Projeto 3 - EP

Universidade de Aveiro

João Torrinhas, Diogo Torrinhas

Projeto 3 - EP

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

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

December 30, 2023

Contents

Chapter 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 `TicTacToe_Game.c`.
2. Após estar no diretório correto, use o seguinte comando no terminal para compilar o programa:

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

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

   ```
   ./TicTacToe_Game [game_mode (m|b|a|t)] [seed (optional)] [board (optional)]
   ```

   [game_mode] - Corresponde ao modo de jogo que o utilizador escolher, **m** é 1v1, ou seja, o utilizador joga contra ele mesmo ou contra outra pessoa no mesmo computador, **b** o utilizador vai jogar contra um *random agent* (bot),na opção **a** o utilizador vai jogar contra um agente treinado e, finalmente, na opção **t** a rede vai ser treinada a partir de um ficheiro com os exemplos dos jogos, *games.txt*.

   [seed (optional)] - Corresponde ao valor da *seed* usada para o *random agent*. É opcional e portanto não é necessário passar como argumento na linha de comandos.

   [board (optional)] - Corresponde ao estado predefinido do tabuleiro. O utilizador se quiser pode começar o jogo com o tabuleiro num estado predefinido. Tal como o parâmetro anterior, este também é opcional, o utilizador pode escolher se quer começar o jogo com o tabuleiro vazio ou num estado predefinido.

   O comando de cima vai correr o programa compilado `TicTacToe_Game` e vai ser possível ver no terminal o resultado do programa.

Chapter 2

Code Explanation

Neste capítulo vai ser explicado em detalhe todo o código desenvolvido neste projeto à exceção do código relativo à rede neural devido a já ter sido explicado nos projetos anteriores.

2.1 Method initializeBoard

Esta função tem como objetivo inicializar o tabuleiro. Em cada posição do array *board* vai ser atribuído o valor ' '.

2.2 Method printBoard

Esta função *"printa"* o estado atual do tabuleiro percorrendo cada posição do array *board* e *"printando"* o valor contido na mesma.

2.3 Method isValidMove

Esta função recebe como parâmetros a coluna e a linha e verifica se essa posição é válida. A posição será considerada válida se o valor da linha e da coluna, ou seja a posição no tabuleiro, estiver dentro do tabuleiro e se essa posição está livre (' ').

O código abaixo mostra o processo de validação descrito acima.

Listing 2.1: isValidMove

```
int isValidMove(int row, int col){
    if (row >= 0 && row<BOARD_SIZE && col>=0 && col<BOARD_SIZE && board[row][col]== ' '){
        return 1;
    }else{
        return 0;
    }
}
```

2.4 Method getEmptyPositions

Esta função retorna um array com as posições livres no tabuleiro (indíces da posição livre, i.e, posição 0,1,2...8) percorrendo todas as posições do mesmo e verificando se nessa posição tem o valor ' '. Caso a posição tenha esse valor, significa que está livre e é adicionada ao array **emptyPositions**.

No fim, a função retorna o array **emptyPositions**. Em baixo, é possível ver o código que comprova o que foi explicado em cima.

Listing 2.2: getEmptyPositions

```
int* getEmptyPositions(int* count) {
    int* emptyPositions = (int*)malloc(BOARD_SIZE * BOARD_SIZE * sizeof(int));
    *count = 0;

    for (int i = 0; i < BOARD_SIZE; ++i) {
        for (int j = 0; j < BOARD_SIZE; ++j) {
            if (board[i][j] == ' ') {
                emptyPositions[(*count)++] = i * BOARD_SIZE + j;
            }
        }
    }
    return emptyPositions;
}
```

2.5 Method getEmptyPositionsCount

Esta função tem um funcionamente semelhante à função anterior. Neste caso retorna o **número** de posições livres no tabuleiro.

2.6 Method checkTie

Esta função retorna um número inteiro(1 ou 0) que indica se o jogo é empate ou não. A estrutura principal da função é composta por dois loops, controlados pelas variáveis i e j, que percorrem todas as posições do tabuleiro. Dentro desses loops, há uma verificação que examina se o caracter na posição (i, j) do tabuleiro é um espaço em branco (' '), indicando que a célula está vazia. Se uma célula vazia é encontrada, a variável tie é alterada para 0, denotando que o jogo não está empatado, e o loop interno é encerrado com a instrução *break*.

Finalmente, a função retorna o valor atual de tie. Se for 1 é empate, se for 0 significa que ainda existe uma célula vazia e que ainda não terminou.

Listing 2.3: checkTie

```
int checkTie(){
    int tie = 1;
    for(int i = 0; i < BOARD_SIZE; i++){
        for(int j = 0; j < BOARD_SIZE; j++){
            if(board[i][j] == ' '){
                tie = 0;
                break;
            }
        }
    }
    return tie;
}
```

2.7 Methot checkWinner

A função checkWinner é responsável por determinar se um jogador específico é o vencedor do jogo, avaliando as sequências de seu símbolo (representado pelo parâmetro player) nas linhas, colunas e diagonais do tabuleiro.

É feita primeiro a verificação das linhas. Dentro do loop, verifica se há três células consecutivas na linha atual, dentro do loop, ocupadas pelo símbolo do jogador. Se essa condição for atendida, a variável winner é definida como 1, indicando a presença de um vencedor.

Em seguida, dentro do loop, há uma verificação semelhante para determinar se há três células consecutivas na coluna atual ocupadas pelo símbolo do jogador. Se essa condição for satisfeita, a variável winner é definida

como 1, indicando a presença de um vencedor.

Por fim são verificadas as diagonais, se uma das diagonais tiver três células consecutivas ocupadas pelo símbolo do jogador, a variável winner é definida como 1, indicando a presença de um vencedor.

Se nenhuma das condições anteriores for atendida, a função retorna o valor padrão de winner, que é 0, indicando a ausência de um vencedor.

Listing 2.4: checkWinner

```
//Check if there is a winner
int checkWinner(char player){
    int winner = 0;
    //check rows
    for(int i = 0; i < BOARD_SIZE; i++){
        if(board[i][0] == player && board[i][1] == player && board[i][2] == player){
            winner = 1;
            return winner;
        }
    }
    //check columns
    for(int i = 0; i < BOARD_SIZE; i++){
        if(board[0][i] == player && board[1][i] == player && board[2][i] == player){
            winner = 1;
            return winner;
        }
    }
    //check diagonals
    if(board[0][0] == player && board[1][1] == player && board[2][2] == player){
        winner = 1;
        return winner;
    }
    if(board[0][2] == player && board[1][1] == player && board[2][0] == player){
        winner = 1;
        return winner;
    }
    return winner;
}
```

2.8 Methot get_higher_value_pos

Esta função tem como objetivo encontrar o índice da posição que contém o valor mais alto no vetor nn->O, que representa as saídas da rede neural.

É usado um loop for para iterar sobre as saídas da rede neural (nn->O) e há uma verificação para determinar se o índice atual (i) está presente no vetor pos que contêm os indíces mais altos que não são válidos, isto é, indíces cujas posições no tabuleiro não são válidas, estão ocupadas. Se estiver presente, esse índice é excluído das considerações de maneira a procurar por outro indíce cuja posição é válida.

Se o índice atual não estiver presente no vetor pos, a função compara o valor correspondente em nn->O[i] com o valor máximo atual (max). Se nn->O[i] for maior que max, o valor máximo é atualizado, e *higher_value_index* é definido como o índice atual (i).

Por fim, é retornado *higher_value_index*, que representa o índice da posição com o valor mais alto entre aqueles não excluídos.

