiro

João Torrinhas, Diogo Torrinhas

Projeto 2 - EP

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

João Torrinhas, Diogo Torrinhas
(98435) joao.torrinhas@ua.pt, (98440) diogotorrinhas@ua.pt

November 27, 2023

Contents

Chapter 1

NN_Base module

1.1 User Manual

Para compilar e correr este programa basta seguir os seguintes passos:

1. Abrir o terminal no diretório onde se encontra o programa `nn_base.c`.
2. Após estar no diretório correto, use o seguinte comando no terminal para compilar o programa:

   ```
   gcc -o nn_base nn_base.c -lm -Wall
   ```

   Após correr este comando vai ser criado um executável com o nome `nn_base`.
3. Em seguida, no mesmo terminal, use o seguinte comando para correr o executável `nn_base`:

   ```
   ./nn_base [iterations] [learning_rate] [threshold] [training_mode]
   ```

   [iterations] - Corresponde ao valor máximo de iterações que a rede neural tem para efetuar o treino, por exemplo 10000 iterações.

   [learning_rate] - É a variável que vai determinar a velocidade no treino da rede neural, isto é, o quanto os pesos da rede são ajustados em relação ao gradiente da função de perda.

   [threshold] - É o limite para o qual consideramos que um valor de output é válido em relação ao valor esperado, por exemplo 0.001.

   [training_mode] - **0** se for o modo de treino normal (passar inputs e outputs, no exemplo deste projeto estes valores já estão definidos por default) e **1** se for o modo de treino *XOR*.

   Este comando vai correr o programa compilado `nn_base` e vai ser possível ver no terminal o resultado do programa.

1.2 Code Explanation

Neste capítulo vai ser explicado em detalhe todo o código desenvolvido neste projeto.

1.2.1 nn_base.h

O ficheiro nn_base.h é um header file e define a estrutura e as funções necessárias para criar, destruir, administrar e usar uma rede neural básica.

O elemento central do ficheiro nn_base.h é a estrutura de dados NeuralNetwork que vai armazenar os atributos necessários para a mesma. Em relação ao projeto anterior, foram adicionados alguns atributos para a resolução deste projeto, tal como mostra o excerto de código abaixo.

Listing 1.1: NeuralNetwork Structure

```
typedef struct
{
    int number_I; //Number of Input Units
    int number_H; //Number of Hidden Units
    int number_O; //Number of Output Units
    double** weights_to_hidden; //I->H
    double** weights_to_output; //H->O
    double* I;  // Array to store input values
    double* H; // Array to store hidden layer values
    double* O; // Array to store output value
    double* bias_to_hidden; // Array to store the weights of the bias to the hidden layer
    double* bias_to_output; // Array to store the weights of the bias to the output layer
    double* errors_output; // Array to store the errors of the output layer
    double* inp_hidden; // Array to store the input values of each hidden unit
    double* inp_output; // Array to store the input values of each output unit
    double* delta_hidden; // // Array to store the delta values of each hidden unit
    double* delta_output; // Array to store the delta values of each output unit
} NeuralNetwork;
```

As variáveis **bias_to_hidden** e **bias_to_output** são arrays que vão guardar o peso das arestas que ligam a unidade bias da *input layer* às unidades da *hidden layer* e o peso das arestas que ligam a unidade bias da *hidden layer* às unidades da *output layer*, respetivamente.

Errors_output é um array que vai guardar o valor do erro para cada *output unit* na *output layer*.

Os arrays **inp_hidden** e **inp_output** vão guardar o valor *inp* (cálculo que vai ser efetuado na propagação) para cada unidade da *hidden layer* e da *output layer*, respetivamente.

Por fim, os arrays **delta_hidden** e **delta_output** têm a mesma função que os arrays **inp_hidden** e **inp_output** mas guardam o valor do delta em vez de guardar o valor inp.

Também foram declarados mais alguns métodos em relação ao projeto anterior que vão ser explicados em detalhe nos próximos capítulos.

Listing 1.2: Declared Functions

```
NeuralNetwork* nn_create(int trainingMode);
void propagate(NeuralNetwork *nn);
void load_initial_bias_weights_to_outputLayer(double values[], NeuralNetwork *nn);
void load_initial_bias_weights_to_hiddenLayer(double values[], NeuralNetwork *nn);
void back_propagate(NeuralNetwork *nn, double learning_rate);
void load_input_values(double input_values[], NeuralNetwork *nn);
void load_weights(NeuralNetwork *nn);
void load_nn(NeuralNetwork *nn);
void nn_destroy(NeuralNetwork *nn);
```

1.2.2 nn_base.c

Neste ficheiro vão ser implementadas as funções declaradas no ficheiro **nn_base.h** e é onde vai ser implementada a função *main* para executar o que é pedido no enunciado.

1.2.2.1 Method nn_create

A função **nn_create** é principalmente responsável por criar uma nova rede neural alocando memória para a rede neural e para um conjunto de atributos da mesma inicializando também as arestas e um conjunto de

variáveis a 0. Por fim, retorna um ponteiro para a rede neural criada.

Inicialmente, é alocada memória para a rede neural e, em seguida, é feita a verificação para saber se a memória foi alocada com sucesso, tal como mostra o excerto de código abaixo.

Listing 1.3: Neural Network allocation

```
NeuralNetwork *nn = malloc(sizeof(NeuralNetwork));
if (nn == NULL) {
    fprintf(stderr, "Error allocating memory for NeuralNetwork\n");
    exit(EXIT_FAILURE);
}
```

Posto isto, de acordo com o parâmetro de entrada **trainingMode** (0 = treino normal, 1 = treino para implementar a função booleana *XOR*), vai ser lida a primeira linha de um ficheiro de entrada (cada ficheiro tem contéudo diferente, ou seja, número de unidades diferentes em cada *layer*, depende do tipo de treino que o utilizador escolher), para atribuir os valores do número de unidades aos respetivos atributos (**number_I**, **number_H** e **number_O**) da estrutura da rede neural.

