

Elementos de Programação

Trabalho nº 1

Jodionísio Muachifi «97147»

Miguel Simões «118200»

Gustavo Reggio «118485»

Grupo nº 1

Mestrado em Engenharia de Computadores e Telemática

Mestrado em Engenharia de Automação Industrial

Professor: Armando José Formoso de Pinto

30 de outubro de 2023

[Link do repositório do Projeto 1](#)

Conteúdo

Abreviaturas	ii
1 Introdução	1
Parte I	2
2.1 Exercício nº 1	2
2.2 Exercício nº 2	3
2.3 Exercício nº 3	4
2.4 Exercício nº 4	5
Parte II	6
3.1 Exercício 5 (a)	6
3.2 Exercício 5(b)	7
3.3 Exercício 5(c)	7
3.4 Exercício 5(d)	8
3.5 Exercício 5(e)	8
4 Testes unitários e programação defensiva	9
4.1 Testes unitários	9
4.2 Programação defensiva	9
5 Contribuição dos autores	10
6 Conclusão	11

Figuras

Figura 1 - Validação das entradas e seus resultados (ex1)	2
Figura 2 - Validação das entradas e seus resultados (ex2)	3
Figura 3 - Validação das entradas e seus resultados (ex3)	4
Figura 4 - Estrutura de dados da rede neural(ex5)	6
Figura 5 - Estrutura de dados do peso(ex5)	7
Figura 6 - Resultados da rede neural/peso escrito no ficheiro	8

Abreviaturas

MECT – Mestrado em Engenharia de Computadores e Telemática

MEAI – Mestrado em Engenharia de Automação Industrial

PD – Programação defensiva

TU – Teste Unitário

Capítulo 1

1 Introdução

O presente relatório tem como objetivo demonstrar os resultados obtidos por meio dos programas desenvolvidos para abordar diversos aspetos, tais como a leitura e escrita em ficheiros, a introdução à compreensão de redes neurais e a sua expansão, bem como a conversão de números inteiros de base decimal para binária, e a conversão de dígitos binários para o respetivo formato decimal.

Para atingir esses objetivos, utilizamos a linguagem de programação C e suas bibliotecas padrão já incorporadas no ambiente de desenvolvimento. Entre as bibliotecas utilizadas, destacam-se a `<stdlib.h>`, que oferece funções para alocação de memória, controlo de processos, conversões e outras operações essenciais, e a `<stdbool.h>`, que fornece macros para a manipulação de dados booleanos (introduzida no padrão C99), entre outras.

Para que a robustez sortisse efeito aos programas desenvolvidos, inserimos todas as validações necessárias. Além disso, utilizamos testes unitários para averiguarmos as funcionalidades em cada parte ou trecho de código.

Todo o código desenvolvido, juntamente com as instruções para executar os programas, encontra-se disponível no nosso repositório no GitHub ou resumido em cada exercício, proporcionando um acesso fácil e conveniente para utilização.

Capítulo 2

Parte I

Os exercícios desenvolvidos nesta parte tinham como principal propósito aplicar e integrar a matéria que estudamos nas aulas teóricas.

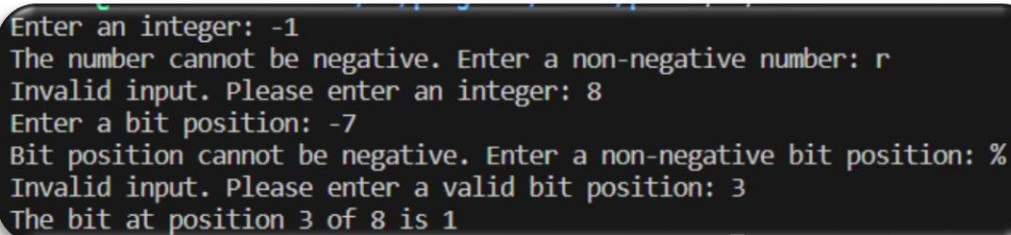
Para fins de segurança, todo o código desenvolvido nesta parte imprime mensagens de erro no terminal caso o utilizador insira “inputs” que não correspondam àqueles exigidos.