Listing 2.5: get_higher_value_pos

```
int get_higher_value_pos(NeuralNetwork *nn, int *pos, int number_iterations){
    int higher_value_index = -1;
    double max = 0.0;
```

```
    for (int i = 0; i < nn->number_O; i++) {
        int skip = 0;
        // Check if the current index i is in the pos array
        for (int j = 0; j < number_iterations; j++) {
            if (i == pos[j]) {
                skip = 1;
                break;
            }
        }
        // Skip this iteration if i is in the pos array
        if (skip) {
            continue;
        }
        // Find the maximum value among non-excluded indices
        if (nn->O[i] > max) {
            max = nn->O[i];
            higher_value_index = i;
        }
    }
    return higher_value_index;
}
```

2.9 Methot bot_random_generator_move

Esta função representa um movimento aleatório realizado por um agente/bot aleatório em um jogo e retorna o valor *win* a indicar se o agente ganhou após a sua jogada.

Inicialmente, vê de acordo com a escolha do utilizador, *choice*, qual é o símbolo que vai corresponder ao *random_ agent* (depende qual é a vez do agente aleatório jogar).

Posto isto, vê quais são as posições livres do tabuleiro e gera um índice aleatório (randomIndex) dentro dessas posições livres utilizando a função rand() e o operador de módulo %.

Em seguida, obtém a posição correspondente no vetor de posições vazias usando *emptyPositions[randomIndex]* e calcula as coordenadas dessa posição no tabuleiro.

Depois, atualiza a posição correspondente no tabuleiro com o símbolo do *random_ agent* e imprime no terminal a jogada realizada pelo agente.

Por fim chama a função *checkWinner(currentPlayer)* para verificar se o agente venceu após realizar a jogada e liberta a memória alocada para o vetor de posições vazias utilizando *free(emptyPositions)*.

Listing 2.6: bot_random_generator_move

```
int bot_random_generator_move(char choice){
    char currentPlayer;
    if (choice == 'n'){
        currentPlayer = 'X';
    }else if(choice == 'y'){
        currentPlayer = 'O';
    }
    int win = 0;
    int count;
    int* emptyPositions = getEmptyPositions(&count);
    if(count != 0){
        int randomIndex = rand() % count;
        int position = emptyPositions[randomIndex];
        int row = position / BOARD_SIZE;
        int col = position % BOARD_SIZE;
        board[row][col] = currentPlayer;
```

```
            printf("\n");
            printf("Bot move (%c): %d, %d\n", currentPlayer, row, col);
        }

        if (checkWinner(currentPlayer)) {
            printf("Player %c wins!\n", currentPlayer);
            win = 1;
        }
        free(emptyPositions);
        return win;
}
```

2.10 Methot agent_move

Esta função tem como objetivo representar a jogada de um agente inteligente em um jogo, onde o agente toma decisões com base numa rede neural treinada representada pela estrutura NeuralNetwork e retorna o valor *win* a indicar se o agente ganhou após a sua jogada.

Inicialmente, vê de acordo com a escolha do utilizador, *choice*, qual é o símbolo que vai corresponder ao agente inteligente e de seguida aloca memória para o array *index_higher* que guarda o index das posições com maior valor de saída da rede (nn->O[i] que corresponde à posição i do tabuleiro) mas que são inválidas porque não estão livres.

Posto isto, verifica se há mais de uma posição vazia (count > 1). Se houver, utiliza um loop para obter o índice do valor mais alto na saída da rede neural (posição preferida para a próxima jogada) com base na função *get_higher_value_pos* e verifica se o índice com o maior valor (higher_pos) está em uma posição vazia. Se o índice com o maior valor estiver em uma posição vazia o programa sai do ciclo while (*break*) caso contrário, verifica que se o número de iterações já atingiu o limite (9), caso não tenha atingido o limite, adiciona esse índice com a posição mais alta inválida (ocupada) no array *index_higher*.

Caso tenha atingido o limite, ou seja, por alguma razão, ainda que **EXTREMAMENTE rara** (fizemos testes manuais e reparámos que poderia acontecer esta possibilidade), não existe um valor mais alto cuja posição é válida/livre (por exemplo, um cenário onde das 9 posições havia 7 delas que eram as mais altas, umas mais altas que outras, mas estavam ocupadas e as restantes é tudo 0.0) ele vai escolher uma posição random das posições livres existentes. Optámos por cobrir esta possibilidade para evitar erros no programa de modo a certificarmo-nos que o programa é 100% contra erros.

Por outro lado, se apenas existe uma posição vazia (count == 1), ele seleciona o índice dessa posição para a sua jogada.

Em seguida calcula as coordenadas da posição no tabuleiro com base no índice escolhido, atualiza a posição correspondente no tabuleiro com o símbolo do jogador atual e imprime no terminal a jogada realizada pelo agente.

Posto isto, é chamada a função checkWinner para verificar se a jogada resultou em vitória para o agente inteligente e retorna o valor de win, indicando se a jogada resultou em vitória.

Listing 2.7: Agent_move

```
int agent_move(NeuralNetwork *nn, char choice){
    char currentPlayer;
    if (choice == 'n'){
        currentPlayer = 'X';
    }else if(choice == 'y'){
        currentPlayer = 'O';
    }

    int win = 0;
    int count;
    int *emptyPositions = getEmptyPositions(&count);
```

```
    //int index_higher = -1;
    int *index_higher = (int*)malloc(9 * sizeof(int));
    if (index_higher == NULL) {
        fprintf(stderr, "Error allocating memory for index_higher variable\n");
        exit(EXIT_FAILURE);
    }
    int empty_pos = 0;
    int higher_pos;
    int number_iterations = 0;

    if(count > 1){ //count != 0
        while(1){
            //Obtain the index of the higher value in the output layer
            higher_pos = get_higher_value_pos(nn, index_higher, number_iterations);

            for(int i = 0; i < count; i++){
                // Check if the higher value index is an empty position
                if(higher_pos == emptyPositions[i]){
                    empty_pos = 1;
                    break;
                }
            }

            if(empty_pos == 1){
                break;
            }else if(number_iterations == 9){
                break;
            }else{
                index_higher[number_iterations++] = higher_pos;
            }
        }

        if(number_iterations == 9){
            int randomIndex = rand() % count;
            higher_pos = emptyPositions[randomIndex];
        }
        int row = higher_pos / BOARD_SIZE;
        int col = higher_pos % BOARD_SIZE;
        board[row][col] = currentPlayer;
        printf("\n");
        printf("Bot move (%c): %d, %d\n", currentPlayer, row, col);
        if (checkWinner(currentPlayer)) {
            printf("Player %c wins!\n", currentPlayer);
            win = 1;
        }
    }else if(count == 1){
        int row = emptyPositions[0] / BOARD_SIZE;
        int col = emptyPositions[0] % BOARD_SIZE;
        board[row][col] = currentPlayer;
        printf("\n");
        printf("Bot move (%c): %d, %d\n", currentPlayer, row, col);
        if (checkWinner(currentPlayer)) {
            printf("Player %c wins!\n", currentPlayer);
            win = 1;
        }
    }
    free(index_higher);
    free(emptyPositions);
    return win;
}
```

2.11 Methot isNumber

Esta função serve para verificar se o array de caracteres passado como parâmetro é um número válido e não uma string normal.

Caso o parâmetro de entrada seja uma string, por exemplo, "Ola" retorna False, caso seja "10" retorna True.

2.12 Methot predefinedBoardState

Esta função tem como objetivo configurar o estado inicial do tabuleiro com base numa string fornecida (boardState).

Inicialmente é feita uma verificação para garantir que o tamanho da string fornecida como parâmetro tem um tamanho de tabuleiro válido, ou seja, é igual a 9.

Em seguida, são usados dois loops para percorrer cada posição do tabuleiro e obtém o caracter atual da string *boardState* usando o índice k que posteriormente é incrementado.