No caso do utilizador escolher o modo de treino para implementar a função booleana *XOR*, a rede neural vai ter 2 unidades de entrada, mais a bias, (valores de entrada: 00, 01, 10, 11), 10 unidades no meio, mais a bias, (por default usamos duas mas pode ser alterado basta mudar o número do meio do ficheiro de entrada *XOR_NeuralNetwork.txt*) e 1 unidade de saída (0 ou 1).

Listing 1.4: Read input files based on trainingMode

```
//Normal training mode, read number inputs,hidden,output from NeuralNetwork1 file
if(trainingMode == 0){
    FILE *fp = fopen("NeuralNetwork1.txt", "r");
    if (fp == NULL) {
        fprintf(stderr, "Error opening file NeuralNetwork1.txt\n");
        exit(EXIT_FAILURE);
    }

    // Read the first line (I H O)
    if (fscanf(fp, "%d %d %d", &number_inputs, &number_hidden, &number_outputs) == 3) {
        //Add one bias unit to the last position of the input layer
        nn->number_I = number_inputs + 1;
        //Add one bias unit to the last position of the hidden layer
        nn->number_H = number_hidden + 1;
        nn->number_O = number_outputs;
        fclose(fp);
    }
///XOR training mode, read number inputs,hidden,output from XOR_NeuralNetwork file
}else{
    FILE *fp = fopen("XOR_NeuralNetwork.txt", "r");
    if (fp == NULL) {
        fprintf(stderr, "Error opening file XOR_NeuralNetwork.txt\n");
        exit(EXIT_FAILURE);
    }

    // Read the first line (I H O)
    if (fscanf(fp, "%d %d %d", &number_inputs, &number_hidden, &number_outputs) == 3) {
        //Add one bias unit to the last position of the input layer
        nn->number_I = number_inputs + 1;
        //Add one bias unit to the last position of the hidden layer
        nn->number_H = number_hidden + 1;
        nn->number_O = number_outputs;
        fclose(fp);
    }
}
```

De salientar que, pelo facto de haver uma nova unidade bias na *input* e *hidden* layer, somamos "1" ao respetivo número de unidades da *input* e *hidden* layer lidas do ficheiro de entrada, tal como mostra o excerto de código acima.

Relativamente ao projeto anterior, tal como foi dito na secção **nn_base.h**, foram adicionados novos atributos à rede neural: **bias_to_hidden** e **bias_to_output** são arrays que guardam o peso das arestas que ligam a bias da *input* layer às unidades da *hidden* layer e que liga a bias da *hidden* layer às unidades da *output* layer, respetivamente, os arrays **inp_hidden** e **inp_output** guardam os valores do input de cada unidade da *hidden* layer e *output* layer, respetivamente, e, por fim, os arrays **delta_hidden** e **delta_output** guardam o valor do delta de cada unidade da *hidden* layer e *output* layer, respetivamente.

Em seguida, vai ser também feita a alocação de memória destes novos atributos da mesma maneira que foram feitas as alocações para os outros atributos, ou seja, multiplicar o tamanho de um double pelo número de unidades nas respetivas camadas.

Como há uma unidade bias na camada de entrada e do meio, para alocar espaço, por exemplo, referente apenas às unidades do meio temos de fazer **(nn->number_H-1)**, ou seja, "excluímos" a bias, o mesmo acontece em relação às unidades da camada inicial. Por fim, inicializamos cada uma das bias com o valor **1** através da última posição (posição da bias) do array que contêm os valores das unidades em cada *layer*.

Listing 1.5: Allocation for the new variables and inicialization of the bias

```
//Allocate memory for the bias unit to the output layer
nn->bias_to_output = malloc(sizeof(double) * nn->number_O);
//Allocate memory for the bias unit to the hidden layer
nn->bias_to_hidden = malloc(sizeof(double) * (nn->number_H-1));

nn->errors_output = malloc(sizeof(double) * nn->number_O);
nn->inp_hidden = malloc(sizeof(double) * (nn->number_H-1));
nn->inp_output = malloc(sizeof(double) * nn->number_O);
nn->delta_hidden = malloc(sizeof(double) * (nn->number_H-1));
nn->delta_output = malloc(sizeof(double) * nn->number_O);

//Initialize bias unit to 1 in each layer (first and hidden layer)
nn->H[nn->number_H-1] = 1.0;
nn->I[nn->number_I-1] = 1.0;
```

No fim, após a alocação de memória para todos os atributos, vai ser feito a inicialização dos mesmos através de um ciclo for que vai percorrer o número de unidades em cada layer e vai inicializar cada atributo com o valor **0.0**, tal como mostra o excerto de código abaixo.

Listing 1.6: Inicialization of the atributes

```
//Initialize weight of the bias unit (HIDDEN TO OUTPUT LAYER) to zero
for(int i = 0; i < nn->number_O; i++) {
    nn->bias_to_output[i] = 0.0;
}

//Initialize weight of the bias unit (INPUT TO HIDDEN LAYER) to zero
for(int i = 0; i < nn->number_H-1; i++){
    nn->bias_to_hidden[i] = 0.0;
}

//Initialize errors to zero
for(int i = 0; i < nn->number_O; i++){
    nn->errors_output[i] = 0.0;
}

//Initialize inputs of the units of the hidden/output layer to zero
for(int i = 0; i < nn->number_H-1; i++){
    nn->inp_hidden[i] = 0.0;
}
```

```
for(int i = 0; i < nn->number_O; i++){
    nn->inp_output[i] = 0.0;
}


//Initialize both unit deltas in each layer to zero
for(int i = 0; i < nn->number_H-1; i++){
    nn->delta_hidden[i] = 0.0;
}

for(int i = 0; i < nn->number_O; i++){
    nn->delta_output[i] = 0.0;
}
```

1.2.2.2 Method load_initial_bias_weights_to_outputLayer

O objetivo desta função é atribuir um peso inicial, que posteriormente vai ser treinado, às arestas que ligam a bias da *hidden layer* às unidades da *output layer*.

