CONTROLO DE UM ROBÔ AUTÓNOMO ATRAVÉS DE REDES NEURONAIS

Adriano Bessa Pinto

Mestrado em Engenharia Eletrotécnica e de Computadores
Área de Especialização de Automação e Sistemas
Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto
2013

Este relatório satisfaz, parcialmente, os requisitos que constam da Ficha de Unidade Curricular de Tese/Dissertação, do 2º ano, do Mestrado em Engenharia Eletrotécnica e de Computadores

Candidato: Adriano Bessa Pinto, Nº 1070186, 1070186@isep.ipp.pt
Orientação científica: Ramiro de Sousa Barbosa, rsb@isep.ipp.pt
Coorientação científica: Manuel Fernando Santos Silva, mss@isep.ipp.pt

Mestrado em Engenharia Eletrotécnica e de Computadores
Área de Especialização de Automação e Sistemas
Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto
6 de novembro de 2013

Ao meu Pai

Agradecimentos

Gostaria de agradecer ao Engº. Ramiro Barbosa e ao Engº. Manuel Silva por toda a compreensão, disponibilidade e ajuda que demonstraram durante a elaboração deste projeto.

Adriano Pinto

Resumo

Neste trabalho pretende-se introduzir os conceitos associados às redes neuronais e a sua aplicação no controlo de sistemas, neste caso na área da robótica autónoma. Foi utilizado um AGV de modo a testar experimentalmente um controlo através de uma rede neuronal artificial.

A grande vantagem das redes neuronais artificiais é estas poderem ser ensinadas a funcionarem como se pretende. A partir desta caraterística foram efetuadas duas abordagens na implementação do AGV disponibilizado. A primeira abordagem ensinava a rede neuronal a funcionar como o controlo por lógica difusa que foi implementado no AGV aquando do seu desenvolvimento. A segunda abordagem foi ensinar a rede neuronal artificial a funcionar a partir de dados retirados de um controlo remoto simples implementado no AGV.

Ambas as abordagens foram inicialmente implementadas e simuladas no MATLAB, antes de se efetuar a sua implementação no AGV. O MATLAB é utilizado para efetuar o treino das redes neuronais multicamada proactivas através do algoritmo de treino por retropropagação de Levenberg-Marquardt.

A implementação de uma rede neuronal artificial na primeira abordagem foi implementada em três fases, MATLAB, posteriormente linguagem de programação C no computador e por fim, microcontrolador PIC no AGV, permitindo assim diferenciar o desenvolvimento destas técnicas em várias plataformas.

Durante o desenvolvimento da segunda abordagem foi desenvolvido uma aplicação Android que permite monitorizar e controlar o AGV remotamente.

Os resultados obtidos pela implementação da rede neuronal a partir do controlo difuso e do controlo remoto foram satisfatórios, pois o AGV percorria os percursos testados corretamente, em ambos os casos.

Por fim concluiu-se que é viável a aplicação das redes neuronais no controlo de um AGV. Mais ainda, é possível utilizar o sistema desenvolvido para implementar e testar novas RNA.

Palavras-Chave

Rede Neuronal Artificial, AGV, MATLAB, Android

Abstract

This paper aims to introduce the concepts associated with neural networks and its application in control systems, in this case in the field of autonomous robotics. An AGV was used in order to test experimentally the control by an artificial neural network (ANN).

The major advantage of neural networks is that they can be taught to work as intended. From this feature were taken two approaches in implementing the AGV control. In the first, the AGV was taught to perform like the control by fuzzy logic that was previously developed. The second approach taught the ANN to work from data taken from a simple remote control system.

Both approaches were initially implemented and simulated in MATLAB, prior to making its implementation in the AGV. The MATLAB was used to perform the training of multilayer feedforward neural networks by using the backpropagation algorithm of Levenberg-Marquardt.

The implementation of the ANN was implemented in three stages, MATLAB, then in the C programming language and, finally, in the PIC microcontroller on the AGV, thus illustrating the development of these techniques in multiple platforms.