Posto isto, verifica se o caracter atual é um dos símbolos válidos: 'X', 'O' ou '-'. Depois, se o caracter atual não for '-', configura a posição correspondente no tabuleiro com o caractere atual, caso contrário, se for '-', configura a posição correspondente no tabuleiro com um espaço em branco (' ').

Listing 2.8: predefinedBoardState

```
void predefinedBoardState(char *boardState){
    int k = 0;

    // Check if the length of the board configuration is valid
    if (strlen(boardState) != BOARD_SIZE * BOARD_SIZE) {
        fprintf(stderr, "Invalid board configuration size.\n");
        exit(1);
    }

    for (int i = 0; i < BOARD_SIZE; i++) {
        for (int j = 0; j < BOARD_SIZE; j++) {
            char currentChar = boardState[k++];

            // Check if the character is a valid symbol (X, O, or -)
            if (currentChar != 'X' && currentChar != 'O' && currentChar != '-') {
                fprintf(stderr, "Invalid character in board configuration: %c\n"
                , currentChar);
                exit(1);
            }

            if(currentChar != '-'){
                board[i][j] = currentChar;
            }else{
                board[i][j] = ' ';
            }
        }
    }
}
```

2.13 Methot writeBoardToFile

Esta função tem como objetivo escrever o estado atual do tabuleiro em um ficheiro de texto.

Inicialmente percorre todas as posições do tabuleiro e vai verificar se o conteúdo da posição atual no tabuleiro é um espaço em branco (' '). Se for um espaço em branco, escreve o caracter '-' no ficheiro caso contrário, escreve o caracter encontrado no tabuleiro.

Após percorrer todas as posições do tabuleiro e escrever o estado atual do tabuleiro no ficheiro, escreve um espaço em branco no ficheiro para separar os diferentes estados do tabuleiro.

Listing 2.9: writeBoardToFile

```
void writeBoardToFile(FILE *file) {
    for (int i = 0; i < BOARD_SIZE; i++) {
        for (int j = 0; j < BOARD_SIZE; j++) {
            if(board[i][j] == ' '){
                fprintf(file, "%c", '-');
            }else{
                fprintf(file, "%c", board[i][j]);
            }
        }
    }
    fprintf(file, " ");
}
```

2.14 Methot inputStringToNeuralNetwork

A função *inputStringToNeuralNetwork* tem como objetivo converter uma string que representa o estado atual do tabuleiro em uma representação que vai servir como entrada para a rede neural.

Inicialmente utiliza um loop para percorrer cada caracter da string *boardState*. Se o caracter atual é 'X', atribui o valor 1 à posição correspondente no array de inputs. Se o caracter atual é 'O', atribui o valor -1 à posição correspondente no array de inputs e se o caracter atual é qualquer outro (' '), atribui o valor 0 à posição correspondente no array de inputs.

Listing 2.10: inputStringToNeuralNetwork

```
void inputStringToNeuralNetwork(char *boardState, double *inputs) {
    int index = 0;

    for (int i = 0; i < 9; i++) {
        if (boardState[i] == 'X') {
            inputs[index++] = 1;
        } else if (boardState[i] == 'O') {
            inputs[index++] = -1;
        } else {
            inputs[index++] = 0;
        }
    }
}
```

2.15 Methot giveDiseredOutputForNeuralNetwork

Esta função tem como objetivo transformar o array de caracteres *boardNextState* num formato específico para ser usado como um target/output desejável na rede neural comparando o estado atual e o próximo estado (próxima jogada desejável).

Inicialmente verifica se o primeiro caracter de *boardNextState* é o caracter nulo ('0'). Isso indica que o *boardNextState* está vazio, portanto, *boardState* é o último estado do ficheiro de jogos.

Se está vazio, atribui zero a todas as posições do array de saídas desejadas. Se *boardNextState* não está vazio, compara cada posição de *boardState* (jogada atual) com a correspondente em *boardNextState* (próxima

jogada tendo em conta a jogada/estado atual): se a posição é igual nos dois estados, atribui "0" à posição correspondente no array de saídas desejadas, se a posição é diferente nos dois estados, atribui "1" à posição correspondente no array de saídas desejadas.

Por fim, o array *desired_outputs* estará preenchido com os valores correspondentes às saídas desejadas para a rede neural com base no estados atual(jogada atual) e no próximo estado (próxima jogada ótima tendo em conta a jogada atual).

Listing 2.11: giveDiseredOutputForNeuralNetwork

```
void giveDiseredOutputForNeuralNetwork(char *boardState, char *boardNextState
, double *desired_outputs){
    int index = 0;

    if(boardNextState[0] == '\0'){
        for(int i = 0; i < 10; i++){
            desired_outputs[index++] = 0;
        }
    }else{
        for(int i = 0; i < 9; i++){
            if(boardState[i] == boardNextState[i]){
                desired_outputs[index++] = 0;
            }else if(boardState[i] != boardNextState[i]){
                desired_outputs[index++] = 1;
            }
        }
    }
}
```

2.16 Methot checkIfLastBoardStateForCurrentGame

Esta função verifica se o estado *boardState* do tabuleiro é o estado inicial do tabuleiro, ou seja, todo o tabuleiro está vazio. Esta função vai receber como parâmetro o estado seguinte para verificar se o mesmo está vazio (atríbuto *boardState*), ou seja, tabuleiro contêm 9 caracteres '-', se estiver vazio significa que o estado atual é o último do jogo.

Utiliza um loop para percorrer cada posição do tabuleiro (*boardState*). Se o caracter na posição atual for '-', incrementa o valor de count, que conta o número de posições vazias no tabuleiro.

Depois, verifica se o valor de count é igual a 9, o que indica que todas as posições do tabuleiro são vazias. Se todas as posições do tabuleiro são vazias, a função retorna 1, indicando que é o primeiro estado do jogo. Caso contrário, retorna 0.

Listing 2.12: checkIfLastBoardStateForCurrentGame

```
int checkIfLastBoardStateForCurrentGame(char *boardState){
    int count = 0;

    for(int i = 0; i < 9; i++){
        if(boardState[i] == '-'){
            count++;
        }
    }

    if(count == 9){
        return 1;
    }else{
        return 0;
    }
}
```

2.17 Methot getCurrentBoardState

Esta função aloca dinamicamente memória para armazenar o estado atual do tabuleiro e retorna uma string que representa esse estado.

Inicialmente, utiliza a função malloc para alocar dinamicamente um array de caracteres (char) com espaço para 10 elementos (9 posições do tabuleiro + 1 para o caracter nulo '0').

Em seguida, utiliza dois loops para percorrer cada posição do tabuleiro (board). Se a posição no tabuleiro contiver um espaço em branco (' '), atribui '-' à posição correspondente em boardState. Caso contrário, atribui o caracter encontrado na posição do tabuleiro a boardState. Depois adiciona o caracter nulo ao final do array boardState para indicar o final da string.

Por fim retorna o ponteiro para a string boardState, que agora contém o estado atual do tabuleiro.

Listing 2.13: checkIfLastBoardStateForCurrentGame

```
char *getCurrentBoardState(){
    char *boardState = malloc(10 * sizeof(char));
    int index = 0;

    for(int i = 0; i < BOARD_SIZE; i++){
        for(int j = 0; j < BOARD_SIZE; j++){
            if(board[i][j] == ' '){
                boardState[index++] = '-';
            }else{
                boardState[index++] = board[i][j];
            }
        }
    }
    boardState[index] = '\0';
    return boardState;
}
```

2.18 Method main

Esta função é o coração do programa, é onde são feitas as decisões tendo em conta os comandos introduzidos pelo o utilizador no terminal, onde está o código que permite ao programa correr os diversos modos de jogos e onde o utilizador pode treinar uma rede neural tendo em conta os exemplos de jogos presentes num ficheiro txt.

Inicialmente esta função vai fazer um conjunto de verificações para o programa certificar-se que o utilizador insere corretamente os comandos e respetivos parâmetros no terminal. Caso insira alguma coisa mal, o programa irá emitir uma mensagem de erro no terminal.