Posto isto, é feito um ciclo for a percorrer as unidades da *output layer* e é atribuído um peso que liga a bias a essa unidade através do parâmetro de entrada **double values[]** que vai possuir os pesos a serem atribuídos.

Listing 1.7: Method load_initial_bias_weights_to_outputLayer

```
void load_initial_bias_weights_to_outputLayer(double values[], NeuralNetwork *nn){
    for(int i = 0; i < nn->number_O; i++){
        nn->bias_to_output[i] = values[i];
    }
}
```

1.2.2.3 Method load_initial_bias_weights_to_hiddenLayer

O objetivo desta função é semelhante à **load_initial_bias_weights_to_outputLayer** mas atribui o valor dos pesos, que estão no parâmetro de entrada **double values[]**, às arestas que ligam a bias da *input layer* às unidades da *hidden layer* (sem contar com a bias da hidden layer porque não existe ligação entre as bias de cada unidade), tal como mostra o código abaixo.

Listing 1.8: Method load_initial_bias_weights_to_hiddenLayer

```
void load_initial_bias_weights_to_hiddenLayer(double values[], NeuralNetwork *nn){
    for(int i = 0; i < nn->number_H-1; i++){
        nn->bias_to_hidden[i] = values[i];
    }
}
```

1.2.2.4 Method propagate

O objetivo desta função é calcular os resultados de saída propagando os valores de entrada pela rede neural.

O processo de propagação baseia-se no seguinte: vai ser calculado o valor do input, **inp**, para cada unidade através da seguinte fórmula.

$$o_j^l = f\left(\sum_i w_{i,j}^l * o_i^{l-1} + b_j^l\right) \tag{1}$$

Ou seja, vai ser feito o somatório da multiplicação do peso das arestas, que ligam as unidades **i** da *layer l-1* com as unidades j da *layer* l, pelo respetivo valor de saída da unidade da *layer l-1* e, em seguida, é feita a soma com o resultado da multiplicação do peso da aresta que faz a ligação da bias às unidades da layer l com o valor

1 (valor de cada unidade bias).

Por fim, o valor de saída das unidades da *hidden layer* e da *output layer* é cálculado através de uma *sigmoid function*. É passado o valor do **inp** como x na fórmula abaixo e é obtido o respetivo valor **out** para cada unidade da respetiva layer l.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

$$out^l = f(inp) \tag{3}$$

A Figura abaixo mostra um esboço de uma estrutura da rede neural e serve como ajuda para entender o que foi explicado acima e o que vai ser explicado na secção **Method back_propagate**.

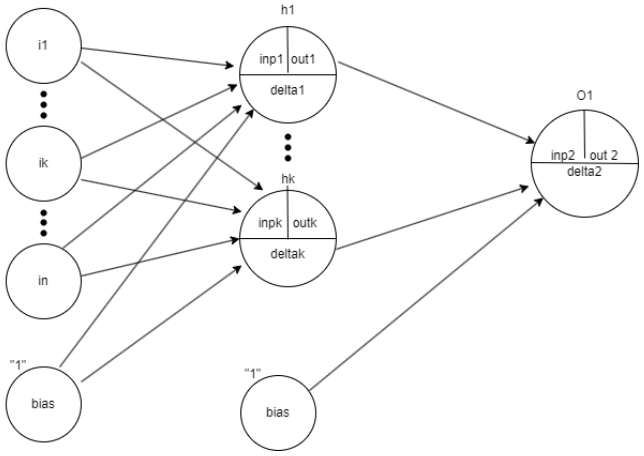


Figure 1.1: Esboço da estrutura da rede neural.

O código abaixo representa o processo de propagação que foi descrito acima.

Listing 1.9: Method propagate

```
void propagate(NeuralNetwork *nn){
    //Propagate input to hidden layer using SIGMOID function
    for(int i = 0; i < nn->number_H-1; i++){
        double somatorio = 0.0;
        for(int j = 0; j< nn->number_I-1; j++){
            somatorio += nn->weights_to_hidden[j][i]*nn->I[j];
        }
        nn->inp_hidden[i] = somatorio + (1*nn->bias_to_hidden[i]);
        nn->H[i] = 1/(1 + exp(-nn->inp_hidden[i]));
    }
    //Propagate hidden to output layer using SIGMOID function
    for(int i = 0; i<nn->number_O; i++){
        double somatorio = 0.0;
        for(int j = 0; j<nn->number_H-1; j++){
            somatorio += nn->weights_to_output[j][i]*nn->H[j];
        }
        nn->inp_output[i] = somatorio + (1*nn->bias_to_output[i]);
        nn->O[i] = 1/(1 + exp(-nn->inp_output[i]));
    }
}
```

1.2.2.5 Method back_propagate

Esta função implementa o algoritmo backpropagation cujo objetivo é minimizar o erro entre a saída desejável e a saída real, ajustando/treinando os pesos da rede. Primeiro, atualiza os pesos que ligam a camada de saída e a camada do meio depois, atualiza os pesos que ligam a camada do meio e a camada inicial.