During the development of the second approach was developed an Android application that allows to monitor and remotely control the AGV.

The results obtained by the implementation of the neural network from the previously implemented fuzzy control and by the remote control were satisfactory since the AGV performed the paths properly.

Finally it may be concluded that it is feasible the application of neural networks to control an AGV. Moreover, it is possible to use the developed system to implement and test new ANNs.

Keywords

Artificial Neural Network, AGV, MATLAB, Android

Índice

Índice de Figuras

Índice de Tabelas

Acrónimos

AGV – *Autonomous Guided Vehicle*

ANN – *Aritificial Neural Network*

API – *Application Programming Interface*

CAN – *Controller Area Network*

DEE – Departamento de Engenharia Eletrotécnica

ECG – Eletrocardiograma

ED – Erro de Distância

EEG – Eletroencefalograma

EO – Erro de Orientação

IDE – *Integrated Development Environment*

ISEP – Instituto Superior de Engenharia do Porto

MEEC – Mestrado em Engenharia Eletrotécnica e de Computadores

MSE – *Mean Squared Error*

PCB – *Printed Circuit Board*

PI – Proporcional Integral

PWM – *Pulse-Width Modulation*

RNA – Rede Neuronal Artificial

SDK – *Software Development Kit*

UUID – *Universally Unique Identifier*

1. INTRODUÇÃO

No contexto da Unidade Curricular de Tese/Dissertação, do 2º ano, do Mestrado em Engenharia Eletrotécnica e de Computadores (MEEC), área de especialização de Automação e Sistemas, do Departamento de Engenharia Eletrotécnica (DEE), do Instituto Superior de Engenharia do Porto (ISEP), desenvolveu-se a Tese com o tema "Controlo de um Robô Autónomo através de Redes Neuronais" onde se abordou a temática das Redes Neuronais Artificiais e da sua implementação na robótica.

O objetivo principal é o estudo dos conceitos associados às redes neuronais artificiais, de modo a controlar um robô com uma rede neuronal artificial que fosse responsável pela tomada de decisão do sistema. A utilização de um *Autonomous Guided Vehicle (*AGV) disponibilizado pelo orientador permitiu efetuar testes ao sistema de controlo através das redes neuronais artificiais de modo a retirarem-se algumas conclusões e validar os sistemas de controlo propostos.

Devido à necessidade de utilizar um sistema de controlo remoto para o AGV foi desenvolvida uma aplicação para um *smartphone* Android.

1.1. CONTEXTUALIZAÇÃO

Este projeto surgiu do desejo de realizar um trabalho no âmbito do controlo, especificamente o controlo de sistemas autónomos através de métodos utilizados na inteligência artificial. O AGV utilizado para testar as redes neuronais criadas durante a elaboração desta tese já existia anteriormente, pois tinha sido criado para testar o controlo de um AGV através de lógica difusa. O AGV e o controlador difuso foram utilizados como ferramentas para permitir a implementação e o treino de uma rede neuronal num AGV. Tanto o AGV como o controlador difuso foram criados na tese "Controlo Difuso de um AGV" efetuada em 2010 por Dário Osório nesta mesma instituição [1].

1.2. OBJETIVOS

O objetivo principal deste projeto é o controlo de um AGV através de redes neuronais artificiais. Dada a complexidade inerente a este objetivo, sentiu-se a necessidade de o subdividir em múltiplas tarefas de realização mais simples, tais como:

- Estudo teórico das redes neuronais artificiais;
- Criação de uma rotina para efetuar a amostragem do controlo difuso;
- Implementação de uma rede neuronal artificial no MATLAB;
- Implementação de uma rede neuronal artificial no programa de linguagem de programação C;
- Implementação de uma rede neuronal artificial no AGV;
- Estudo e implementação de uma interface de controlo e monotorização do AGV;
- Avaliação e análise dos resultados obtidos.