2.1 Exercício nº 1

Este exercício é útil para entender a manipulação de números inteiros em bits e também para praticar a entrada de dados e validação em programas em linguagem C.

Basicamente, este exercício consiste na implementação de um programa chamado “*print_bit*”. Este programa tem como objetivo ler um número inteiro (*i*) e uma posição de bit (*p*) a partir do teclado e imprimir o valor correspondente desse bit na representação binária de (*i*).

Criamos neste programa uma função designada “*isInteger()*”, para verificar se a entrada é um número inteiro válido. Se a entrada for negativa ou não for um número inteiro válido, o utilizador é solicitado a fornecer uma entrada válida. Na Figura 1, podemos constatar essas entradas.



```
Enter an integer: -1
The number cannot be negative. Enter a non-negative number: r
Invalid input. Please enter an integer: 8
Enter a bit position: -7
Bit position cannot be negative. Enter a non-negative bit position: %
Invalid input. Please enter a valid bit position: 3
The bit at position 3 of 8 is 1
```

Figura 1 - Validação das entradas e seus resultados (ex1)

Para correr este programa, é necessário seguir as instruções que se segue e, assegurar que está no diretório partI:

```
user@host: partI$ make
```

```
user@host: partI$ ./ex1
```

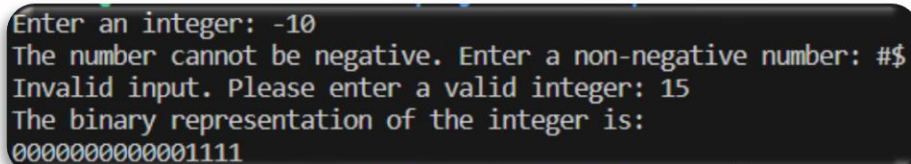
Na Figura 1 mostra os resultados após execução deste exercício.

2.2 Exercício nº 2

Este exercício é útil para entender como os números inteiros podem ser representados em formato binário e para praticar a manipulação de bits em linguagem C.

Basicamente, este exercício consiste na implementação de um programa chamado “*print_bits*”. O objetivo deste programa é ler um número inteiro (*i*) a partir do teclado e imprimir sua representação binária, da parte mais significativa para a menos significativa.

Tal como no exercício anterior, reutilizamos a função “*isInteger()*” para validar as entradas. Na Figura 2, apresenta-se a validação das entradas e os respetivos resultados. O programa “*print_bits*” foi renomeado como “*print_bits(int num)*” (neste caso, uma função) que recebe um número inteiro (*num*) como argumento e imprime a sua representação binária no formato de 16 bits. Esta função calcula o valor do bit atual utilizando operações de deslocamento à direita e operadores de bits.



```
Enter an integer: -10
The number cannot be negative. Enter a non-negative number: #
Invalid input. Please enter a valid integer: 15
The binary representation of the integer is:
00000000000001111
```

Figura 2 - Validação das entradas e seus resultados (ex2)

Para executar este programa, é necessário seguir as instruções que se seguem e certificar-se de que se encontra no diretório partI:

```
user@host: partI$ make
```

```
user@host: partI$ ./ex2
```

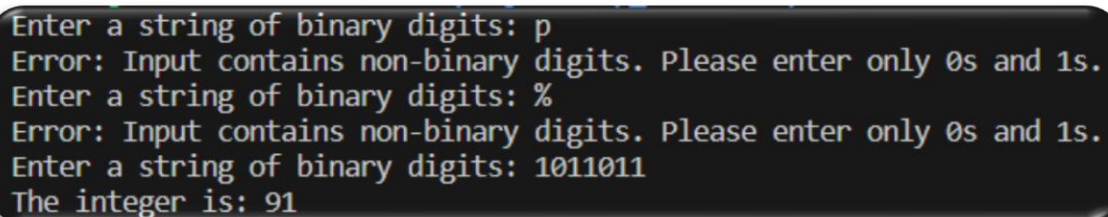
Na Figura 2 mostra os resultados após execução deste exercício.