Através do primeiro ciclo for, para cada unidade de saída (indexada por i) irá ser calculado o delta multiplicando a derivada da função sigmóide pelo respetivo erro nessa unidade.

$$error = (target_i - out_i) \tag{4}$$
$$delta = error_i * f'(inp_i) \tag{5}$$

Depois, atualiza os pesos das arestas que conectam a camada do meio à camada de saída e os pesos que ligam o bias da camada do meio à camada de saída através da seguinte fórmula.

$$w_{ij} = w_{ij} + \mathrm{lr} \cdot \delta_j \cdot o_i \tag{6}$$

Onde w_{ij} representa o peso entre as unidades i e j, lr representa um fator que influencia a velocidade de aprendizagem denominado por *learning rate*, δ_j representa o delta que foi calculado anteriormente e, por fim, o_i representa a saída da unidade i.

Em baixo, encontra-se o código que representa o que foi explicado anteriormente.

Listing 1.10: backpropagate

```
// Update weights from hidden to output layer
for (int i = 0; i < nn->number_O; i++) {
    nn->delta_output[i] = (1/(1 + exp(-nn->inp_output[i]))) * (1 - (1/(1 +
        exp(-nn->inp_output[i])))) * nn->errors_output[i];

    // Update weights
    for (int j = 0; j < nn->number_H - 1; j++) {
        nn->weights_to_output[j][i] = nn->weights_to_output[j][i] + (
            learning_rate * nn->delta_output[i] * nn->H[j]);
    }

    // Update bias
    nn->bias_to_output[i] = nn->bias_to_output[i] + (learning_rate * nn->
        delta_output[i] * 1);
}
```

Em seguida, é calculado o delta (para as unidades do meio) através da multiplicação da derivada da função sigmóide ($f'(inp_i)$) nas unidades do meio, referida anteriormente, pela soma do produto dos pesos conectados à camada de saída ((w_{ij})) com os seus valores delta correspondentes nas unidades de saída ((δ_j)). A fórmula seguinte representa o que foi descrito acima.

$$\delta_i = f'(inp_i) \cdot \sum_{i,j} w_{ij} \cdot \delta_j \tag{7}$$

Depois, atualiza os pesos que conectam a camada de entrada à camada do meio e, por fim, atualiza os pesos que ligam a bias da camada inicial às unidades do meio através da fórmula (6).

O código abaixo descreve o que foi explicado acima

Listing 1.11: backpropagate

```
for (int i = 0; i < nn->number_H - 1; i++) {
    double derivative = (1/(1 + exp(-nn->inp_hidden[i]))) * (1 - (1/(1 + exp
        (-nn->inp_hidden[i])))); //(1/1+(e^-x))*(1-(1/1+(e^-x)) nn->H[i]
        * (1 - nn->H[i]);
    double somatorio = 0.0;

    // Calculate Delta Hidden
    for (int k = 0; k < nn->number_O; k++) {
```

```
            somatorio += (nn->weights_to_output[i][k] * nn->delta_output[k]);
        }

        nn->delta_hidden[i] = derivative * somatorio;

        for(int j = 0; j < nn->number_I-1; j++){
            nn->weights_to_hidden[j][i] = nn->weights_to_hidden[j][i] + (
                learning_rate* nn->delta_hidden[i] * nn->I[j]);
        }

        // Update bias
        nn->bias_to_hidden[i] = nn->bias_to_hidden[i] + (learning_rate * nn->
            delta_hidden[i] * 1);
    }
```

1.2.2.6 Method load_input_values

Tal como o projeto anterior, esta função tem como objetivo guardar os valores presentes no vetor de entrada (vetor de entrada é um parâmetro da função) no array **I** da estrutura de dados, que guardará os valores das unidades entrada.

1.2.2.7 Method load_weights

Esta função tem como objetivo atribuir pesos aleatórios entre -0.5 e 0.5 às arestas da rede neural (ligações entre as unidades excluindo as ligações das bias às unidades, nesse caso definimos valores por default).

O primeiro loop vai atribuir pesos aleatórios às arestas que ligam a *input layer* e a *hidden layer* e o segundo loop atribuí pesos aleatórios às arestas que ligam a *hidden layer* e a *output layer*.

Listing 1.12: load_weights

```
void load_weights(NeuralNetwork *nn){
    // Seed the random number generator with the current time
    srand(time(NULL));

    // Assign random weights to the edges
    for (int i = 0; i < nn->number_I - 1; i++) {
        for (int j = 0; j < nn->number_H - 1; j++) {
            // Random value between -0.5 and 0.5
            nn->weights_to_hidden[i][j] = (((double)rand()/RAND_MAX) -0.5)*1.0;
        }
    }

    for (int i = 0; i < nn->number_H - 1; i++) {
        for (int j = 0; j < nn->number_O; j++) {
            // Random value between -0.5 and 0.5
            nn->weights_to_output[i][j] = (((double)rand()/RAND_MAX) -0.5)*1.0;
        }
    }
}
```

1.2.2.8 Method load_nn

Tal como no projeto anterior, esta função tem como objetivo carregar os pesos de uma rede neural a partir de um ficheiro de texto e atribuí-los às arestas da estrutura de dados.

A função começa por tentar abrir um ficheiro (neste exemplo chama-se NeuralNetwork1.txt) em modo leitura usando a função **fopen()** e caso o mesmo não possa ser aberto, será exibida no terminal uma mensagem de erro.

Posto isto, vai ser lido o texto presente no ficheiro. A primeira linha, que contêm a informação sobre o número de unidades de cada camada, é pulada porque na função **nn_create** essa linha já é lida para atribuir o número de unidades às respetivas *layers* na estrutura de dados da rede. Em seguida, vão ser lidas as linhas seguintes que seguem um formato específico, sendo que cada linha possui a estrutura: **layer:unit layer:unit peso**.