Em seguida, há um conjunto de *ifs statements* para cada *game_mode* introduzido no terminal (m|b|a|t), obtido da seguinte maneira:

```
char game_mode  = argv[1][0];
```

Cada um desses *ifs statement* vai executar um determinado conjunto de instruções para jogar um tipo de jogo ou efetuar o treino, tendo em conta o *game_mode*. Caso o *game_mode* seja **m**, **b** ou **a** vão ser executadas as instruções para ser jogado um tipo de jogo (1v1, contra um *random agent* ou contra um agente inteligente), caso seja **t** é efetuado o treino da rede através de um ficheiro com exemplos de jogos.

O processo para a execução dos jogos é semelhante para cada *game_mode* (m|b|a), se o utilizador jogar primeiro é lhe pedido que insira uma posição, é feita a validação dessa posição, a seguir, é verificado se existe vencedor ou se há um empate e caso não haja é a vez do próximo jogador jogar (outro utilizador, o *random agent* ou o agente inteligente). Se o utilizador não quiser jogar em primeiro lugar, o processo é semelhante à exceção que é o outro jogador que joga primeiro.

Nas próximas secções vão ser explicados cada um destes *ifs* em detalhe. Após os mesmos serem executados, o programa termina.

2.18.1 If game_mode equal to m

Por exemplo, caso o utilizador tenha inserido no terminal o comando **./TicTacToe_Game m**, este if vai ser executado.

Vai ser aberto um ficheiro em modo *append* para escrever no ficheiro *games.txt* os exemplos de jogos que posteriormente vão ser treinados no modo **t** (**Nota:** Caso não queira escrever os jogos no ficheiro basta comentar as linhas que dizem respeito à escrita do ficheiro).

Listing 2.14: Abrir o ficheiro em modo append

```
FILE *fp;
fp = fopen("games.txt", "a");

if (fp == NULL) {
    fprintf(stderr, "Error opening the file.\n");
    return 1;
}
```

Em seguida, inicialmente, dentro de um ciclo while infinito, é "printado" no terminal o estado atual do tabuleiro e é obtido o número de posições livres no tabuleiro e caso esse número seja igual a 9 (todas as posições livres), escreve o primeiro estado do tabuleiro numa linha no ficheiro. Depois, no terminal, através da função *scanf()* é pedido que o utilizador introduza uma posição do tabuleiro onde quer jogar (valor de uma coluna e de uma linha) e é feita a validação dessa posição. Caso essa posição seja inválida, é emitida no terminal uma mensagem de erro, caso contrário é introduzida no tabuleiro a posição que o utilizador escolheu e é escrito no ficheiro de jogos o estado atual do tabuleiro na mesma linha (próximo jogo já escreve na linha seguinte).

Posto isto, o programa verifica se o jogo terminou, isto é, se há um vencedor ou se o resultado do jogo foi um empate. Caso haja um vencedor ou um empate, o programa sai do ciclo while (break), caso contrário o programa verifica quem é que jogou e dá a vez para o outro jogador e volta a ser repetido todo o processo até que haja um vencedor ou um empate.

No fim, após haver um vencedor ou um empate, o ficheiro é fechado. O excerto de código abaixo mostra todo este processo.

Listing 2.15: While loop

```
while(1){
    printBoard();

    count = getEmptyPositionsCount();
    if (count == 9) {
        // write the first state of the board
        writeBoardToFile(fp);
    }

    printf("Player %c, enter row and column (e.g, 0 0 or 0 1): ", currentPlayer);
    scanf("%d %d", &row, &col);

    // Validate the move
    if (!isValidMove(row, col)) {
        printf("Invalid move! Try again.\n");
        continue;
    }

    board[row][col] = currentPlayer;
    writeBoardToFile(fp);
```

```
    // Check if the game is over (winner or tie)
    if (checkWinner(currentPlayer)) {
        printBoard();
        printf("Player %c wins!\n", currentPlayer);
        break;
    }else if(checkTie()){
        printBoard();
        printf("Tie!\n");
        break;
    }

    if(currentPlayer == 'X'){
        currentPlayer = 'O';
    }else{
        currentPlayer = 'X';
    }
}
```

2.18.2 If game_mode equal to b

Se o utilizador inseriu, por exemplo, no terminal o comando **./TicTacToe_Game b**, este if vai ser executado porque a condição vai ser verdadeira.

Inicialmente, vai ser emitido no terminal uma mensagem para o utilizador indicar se quer jogar primeiro ou não.

Listing 2.16: Initial terminal message

```
char choice;
printf("Do you want to play first? (y/n):\n");
scanf("%c", &choice);
```

Em seguida, através de um *swich case*, o programa vai executar um conjunto de instruções dependendo da decisão do utilizador.

Caso o utilizador queira jogar em primeiro lugar, dentro do ciclo while infinito, vai ser "printado" o estado atual do tabuleiro, vai ser pedido para que o utilizador escolha uma posição, em seguida é feita a validação da mesma e caso a mesma seja válida vai ser introduzida no tabuleiro a jogada do utilizador, caso contrário é emitada uma mensagem de erro no terminal e o utilizador tem de voltar a introduzir uma posição. Após ser introduzida no tabuleiro a jogada do utilizador, tal como em qualquer modo de jogo, vai ser verificado se existe um vencedor ou se há um empate. Caso exista um deles, o jogo termina (break), caso contrário é a vez do *random agent* jogar. Após a jogada do agente, é verificado, novamente, se há um vencedor ou se há empate e volta a ser executado de novo todo o processo até haver um empate ou um vencedor (utilizador volta a jogar, verifica se há vencedor/empate...).

Listing 2.17: While loop

```
while(1){
    printBoard();

    printf("Player %c, enter row and column (e.g, 0 0 or 0 1): ", currentPlayer);
    scanf("%d %d", &row, &col);

    // Validate the move
    if (!isValidMove(row, col)) {
        printf("Invalid move! Try again.\n");
        continue;
    }

    board[row][col] = currentPlayer;

    // Check if the game is over (winner or tie)
    if (checkWinner(currentPlayer)) {
```

```
            printBoard();
            printf("Player %c wins!\n", currentPlayer);
            break;
        }else if(checkTie()){
            printBoard();
            printf("Tie!\n");
            break;
        }

        printBoard();

        bot_win = bot_random_generator_move(choice);
        if(bot_win){
            printBoard();
            break;
        }else if(checkTie()){
            printBoard();
            printf("Tie!\n");
            break;
        }
}
```

Caso o utilizador deixe o *random agent* jogar primeiro, o processo é muito semelhante ao anterior à exceção de que, em primeiro lugar, é feita a jogada do *random agent* e só depois é que o utilizador pode jogar, isto é, o jogador 'X' (primeiro) passa a ser o agente e o jogador 'O' (segundo) passa a ser o utilizador. O excerto de código abaixo mostra o que foi dito em cima.

Listing 2.18: while loop

```
while(1){
    currentPlayer = 'O';
    bot_win = bot_random_generator_move(choice);
    if(bot_win){
        printBoard();
        break;
    }else if(checkTie()){
        printBoard();
        printf("Tie!\n");
        break;
    }
    printBoard();

    int validMove = 0;
    do {
        printf("Player %c, enter row and column (e.g., 0 0 or 0 1): ", currentPlayer);
        scanf("%d %d", &row, &col);

        // Validate the move
        validMove = isValidMove(row, col);

        if (!validMove) {
            printf("Invalid move! Try again.\n");
        }
    } while (!validMove);

    board[row][col] = currentPlayer;

    // Check if the game is over (winner or tie)
    if (checkWinner(currentPlayer)) {
        printBoard();
        printf("Player %c wins!\n", currentPlayer);
        break;
```

```
    }else if(checkTie()){
        printBoard();
        printf("Tie!\n");
        break;
    }

    printBoard();
}
```

2.18.3 If game_mode equal to a

Se o utilizador introduzir, por exemplo, o comando **./TicTacToe_Game a**, as instruções dentro deste *if* vão ser executadas pelo programa.