2.3 Exercício nº 3

Este exercício é equiparado ao exercício anterior quanto a sua utilidade em entendimento de números em formato binário.

O programa “*bits_to_int*”, tem como objetivo ler um número em formato binário (i) a partir do teclado introduzido pelo utilizador e imprimir a sua representação em notação decimal.

O programa acima mencionado, renomeamos como “*bits_to_int(char* bits)*”, sendo uma função garante que seja introduzido somente os números: 0 e 1. A função percorre cada bit na *string*, do bit mais significativo para o menos significativo. Para cada bit igual a '1', ela adiciona o valor correspondente (2^i) ao número inteiro *num*. No final, a função retorna o número inteiro resultante.



```
Enter a string of binary digits: p
Error: Input contains non-binary digits. Please enter only 0s and 1s.
Enter a string of binary digits: %
Error: Input contains non-binary digits. Please enter only 0s and 1s.
Enter a string of binary digits: 1011011
The integer is: 91
```

Figura 3 - Validação das entradas e seus resultados (ex3)

Para executar este programa, é necessário seguir as instruções que se seguem e certificar-se de que se encontra no diretório *partI*:

```
user@host: partI$ make
```

```
user@host: partI$ ./ex3
```

Na Figura 3 mostra os resultados após execução deste exercício.

2.4 Exercício nº 4

Neste exercício, foi solicitado que uníssemos as funções previamente criadas nos exercícios anteriores, nomeadamente “**print_bits**” e “**bits_to_int**” num único programa denominado “**bits**”. Deste modo, aproveitamos todas as funcionalidades desenvolvidas nessas funções para atender aos objetivos deste exercício. No presente caso, os parâmetros necessários para as funções serão fornecidos ao programa como argumentos na linha de comando.

Se o utilizador quiser correr a função “**print_bits**” é necessário seguir as instruções que se seguem e certificar-se de que se encontra no diretório partI:

```
user@host: partI$ make  
user@host: partI$ ./ex4 pb 75
```

Caso o utilizador queira correr a função “**bits_to_int**” é necessário seguir as instruções que se seguem e certificar-se de que se encontra no diretório partI:

```
user@host: partI$ make  
user@host: partI$ ./ex4 bti 11010111
```


Capítulo 3

Parte II

O objetivo desta parte consistiu em introduzir o trabalho com redes neurais de forma introdutória, utilizando funções lineares, bem como abordar e desenvolver o código que permita à leitura e escrita dos pesos das ligações num determinado ficheiro de texto. Parte do pressuposto que as redes neurais, também conhecidas como redes neuronais, representam um tipo de modelo de aprendizagem de máquina inspirado no funcionamento do cérebro humano.

Estas redes são compostas por camadas de neurónios artificiais interligados, sendo utilizadas para resolver problemas complexos de processamento de dados e reconhecimento de padrões. As redes neurais podem ser aplicadas em diversas áreas, abrangendo desde o reconhecimento de voz e imagem até a previsão de séries temporais e tradução automática.

Neste exercício nós temos 3 ficheiros:

nn_base.h - composto por protótipos de estruturas e funções de redes neurais.

nn_base.c - composto por funções de desenvolvimento.

base.c - composto por testes de funções desenvolvidas.