Enquanto as linhas estiverem neste formato, vão ser lidos os pesos das arestas que unem as respetivas unidades entre as *layers* e vão ser atribuídos esses pesos às respetivas variáveis da rede. Quando a próxima linha não estiver no formato específico, é terminada a leitura e fechado o ficheiro usando a função **fclose()**.

1.2.2.9 Method nn_destroy

Este método tem o mesmo objetivo do projeto anterior, é realizada a limpeza da memória libertando a memória alocada dinamicamente para a rede neural, ou seja, desaloca os arrays e matrizes associados à rede. A única diferença em relação ao projeto anterior é que como foram adicionados novos atributos à estrutura da rede neural a memória alocada para esses atributos também vai ter de ser libertada.

Listing 1.13: New atributes to be freed

```
free(nn->bias_to_output);
free(nn->bias_to_hidden);
free(nn->errors_output);
free(nn->delta_hidden);
free(nn->delta_output);
```

1.2.2.10 Method main

Na função **main** é onde está implementada a resolução do enunciado com a ajuda de um conjunto de funções que estão explicadas em cima.

Vai haver 2 modos de treino, o treino **"normal"** onde vão ser passados valores de entrada (para as unidades da camada principal) para a rede treinar os pesos de maneira a atingir valores de saída próximos/iguais aos desejáveis (definidos por default no código para este método de treino), cujo único objetivo deste modo de treino é verificar o funcionamente correto de treino da rede neural tendo valores de pesos iniciais fixos (lidos de um ficheiro) e as mudanças no desempenho de treino da rede alterando o *learning rate*. No segundo modo de treino e mais importante é o **XOR** que serve para treinar a rede neural para implementar a função booleana *XOR*.

Em primeiro lugar são extraidos os valores dos parâmetros-chave: maxIterations (iterações máximas de treino), learningRate (taxa de aprendizagem), *threshold* (limite para considerar uma saída válida) e xorMode (modo de treino).

Listing 1.14: main

```
if (argc < 5) {
    printf("Usage: %s [iterations] [learning_rate] [threshold] [training_mode
        (0|1)]\n", argv[0]);
    return 1;
}

int maxIterations = atoi(argv[1]);
double learningRate = atof(argv[2]);
double threshold = atof(argv[3]); // Threshold to stop the training, 0.0001
int xorMode = atoi(argv[4]); // 0 -> (normal training mode) | 1 - (XOR training
    mode)

if(xorMode != 0 && xorMode != 1){
    printf("Usage: [training_mode] needs to be 0 (normal mode) or 1 (xor mode)\n
        ");
    return 1;
}
```

Em seguida, o programa verifica se xorMode é 0 (treino normal) ou 1 (treino XOR) e vai inicializar a neural network.

Caso o **xorMode seja 0**, são inicializadas os arrays com os valores de entrada e saida por default e, em seguida são carregados os pesos do ficheiro para as arestas da rede . Depois são postos os valores de entrada na rede neural e, em seguida, são inicializados e carregados na rede neural os arrays que contém os valores dos pesos das ligações do bias da camada principal para a camada do meio , e o do bias da camada do meio para a camada de saída.

Listing 1.15: main

```
NeuralNetwork *nn = nn_create(xorMode);
double input_values[] = {2.0, 1.0};
double output_values[] = {0.7, 0.4}; //Desired values, could be changed (
    between 0 and 1 because its a sigmoid function)

load_nn(nn);

load_input_values(input_values, nn);

double bias_weights_input_to_hidden[] = {2.0, 1.0};
load_initial_bias_weights_to_hiddenLayer(bias_weights_input_to_hidden, nn);

double bias_weights_hidden_to_output[] = {1.0, 2.0};
load_initial_bias_weights_to_outputLayer(bias_weights_hidden_to_output, nn);
```

Posto isto, vai ser inicializado o loop de treino que irá continuar até o número máximo de iterações ou caso os valores de saída sejam próximos/iguais aos desejáveis. É chamada a funcão propagate para calcular os resultados de saída propagando os valores de entrada pela rede neural. Depois, calcula os erros entre a saída real e a saída desejada e conta o número de saídas que estão dentro do limite especificado.

Caso todas as saídas estejam dentro do limite, ou seja, todas as saídas estão próximas/iguais às saídas desejadas, o treino termina. Por outro lado, se todas as saídas não estiverem dentro do limite, o treino continua e a função *backpropagate* é chamada para realizar a retropropagação, ajustando os pesos com base no erro.

Listing 1.16: main

```
while(iteration < maxIterations){
    printf("Iteration number %d\n", iteration);

    propagate(nn);

    //Calculate error of the outputs
    for (int i = 0; i < nn->number_O; i++) {
        nn->errors_output[i] = output_values[i] - nn->O[i];
    }

    int number_outputs_within_threshold = 0;
    for(int i = 0; i < nn->number_O; i++){
        if(fabs(nn->errors_output[i]) <= threshold){
            number_outputs_within_threshold++;
        }
    }

    //All output are within the threshold limit
    if(number_outputs_within_threshold == nn->number_O){
        printf("The output values are close to the desired values within the
            threshold lime at the iteration: %d\n", iteration);
        break;
```

```
    }

    //If still not valid output for each output unit, backpropagate
    back_propagate(nn, learningRate);

    iteration++;
}
```

Depois do treino, os resultados finais são impressos e a rede neural é destruída para libertar a memória alocada. Por outro lado, caso o **xorMode seja 1**, são definidos os valores de entrada e os seus valores de saída desejados correspondentes para lógica XOR. Esses valores representam a tabela de verdade *XOR*.