Inicialmente, através da função *nn_create*, vai ser criada a estrutura da rede neural inicializando e alocando memória para os atributos da mesma bem como atribuir o número de unidades em cada layer aos respetivos atributos. Em seguida, é carregado para estrutura da rede o estado da rede treinada (leitura dos pesos entre as unidades da rede treinada do ficheiro *Trained_Network.txt*).

Listing 2.19: Create neural network and load

```
NeuralNetwork *nn = nn_create();
load_nn(nn);
```

Posto isto, vai ser novamente pedido ao utilizador que escolha se quer jogar em primeiro ou se quer o agente jogue em primeiro.

Caso o utilizador queira jogar em primeiro lugar, inicialmente é "printado" o estado atual do tabuleiro, e depois é pedido que o utilizador insira uma posição no terminal (linha e coluna) através da função *scanf()*. Após o utilizador inserir a posição, tal como em qualquer modo de jogo, é feito a validação da mesma e se for válida é inserida no tabuleiro caso contrário é emitida no terminal uma mensagem de erro e o utilizador tem de voltar a inserir outra posição. Após a jogada do utilizador, é verificado se há um vencedor ou se existe um empate e na ocurrência de um dos dois o jogo termina, caso contrário é a vez do agente jogar.

A jogada do agente consiste em introduzir o estado atual do tabuleiro nos *inputs* da rede neural treinada através da função *inputStringToNeuralNetwork()* que transforma o estado do tabuleiro num formato específico (`----X---O` -> 0 0 0 0 1 0 0 0 -1) para puderem ser introduzidos como inputs na rede através da função *load_input_values()* e, por fim, propaga esses valores até à saída de maneira a descobrir a próxima jogada do agente inteligente através da função *agent_move()*.

Após a jogada do agente, volta a ser verificado se existe vencedor ou empate e caso não haja nenhum dos dois, volta a ser repetido o processo todo até haver um vencedor/empate.

Listing 2.20: While loop

```
while(1){
    printBoard();

    printf("Player␣%c,␣enter␣row␣and␣column␣(e.g.,␣0␣0␣or␣0␣1):␣", currentPlayer);
    scanf("%d␣%d", &row, &col);

    // Validate the move
    if (!isValidMove(row, col)) {
        printf("Invalid␣move!␣Try␣again.\n");
        continue;
    }

    board[row][col] = currentPlayer;

    // Check if the game is over (winner or tie)
    if (checkWinner(currentPlayer)) {
        printBoard();
        printf("Player␣%c␣wins!\n", currentPlayer);
```

```
            break;
        }else if(checkTie()){
            printBoard();
            printf("Tie!\n");
            break;
        }

        printBoard();

        currentBoardState = getCurrentBoardState();
        inputStringToNeuralNetwork(currentBoardState, current_inputs);
        load_input_values(current_inputs, nn);
        propagate(nn);

        agent_win = agent_move(nn, choice);
        if(agent_win){
            printBoard();
            break;
        }else if(checkTie()){
            printBoard();
            printf("Tie!\n");
            break;
        }
}
```

Por outro lado, caso o utilizador deixa o agente jogar primeiro, o processo é muito semelhante ao de cima à exceção de que é o agente a efetuar a primeira jogada (Player X) e só depois é que o utilizador pode efetuar a sua jogada (Player O).

Listing 2.21: While loop

```
currentPlayer = 'O';
while(1){
    currentBoardState = getCurrentBoardState();
    inputStringToNeuralNetwork(currentBoardState, current_inputs);
    load_input_values(current_inputs, nn);
    propagate(nn);

    agent_win = agent_move(nn, choice);
    if(agent_win){
        printBoard();
        break;
    }else if(checkTie()){
        printBoard();
        printf("Tie!\n");
        break;
    }

    printBoard();

    int validMove = 0;
    do {
        printf("Player␣%c,␣enter␣row␣and␣column␣(e.g.,␣0␣0␣or␣0␣1):␣", currentPlayer);
        scanf("%d␣%d", &row, &col);

        // Validate the move
        validMove = isValidMove(row, col);

        if (!validMove) {
            printf("Invalid␣move!␣Try␣again.\n");
        }
    } while (!validMove);
```

```
        board[row][col] = currentPlayer;

        // Check if the game is over (winner or tie)
        if (checkWinner(currentPlayer)) {
            printBoard();
            printf("Player %c wins!\n", currentPlayer);
            break;
        }else if(checkTie()){
            printBoard();
            printf("Tie!\n");
            break;
        }

        printBoard();
}
```

2.18.4 If game_mode equal to t

Finalmente, na última condição *if*, caso o utilizador introduzir, por exemplo, o comando **./TicTacToe_Game t**, o programa vai executar este *if*.

Inicialmente, vai ser aberto o ficheiro de jogos no modo de leitura e vai ser alocada memória para o array *boardStates* (definimos por default o valor 1000 como tamanho máximo de board states dentro do ficheiro) que vai armazenar os estados que foram lidos do ficheiro *games.txt* com o exemplo dos jogos, através da função *fscanf()*.

Listing 2.22: Open file in read mode and store info

```
FILE *fp;
fp = fopen("games.txt", "r");
if (fp == NULL) {
    fprintf(stderr, "Error opening the file.\n");
    return 1;
}

//1000 board states, default size, which means, 1000 max board states in the file
char *boardStates[1000];

// Allocate memory for each board state
for (int i = 0; i < 1000; i++) {
    boardStates[i] = malloc(10); // 9 + '\0' = 10
    if (boardStates[i] == NULL) {
        fprintf(stderr, "Memory allocation error.\n");
        // Handle error and free allocated memory
        for (int j = 0; j < i; j++) {
            free(boardStates[j]);
        }
        fclose(fp);
        return 1;
    }
}
int numberOfBoardStates = 0;
while(numberOfBoardStates < 1000 && fscanf(fp, "%9s", boardStates[numberOfBoardStates])
!= EOF){
    numberOfBoardStates++;
}
fclose(fp);
```

Em seguida, vai ser inicializada a estrutura da rede neural e vão ser atribuídos valores aleatórios às arestas da rede inicializada. Por default, definimos 150 000 iterações para dar o máximo de tempo de treino à rede (apesar de um certo ponto o progresso de treino ser quase nulo ou não existir), definimos um *learning rate* de

0.7 e um *threshold* de 0.01.

Posto isto, o processo de treino vai começar. A cada iteração do ciclo while, vai existir um ciclo *for* que vai percorrer todos os estados do tabuleiro e vai ser realizado o processo de treino (propagate e back propagate). Em seguida, verifica se o próximo estado *(i+1)* é o estado do próximo jogo a ser treinado, ou seja, quando o estado *(i)* é o último estado do jogo atual que está a ser treinado. Esta condição é usada para nos certificarmos que a rede está a aprender cada jogo independentemente sem misturar jogos.

Cada estado *(i)* vai ser carregado como inputs na rede neural, através da função *inputStringToNeuralNetwork()* que mete no formato correto, e o *target* na saída para esse estado *(i)* vai ser o estado *(i+1)* que, através da função *giveDiseredOutputForNeuralNetwork*, vai meter as saídas no formato correto para posteriormente puder ser calculado o erro para cada saída.

Após ser calculado o erro, é verificado se para a entrada atual o *output* é válido, ou seja, próximo do *target* e é realizado o back propagation para atualizar os pesos das arestas.