3.1 Exercício 5 (a)

o objetivo deste exercício é criar uma estrutura de dados que possa representar uma rede neural com entradas (I), uma camada de unidades ocultas (H) que implementam funções lineares e saídas (O). Neste exercício começou-se por construir as estruturas pedidas com as informações da rede neural tal como mostra a Figura 4.

```
typedef struct NeuralNetwork
{
    int I; // Number of input units
    int H; // Number of hidden units
    int O; // Number of output units
} NeuralNetwork_t;
```

Figura 4 - Estrutura de dados da rede neural(ex5)

3.2 Exercício 5(b)

Neste exercício criamos uma estrutura de dados que foi usada para armazenar os valores das conexões (ou pesos) entre as unidades de uma rede neural. Esta estrutura é fundamental para o funcionamento da rede neural, pois os pesos representam a importância de cada conexão entre as unidades tal como ilustra a Figura 5.

```
typedef struct Weight
{
    int from_layer; // Index of the source layer
    int from_unit;  // Index of the source unit
    int to_layer;   // Index of the destination layer
    int to_unit;    // Index of the destination unit
    double weight;  // Weight of the connection
} Weight_t;
```

Figura 5 - Estrutura de dados do peso(ex5)

3.3 Exercício 5(c)

Criamos neste exercício uma função chamada *“load_weights(NeuralNetwork_t *nn, Weight_t **weights, const char *filename)”* que consiste em carregar pesos a partir de um ficheiro, onde esses pesos são especificados em um formato que utiliza a notação *“X:Y”* para denotar as conexões entre unidades e valores numéricos para representar o peso dessas conexões.

Conforme nota-se no protótipo da função temos ponteiros simples e duplos. Ponteiros são variáveis especiais em programação que armazenam endereços de memória em vez de valores diretamente. Eles são usados para referenciar e acessar dados armazenados em locais específicos na memória do computador. Os ponteiros são fundamentais para tarefas como alocação dinâmica de memória, passagem de parâmetros por referência e manipulação de estruturas de dados complexas.

Portanto, **decidimos** utilizar ponteiros no projeto pelas razões supracitadas e, além disso, os ponteiros permitirão neste caso à modificação direta dos pesos e economizar memória ao evitar a cópia de grandes conjuntos de pesos. Isso torna a função mais eficiente e flexível ao carregar e incorporar pesos em uma rede neural.

3.4 Exercício 5(d)

Neste exercício criamos a função ***“propagate_input(NeuralNetwork_t *nn, Weight_t *weights, double *input, double *output)”*** que implementa a etapa de propagação de entrada de uma rede neural, onde os valores de entrada são processados através das conexões de pesos para calcular os valores de saída da rede. A função utiliza uma ativação linear, mas é possível substituir a função ***“activate_unit(double input, double weight)”*** por outras funções de ativação mais complexas, dependendo dos requisitos da rede neural.

3.5 Exercício 5(e)

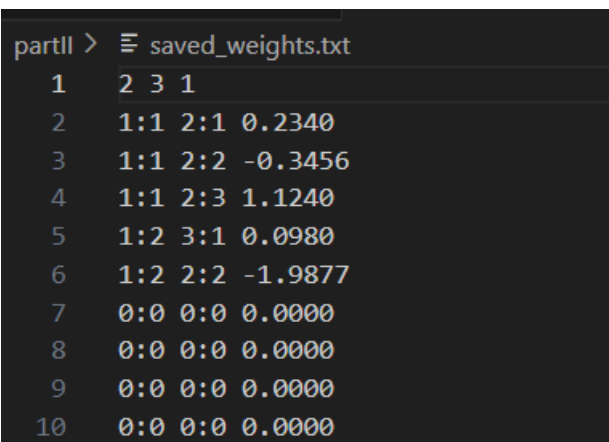
Este exercício tal como nos demais já detalhados, criamos uma função chamada ***“save_weights(NeuralNetwork_t *nn, Weight_t *weights, const char *filename)”***, que tem como objetivo gravar em um ficheiro os parâmetros de uma rede neural existente na memória, seguindo um formato específico, tal como mencionamos no exercício anterior.