Listing 1.17: main

```
double input_XOR_values[4][2] = {{0.0,0.0}, {0.0,1.0}, {1.0,0.0},
    {1.0,1.0}};
double output_XOR_values[4] = {0.0, 1.0, 1.0, 0.0};
```

Em seguida, é criada a rede neural, são atribuídos pesos aleatórios às arestas da rede e são carregados na rede neural os arrays que contêm os valores dos pesos das ligações do bias da camada principal para a camada do meio, e o do bias da camada do meio para a camada de saída (valores default). Por fim, é aberto o ficheiro *output.txt* para escrita onde vai guardar o desempenho da rede ao longo do tempo, ou seja, as iterações e o respetivo número de outputs considerados válidos para cada sequência de entrada (00,01,10,11) para posteriormente ser criado um gráfico que mostra o desempenho da rede (script python *graph_creation.py*).

Listing 1.18: main

```
NeuralNetwork *nn = nn_create(xorMode);
load_weights(nn);

double bias_weights_input_to_hidden[] = {0.3, 0.6}; //{2.0, 1.0}
load_initial_bias_weights_to_hiddenLayer(bias_weights_input_to_hidden, nn);

double bias_weights_hidden_to_output[] = {0.6, 0.3}; //{1.0, 2.0};
load_initial_bias_weights_to_outputLayer(bias_weights_hidden_to_output, nn);

FILE *outputFile;
outputFile = fopen("output.txt", "w");

if (outputFile == NULL) {
    printf("Error opening the output file.\n");
    return 1;
    }
```

Posto isto são criados dois loops, o loop externo que representa as iterações de treino, e o loop interno que processa cada sequência de valores de entrada (00,01,10,11). Para cada sequência de entrada, são carregados os valores da sequência na rede neural, é feita a propagação dos valores dessa sequência, é calculado o erro relativo ao valor desejável e o valor de saída, é verificado se a saída dessa sequência é válida, e, por fim, é feito *"backpropagate"* dos erros e a próxima sequência de entrada volta a repetir este processo. O treino continua até que o número máximo de iterações seja atingido ou para qualquer sequência de entrada o valor de saída é válido (4 valores válidos).

Listing 1.19: main

```
while(iteration < maxIterations){
    all_outputs_within_threshold = 0;
    for(int i = 0; i < 4; i++){ // 4 vectors of input values
        load_input_values(input_XOR_values[i], nn);
```

```
            propagate(nn);

            //Calculate error of the outputs
            // Its not necessary a for loop because the network just have 1 output!
            for (int j = 0; j < nn->number_O; j++) {
                nn->errors_output[j] = output_XOR_values[i] - nn->O[j];
            }

            for(int j = 0; j < nn->number_O; j++){
                if(fabs(nn->errors_output[j]) <= threshold ) {
                    all_outputs_within_threshold++;
                }
            }

            printf("[%d,%d] - %lf -> %.2lf\n", (int)input_XOR_values[i][0], (int)
                input_XOR_values[i][1], nn->O[0], nn->O[0]);

            back_propagate(nn, learningRate);
        }
        fprintf(outputFile, "%d %d\n", iteration, all_outputs_within_threshold);

        //The outputs for each input vector are within the threshold limit
        if(all_outputs_within_threshold == 4){
            printf("The output values are close to the desired values within the
                threshold lime at the iteration: %d\n", iteration);
            break;
        }

        iteration++;
    }
```

Por outras palavras, o treino continua até haver um consenso entre as arestas e as sequências de entrada, isto é, o peso das arestas vai ser treinado até possuir um valor que, para qualquer sequência de entrada (00, 01, 10, 11), a saída é sempre válida, ou seja, próxima de "0" ou "1" dependendo dos valores de entrada (Algorítmo *XOR*).

1.3 Results

Neste capítulo vão ser mostrados os resultados do nosso projeto e como os mesmos variam tendo em conta um conjunto de fatores (learning rate, variação nos pesos iniciais das arestas da rede neural...). Além disso, ainda vai ser apresentado um conjunto de gráficos que mostram o progresso da rede neural ao longo do tempo.

1.3.1 Normal Training

Para este exemplo vamos usar como default os valores de entrada de 2.0 e 1.0 e como saída de 0.7 e 0.4. Em seguida, vamos correr o comando abaixo onde vamos passar, como argumentos de entrada, o valor 100 000 que corresponde às iterações máximas, 0.1 que corresponde ao *learning rate*, 0.000001 que corresponde ao threshold e 0 que corresponde ao modo de treino normal. A secção **User Manual** explica estes parâmetros de entrada em detalhe.

```
./nn_base 100000 0.1 0.000001 0
```

Após correr o programa obtemos os seguintes resultados:

```
The output values are close to the desired values within the threshold lime at the iteration: 1603
Value of the final Output[0]: 0.700001
Value of the final Output[1]: 0.400001
```

Figure 1.2: Resultados com Learning Rate de 0.1

Tal como é possível verificar pela imagem de cima, a rede conseguiu treinar os pesos das arestas de maneira aos resultados serem iguais/próximos aos valores desejáveis, definidos inicialmente neste exemplo como sendo 0.7 e 0.4, na iteração 1603 (dá 0.700001 e 0.400001 por causa do threshold, sem o threshold os valores iriam ser iguais).

Posto isto, vamos correr o programa com um *learning rate* de 0.3 de maneira a verificar o que acontece ao processo de treino com um *learning rate* maior.

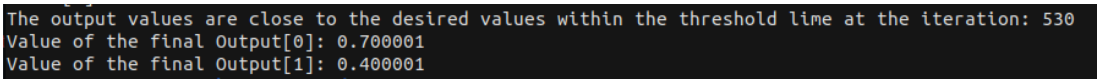


Figure 1.3: Resultados com Learning Rate de 0.3

De acordo com a imagem, é possível verificar que o processo de treino foi mais rápido, a rede aprendeu os mesmos resultados de saída na iteração 530.