Após terminar de se percorrer todos os estados do tabuleiro armazenados no atributo *boardStates* na iteração do ciclo *while*, provenientes da leitura do ficheiro com os exemplos dos jogos, é verificado se todos os outputs para todas as sequências de entrada estão dentro do desejável (*numberOfBoardStates* * 9 porque cada sequência de entrada, estado, vai ter 9 *outputs*), isto é, as saídas estão próximas do *target*. Se todos estiverem dentro do desejável o treino termina caso contrário, o treino continua até o número de iterações atingir o limite *maxIterations*.

O código abaixo mostra todo o processo de treino que foi explicado nos parágrafos de cima.

Listing 2.23: Training process

```
while(iteration < maxIterations){
    all_outputs_within_threshold = 0;
    //Iterate over the board states
    for(int i = 0; i < numberOfBoardStates; i++){
        if(checkIfLastBoardStateForCurrentGame(boardStates[i+1]) == 0){
            inputStringToNeuralNetwork(boardStates[i], inputs);
            giveDiseredOutputForNeuralNetwork(boardStates[i],boardStates[i+1]
            ,desired_outputs);
            load_input_values(inputs, nn);
            propagate(nn);

            //Calculate error of the outputs
            for (int j = 0; j < nn->number_O; j++) { // number_O is 9
                nn->errors_output[j] = desired_outputs[j] - nn->O[j];
            }

            for(int j = 0; j < nn->number_O; j++){
                if(fabs(nn->errors_output[j]) <= threshold ) {
                    all_outputs_within_threshold++;
                }
            }
            back_propagate(nn, learningRate);
        }
    }
    if(all_outputs_within_threshold == numberOfBoardStates * 9){
        printf("All outputs within threshold! Finished at iteration %d !\n", iteration);
        break;
    }
    iteration++;
}
```

Chapter 3

Results

Nesta secção vamos mostrar os resultados para as 3 perguntas do enunciado do projeto. Estes resultados foram obtidos com 9 unidades de entrada, 50 unidades no meio e 9 unidades de saída. O número de unidades do meio pode ser alterado no ficheiro *TicTac_NeuralNetwork.txt* (Se for alterado tem de se treinar a rede outra vez para se voltar a jogar com a rede treinada para esse número de unidades).

3.1 Exercise 1

Após correr o comando **./TicTacToe_Game m** (sem um estado predefinido do tabuleiro) vai ser iniciado o jogo e vai aparecer no terminal o estado inicial do tabuleiro bem como um pedido no prompt do terminal para o utilizador escolher uma posição no tabuleiro.

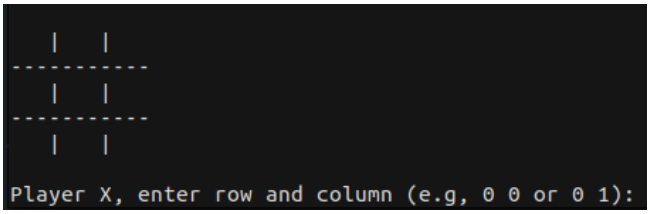
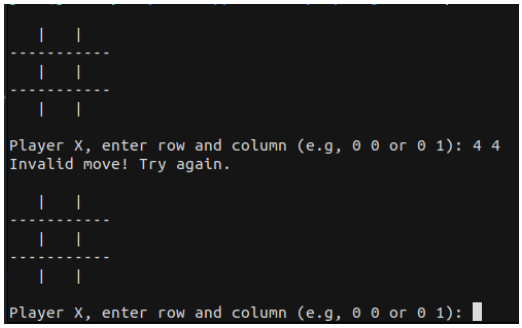
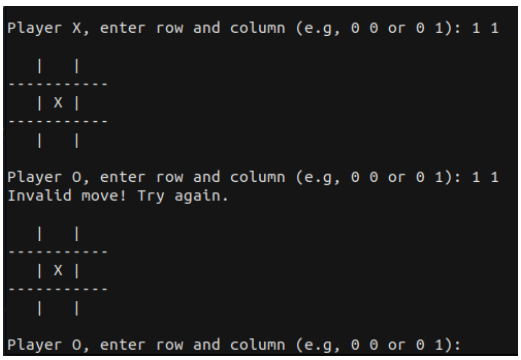


Figure 3.1: Terminal após correr o camando de cima

Para qualquer modo de jogo, caso o utilizador escolha uma posição fora do tabuleiro ou uma posição que não está livre irá obter uma mensagem de erro, tal como mostra as imagens em baixo.



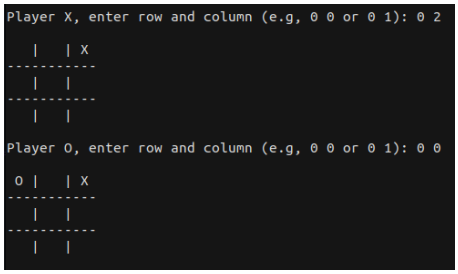
(a) Posição fora do tabuleiro



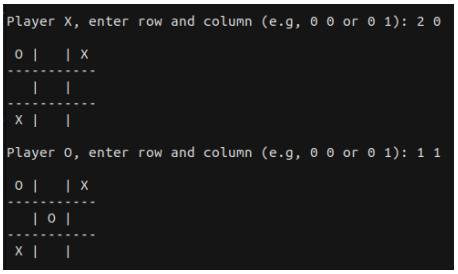
(b) Posição ocupada

Figure 3.2: Posições inválidas

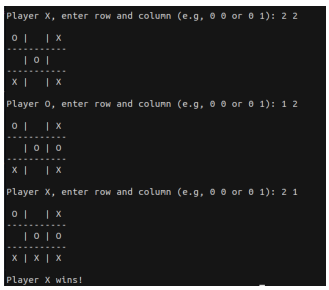
Em seguida, está o exemplo de um jogo, entre os dois colegas do grupo, que resultou na vitória do jogador **X**.



(a) Jogada 1 e 2



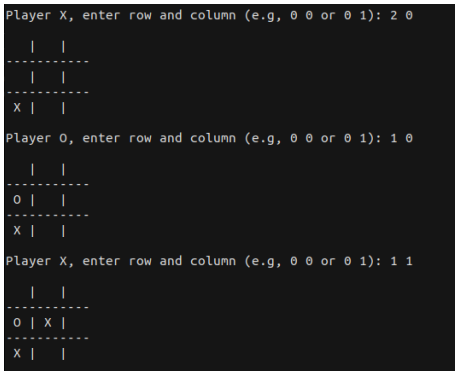
(b) Jogada 3 e 4



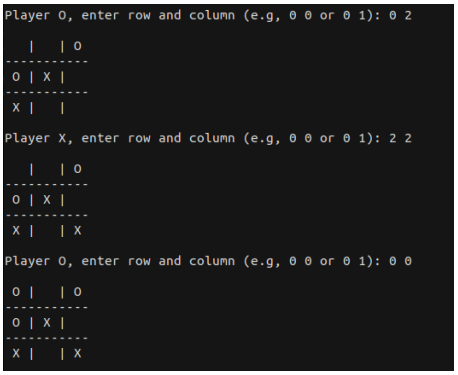
(c) Últimas 3 jogadas

Figure 3.3: Exemplo de uma vitória

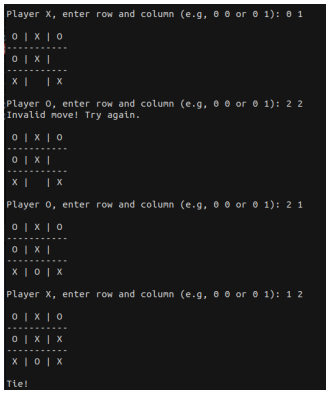
Posto isto, jogámos novamente para mostrar um exemplo onde o resultado do jogo deu empate.