Para correr este programa (exercício 5 por completo), é necessário seguir as instruções que se segue e, assegurar que está no diretório partII:

```
user@host: partII$ make
```

```
user@host: partII$ ./base
```

Após execução do programa proposto contido em ***base.c (program main)***, é criado um ficheiro designado de ***saved_weights.txt*** com resultados esperados, tal como ilustra a Figura x.



```
partII > ≡ saved_weights.txt
1 2 3 1
2 1:1 2:1 0.2340
3 1:1 2:2 -0.3456
4 1:1 2:3 1.1240
5 1:2 3:1 0.0980
6 1:2 2:2 -1.9877
7 0:0 0:0 0.0000
8 0:0 0:0 0.0000
9 0:0 0:0 0.0000
10 0:0 0:0 0.0000
```

Figura 6 - Resultados da rede neural/peso escrito no ficheiro

Capítulo 4

4 Testes unitários e programação defensiva

Testes unitários e programação defensiva são práticas cruciais na engenharia de software que desempenham papéis fundamentais na criação de software (programa) robusto e confiável.

4.1 Testes unitários

Os testes unitários (TU) são projetados para verificar o funcionamento de unidades individuais de código, como funções ou métodos. Eles ajudam a identificar erros e problemas de lógica no código logo no início do desenvolvimento, o que é crucial para corrigir problemas antes que se tornem mais complexos e difíceis de resolver.

No projeto, seja na parte 1 ou parte 2, criamos testes unitários para identificar erros e garantir que os programas criados em cada trecho do código funcionem corretamente.

4.2 Programação defensiva

A programação defensiva (PD) envolve escrever código de forma a prevenir erros e lidar com entradas inesperadas ou situações de erro. Isso inclui a validação de entradas do usuário e a verificação de pré-condições e pós-condições. Além disso, a PD, ajuda a tornar o software mais robusto, reduzindo a probabilidade de falhas e travamentos.

No projeto, não entramos de forma profunda sobre programação defensiva, mas utilizamos conceitos introdutórios sobre este assunto como verificação de pré-condições.

Para depurar os programas, utilizamos duas ferramentas importantes: o GNU Debugger (gdb) e o Valgrind, especialmente na Parte II, quando estávamos lidando com alocação de memória.

Para testar os testes unitários das duas partes do projeto, é necessário seguir as instruções que se segue e, assegurar que está no diretório `partI` ou `partII`:

```
user@host: partI$ make
```

```
user@host: partI$ ./unit_tests
```

Para limpar os executáveis, basta fazer ***make clean*** nos diretórios correspondentes.

Capítulo 5

5 Contribuição dos autores

A participação de cada autor foi excelente no que diz respeito à pesquisa, discussão e escrita deste trabalho, destacando-se o esforço de cada membro do grupo em relação ao desenvolvimento do código. A percentagem de contribuição para cada estudante fica como segue:

- Jodionísio Muachifi – 37%
- Miguel Simões – 33%
- Gustavo Reggio – 30%

Capítulo 6

6 Conclusão

Com este trabalho, pudemos compreender na prática como funcionam as operações com números binários e a construção de uma estrutura básica de rede neural em C, com foco na representação de unidades e conexões, carregamento de pesos, propagação de entradas e exportação de parâmetros. A implementação destas funcionalidades permitiu a manipulação de dados binários e a construção de redes neurais simples.

Referências bibliográficas

- https://www.w3schools.com/c/c_structs.php
- [https://en.wikipedia.org/wiki/Pointer_\(computer_programming\)](https://en.wikipedia.org/wiki/Pointer_(computer_programming))
- <https://www.youtube.com/watch?v=aircAruvnKk>
- <https://www.oreilly.com/library/view/learn-c-the/9780133124385/>
[exercício 15, 16, 17 e 18]
- https://en.wikipedia.org/wiki/Neural_network
- https://en.wikipedia.org/wiki/GNU_Debugger
- <https://en.wikipedia.org/wiki/Valgrind>