Portanto, é possível afirmar que ao aumentar o learning rate, numa rede onde os pesos iniciais das arestas são sempre os mesmos (neste exemplo, os pesos foram carregados de um ficheiro), o processo de treino é mais rápido.

1.3.2 XOR Training

Input 1	**Input 2**	**Output**
0	0	0
0	1	1
1	0	1
1	1	0

Table 1.1: XOR Algorithm

Neste exemplo, para treinar o algorítmo *XOR*, usamos duas unidades de entrada (00,01,10,11), dez unidades no meio (podem ser mais ou menos, basta alterar o número de unidades do meio no ficheiro *XOR_ NeuralNetwork.txt*) e uma unidade de saída (0 ou 1).

```
./nn_base 100000 0.2 0.01 1
```

Após executar o comando de cima no terminal, obtemos os seguintes resultados:

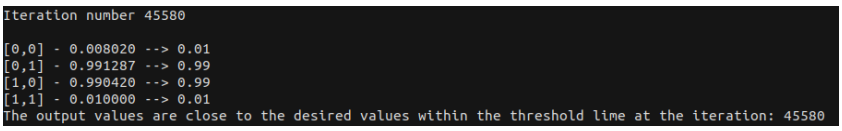


Figure 1.4: Resultados com Learning Rate de 0.2 e pesos iniciais entre -0.5 e 0.5

Como é possível verificar pela imagem de cima e pela **tabela 1.1**, a rede conseguiu treinar os seus pesos para implementar o algorítmo XOR. De salientar que os resultados não são exatamente iguais ao desejável (0 ou 1) porque trata-se de uma função sigmoid daí passar-mos um *threshold* de 0.01.

Além disso, criámos um gráfico que mostra o processo de treino ao longo do tempo, isto é, a cada iteração de treino para cada sequência, mostra os erros, considerando que um "erro" é quando dado uma sequência de entrada (00, 01, 10, 11) o valor do *output* está longe do desejável, ou seja, não é considerado válido (fora do threshold).

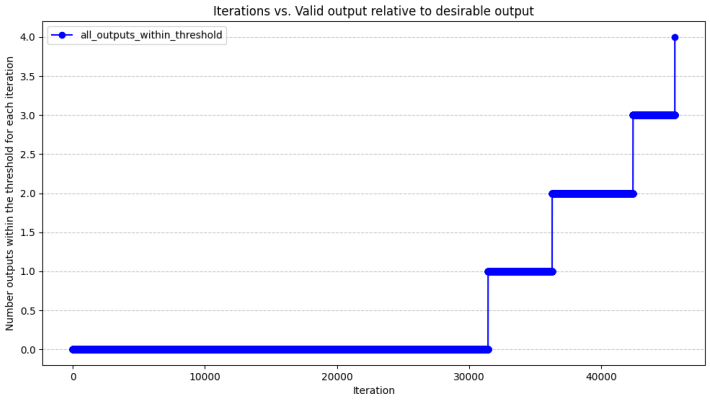


Figure 1.5: Processo de treino com learning rate de 0.2 e pesos iniciais entre -0.5 e 0.5

Pelo gráfico, é possível verificar que inicialmente não havia nenhum valor de output válido dado as quatro sequências de entrada e as respetivas saídas, 4 erros. A partir da iteração 3000 é possível verificar uma diminuição dos erros, isto é, o número de *outputs* válidos começou a aumentar ao ponto de chegar à iteração 45580 e haver 4 outputs válidos, ou seja, para qualquer sequência de entrada (00, 01, 10, 11) o *output* era considerado válido, isto é, próximo do desejável, dentro do limite do *threshold*.

A alteração do learning rate para 0.4 também mostra, tal como foi dito na secção **Normal Training**, que o processo de treino é mais rápido se o learning rate for maior. O gráfico abaixo comprova esta afirmação.

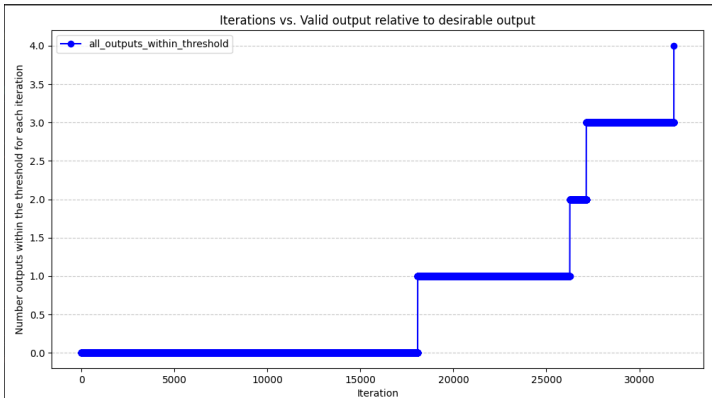


Figure 1.6: Processo de treino com learning rate de 0.4 e pesos iniciais entre -0.5 e 0.5

Em seguida, alterámos o valor inicial dos pesos das arestas na função **load_weights** para vermos as diferenças no desempenho da rede neural. Em vez de atribuir inicialmente pesos aleatórios entre -0.5 e 0.5, passámos a atribuir pesos entre -3.0 e 3.0 obtendo os seguintes resultados.

```
Iteration number 68452

[0,0] - 0.004785 --> 0.00
[0,1] - 0.990880 --> 0.99
[1,0] - 0.992888 --> 0.99
[1,1] - 0.010000 --> 0.01
The output values are close to the desired values within the threshold lime at the iteration: 68452
```

Figure 1.7: Resultados com learning rate de 0.2 e pesos iniciais entre -3 e 3

Pode-se ver pelos resultados de cima, que o processo de treino demorou um pouco mais com a inicialização do peso das arestas entre -3.0 e 3.0 em relação à inicialização dos pesos entre -0.5 e 0.5. De salientar que, isto não significa que ao aumentar sempre o *"range"* dos pesos iniciais das arestas vá aumentar a duração do processo de treino uma vez que neste exemplo a inicialização dos pesos é *random*, mas sem dúvida que é um fator a ter em conta para avaliar o desempenho de treino de uma rede neural.