(a) Jogada 1,2 e 3



(b) Jogada 4,5 e 6



(c) Últimas jogadas

Figure 3.4: Exemplo de um empate

Por fim, caso o utilizador queira jogar inicializando o jogo com um estado predefinido do tabuleiro, também é possível correndo, por exemplo, o comando da figura abaixo.

```
./TicTacToe_Game m ----X---O
```

Figure 3.5: Comando para inicializar o jogo com um estado predefinido do tabuleiro

Onde **"m"** corresponde ao modo de jogo deste exercício e `"----X---O"` ao estado predefinido do tabuleiro.

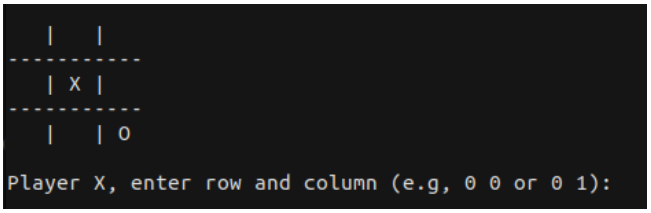


Figure 3.6: Estado predefinido do tabuleiro

Caso o utilizador selecione um estado predefinido do tabuleiro inválido irá receber uma mensagem de erro. O estado é considerado inválido se introduzir um caracter que seja diferente de 'X', 'O' ou '-', que corresponde ao espaço livre no tabuleiro, e caso introduza um tamanho inválido do tabuleiro (!=9).

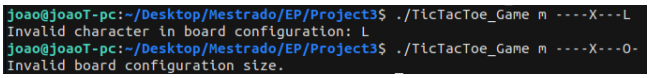
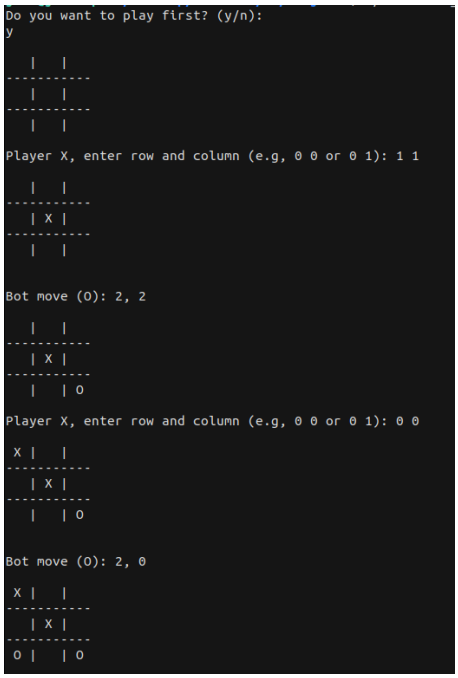


Figure 3.7: Estado predefinido do tabuleiro inválido

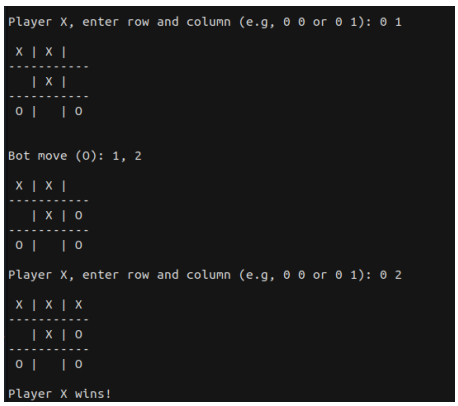
3.2 Exercise 2

Neste exercício o utilizador vai jogar contra um *random agent* e, se quiser, também poderá inicializar o jogo com um estado predefinido (por exemplo, `./TicTacToe_Game b 10 XX-------`), tal como foi mostrado na secção anterior, e poderá também alterar a seed que irá gerar a posição aleatória no tabuleiro.

Nas figuras abaixo, encontra-se o exemplo de um jogo onde o utilizador joga primeiro (tem de selecionar y ou n, caso não selecione nenhum dos dois vai aparecer uma mensagem de erro) cujo valor de seed passada como argumento no terminal é 10 (**./TicTacToe_Game b 10**).



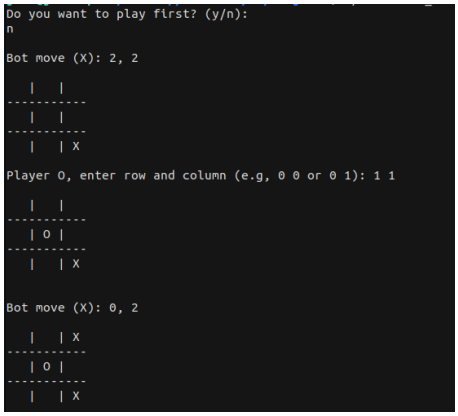
(a) Parte 1



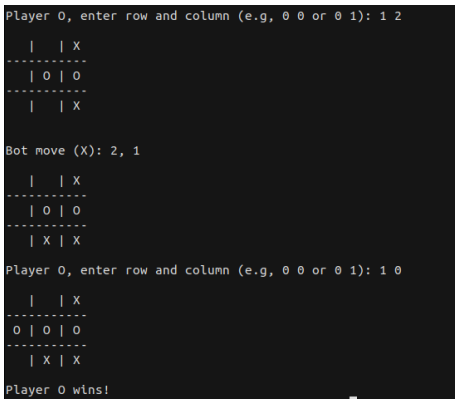
(b) Parte 2

Figure 3.8: Jogar contra um random agent com valor de seed igual a 10

Em seguida, voltámos a correr o mesmo comando com uma seed igual a 12 e desta vez era o random agent que começava a jogar em primeiro e obtemos os seguintes resultados.



(a) Parte 1



(b) Parte 2

Figure 3.9: Jogar contra um random agent com valor de seed igual a 12

3.3 Exercise 3

3.3.1 Training Process

Para treinar a rede neural, corremos o comando **./TicTacToe_Game t** e obtivemos um ficheiro com os pesos das arestas, entre as unidades, da rede treinada (Demorou cerca de 1 hora a treinar para os jogos atuais do ficheiro games.txt e para o número de iterações). A seguinte imagem mostra um exemplo do resultado do ficheiro *Trained_Network.txt* após o treino.

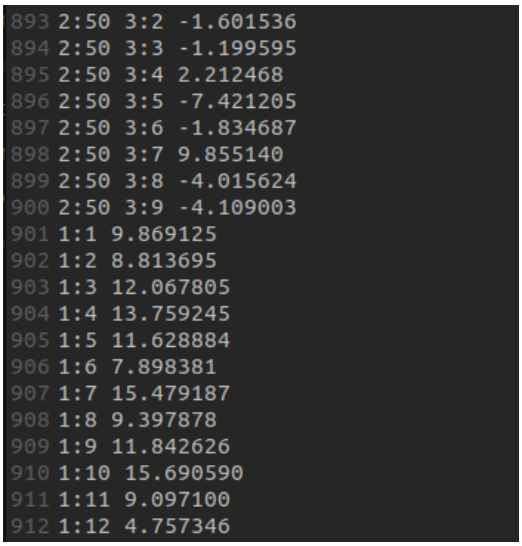


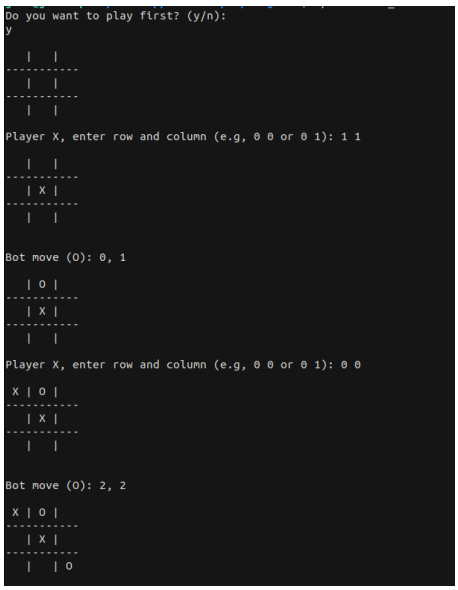
Figure 3.10: Ficheiro após treino

Como é possível ver pela imagem, o formato do ficheiro é muito semelhante ao do primeiro projeto à exceção que tivemos de adicionar também os pesos da ligações das bias da camada inicial e do meio às respetivas unidades seguindo o formato seguinte -> bias(1|2):units weight.