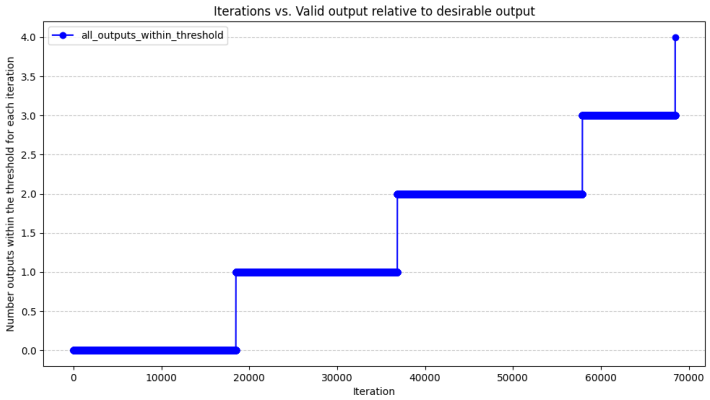


Figure 1.8: Processo de treino com learning rate de 0.2 e pesos iniciais entre -3 e 3

O gráfico de cima também comprova o que foi dito anteriormente, isto é, que o processo de treino demorou ligeiramente mais com o aumento do *"range"* dos pesos iniciais das arestas.

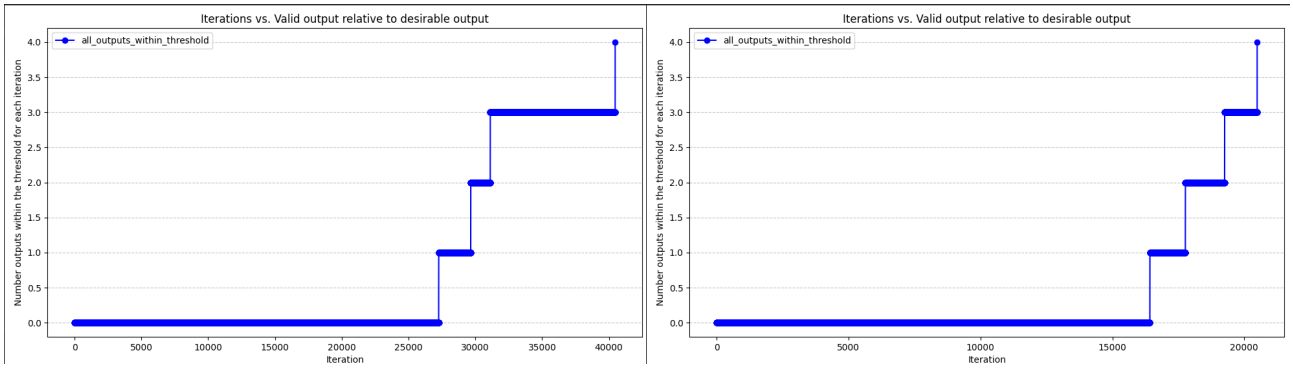
Posto isto, decidimos avaliar o desempenho da rede neural ao alterar o número de unidades da *hidden layer*. Alterámos o número de unidades da camada do meio para 3 em vez de 10 unidades, usámos o mesmo *learning rate* (0.4), o mesmo *threshold* (0.01) e executámos o programa 5 vezes com o objetivo de analisar os resultados e a performance no treino. A tabela em baixo, mostra os resultados obtidos.

	3 Hidden Units	**10 Hidden Units**
1	40466	20488
2	73673	26629
3	42037	31249
4	77462	26114
5	47267	44291

Table 1.2: Tabela com o número da iteração após a conclusão do treino para as 4 sequências

Tal como é possível ver pela tabela de cima, o processo de treino da rede tem melhor desempenho com um número maior de unidades na camada do meio, isto é, aprende mais rápido o valor desejável.

Em baixo, também é possível ver o resultado do processo de aprendizagem durante o tempo referente à primeira linha da tabela para cada número de unidades na camada do meio.



(a) 3 hidden units

(b) 10 hidden units

Figure 1.9: Processo de Treino

Concluindo, podemos afirmar que há um conjunto diverso de fatores que podem afetar o desempenho no treino das redes neurais, tal como foi comprovado pelos resultados deste capítulo. Aumentar o learning rate, aumentar o número de unidades na camada do meio e diminuir o intevalo de pesos iniciais para as arestas são um exemplo de fatores que fazem com que o tempo de treino da rede diminua para atingir um valor de saída próximo ao desejável.

É importante salientar que o impacto destas alterações depende muito da estrutura da rede neural e que existem muitos mais fatores que podem alterar a performance da rede neural, como por exemplo, a alteração da função de ativação.

Chapter 2

Author's Contribution

Todos participaram de forma igual na divisão e elaboração deste projeto, pelo que a percentagem de contribuição de cada aluno fica:

- João Torrinhas - 50%
- Diogo Torrinhas - 50%

Bibliography

[1] Dustin Stansbury, *Derivation of Backpropagation*. Available online: `https://dustinstansbury.github.io/theclevermachine/derivation-backpropagation`

[2] Towards Data Science, *Derivative of the Sigmoid Function*. Available online: `https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e`