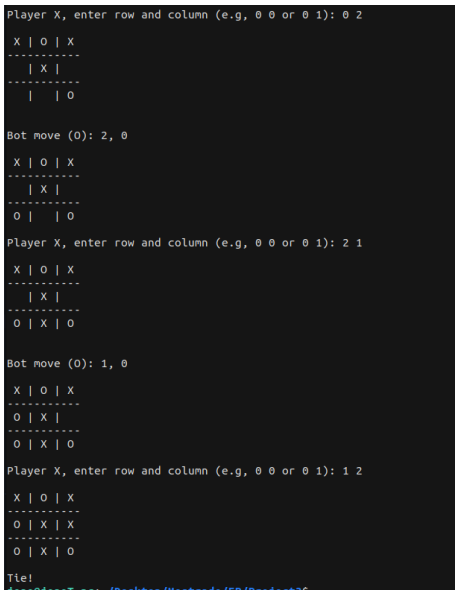
Bias é a unidade bias da camada inicial (1) ou a unidade bias da camada do meio (2), **unit** é a unidade que faz a ligação com a respetiva bias (por exemplo, se bias for 1 está na camada principal e faz ligação com uma unit que está na camada do meio à exceção da bias dessa camada) e, por fim, **weight** é o peso da ligação da bias com essa unit.

3.3.2 Game against the agent

Para jogar contra o agente inteligente corremos o comando **./TicTacToe_Game a** no terminal, obtendo os seguintes resultados.



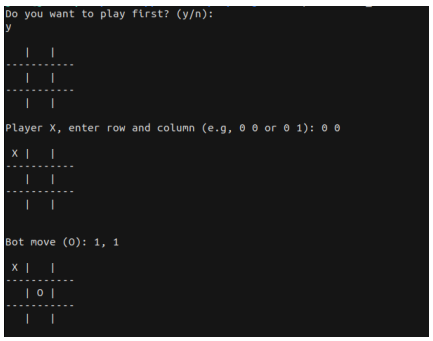
(a) Parte 1



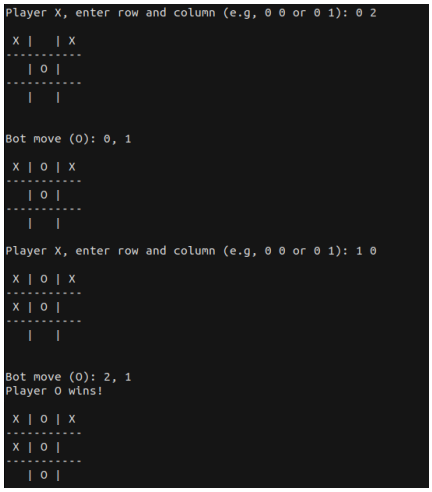
(b) Parte 2

Figure 3.11: Jogar contra um agente inteligente

E agora um exemplo onde o agente ganhou.



(a) Parte 1



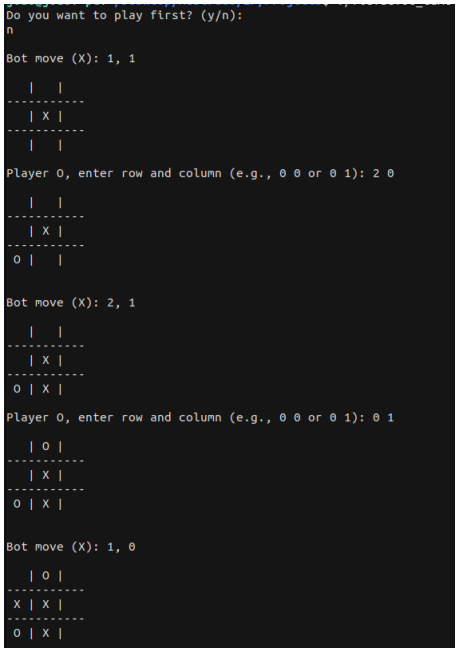
(b) Parte 2

Figure 3.12: Jogar contra um agente inteligente (derrota)

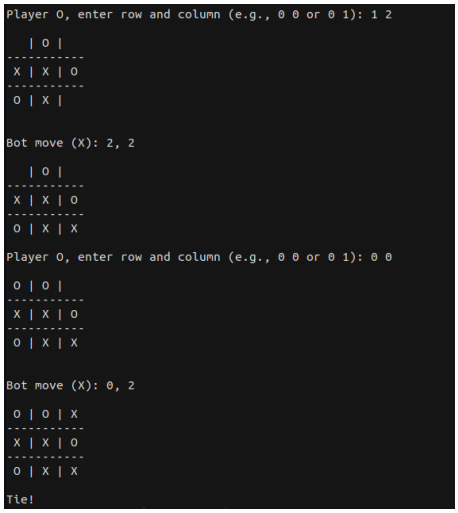
Tal como é possível ver pelas figuras de cima, o agente jogou corretamente tanto no jogo que terminou em empate como no jogo onde o agente ganhou.

De salientar que, como é normal, este agente não é perfeito, tem falhas podendo haver ocasiões onde ele jogue mal, isto é, não faça a melhor jogada. Isto acontece porque o ficheiro no qual a rede se baseia para treinar/aprender não tem todas as combinações possíveis de jogadas sendo normal ele fazer uma jogada menos boa pois não aprendeu todas as combinações possíveis e também poderá haver ocasiões onde para um dado estado de jogo o correspondente output está longe do desejável/target (não teve tempo de treino suficiente ou simplesmente não conseguiu aprender) levando ao agente a tomar uma decisão menos certa. De qualquer maneira, este agente tem muitos melhores resultados e toma bastantes melhores decisões em relação ao *random agent*, tal como é esperado.

Posto isto, corremos novamente o programa mas desta vez deixámos o agente jogar em primeiro lugar, obtendo os seguintes resultados.



(a) Parte 1



(b) Parte 2

Figure 3.13: Jogar contra um agente inteligente onde ele joga primeiro

Depois de vários jogos, foi possível verificar que o agente inteligente tem melhor comportamento quando ele joga em primeiro lugar em relação a quando ele joga em segundo lugar.

Por fim, se o utilizador quiser, também poderá inicializar o jogo com um estado predefinido do tabuleiro. De maneira a testar este cenário corremos o seguinte comando:

```
./TicTacToe_Game a XXO-O----
```

E deixámos o agente jogar em primeiro lugar, obtendo os seguintes resultados.

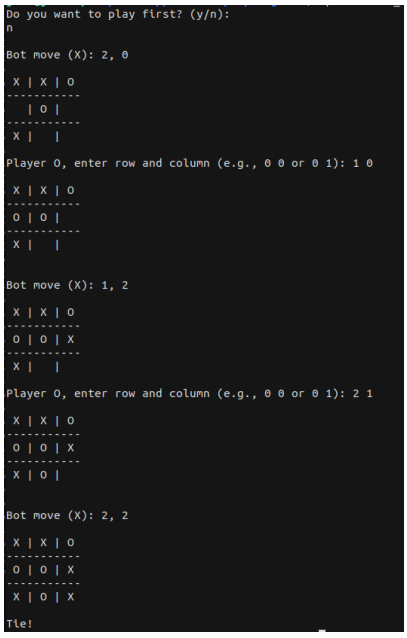


Figure 3.14: Resultado do jogo com um estado predefinido

Chapter 4

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%