1.3. CALENDARIZAÇÃO

Para a execução deste trabalho foi necessário efetuar várias tarefas tais como: estudo dos conceitos associados às redes neuronais artificias, estudo do sistema utilizado para efetuar os testes experimentais das redes neuronais implementadas, entre outras que derivam do

desenvolvimento do projeto. A sua prossecução conduziu à calendarização apresentada na Tabela 1.

Tabela 1 Calendarização do projeto

ID	Tarefa	Duração	Q4 12		Q1 13			Q2 13			Q3 13		
			nov	dez	jan	fev	mar	abr	mai	jun	jul	ago	set
1	Estudo das redes neuronais artificiais	60d											
2	Redação do relatório do projeto	200d											
3	Estudo do AGV disponibilizado	25d											
4	Implementação das RNA no MATLAB	120d											
5	Implementação das RNA no AGV	100d											
6	Desenvolvimento da aplicação de monitorização e controlo em Android	58d											

1.4. ORGANIZAÇÃO DO RELATÓRIO

O relatório é constituído por seis capítulos sendo eles: Introdução, Redes Neuronais Artificiais, Plataforma do AGV e suas Funcionalidades, Projeto e Simulação da RNA, Testes e Resultados e Conclusões. No primeiro capítulo é feita uma introdução ao projeto e aos objetivos do mesmo. No Capítulo 2 são introduzidos os conceitos associados às redes neuronais artificiais. No terceiro capítulo é apresentada a arquitetura do sistema utilizado e da aplicação desenvolvida para um *smartphone* Android. No Capítulo 4 apresenta-se a fase de projeto e simulação das várias redes neuronais implementadas para o controlo de um AGV. No quinto capítulo é referido a implementação das redes neuronais, apresentados os resultados obtidos e a sua análise. No capítulo seguinte, Capítulo 6, são apresentadas as conclusões retiradas do projeto efetuado e propostas melhorias para trabalhos futuros.

2. REDES NEURONAIS ARTIFICIAIS

O trabalho em Redes Neuronais Artificiais (RNA) tem sido motivado e desenvolvido pelo reconhecimento de que o cérebro humano processa todas as informações captadas de uma forma muito própria. O cérebro pode ser comparável a um computador altamente complexo, não linear e paralelo. Ele tem a capacidade de estruturar e organizar as suas unidades de processamento, conhecidas por neurónios, de forma a realizar um processamento muito mais rápido do que qualquer outro computador digital hoje existente, para algumas funções particulares, como por exemplo, reconhecimento de padrões (imagens, texto escrito manualmente). Uma rede neuronal artificial pode considerar-se como inspirada nessa interpretação do funcionamento do cérebro humano. Sendo no entanto bastante diferente quer na dimensão, quer no modelo e funcionamento dos seus elementos.

Uma RNA pode ser implementada utilizando componentes eletrónicos ou simulada por programação num computador. Os modelos de redes neuronais realizam a manipulação de informações através da interação de um grande número de unidades básicas de processamento, às quais se dá o nome de neurónios artificiais, ou mais frequentemente nós, sendo fundamental também, a forma e tipo de interligações entre esses elementos. As RNA

podem apresentar uma ou mais camadas intermédias, ditas também de escondidas (*hidden layers*), de neurónios.

Do ponto de vista prático, as RNA têm como vantagem o facto de não necessitarem de conhecimento explícito do problema para tomar decisões, baseando-se unicamente nos exemplos que lhes são fornecidos. As RNA podem ser utilizadas na solução de uma grande quantidade de problemas encontrados nas mais diversas áreas de aplicação: classificação, diagnóstico, análise de sinais e de imagens, reconhecimento de padrões, otimização e controlo. As redes são particularmente eficientes na resolução de problemas em que não se dispõe de uma formulação analítica, nem conhecimento explícito acessível. No limite pode considerar-se uma RNA como uma "caixa de processamento" de entradas e saídas que pode ser treinada com base num conjunto limitado de exemplos, conforme apresentado na Figura 1.

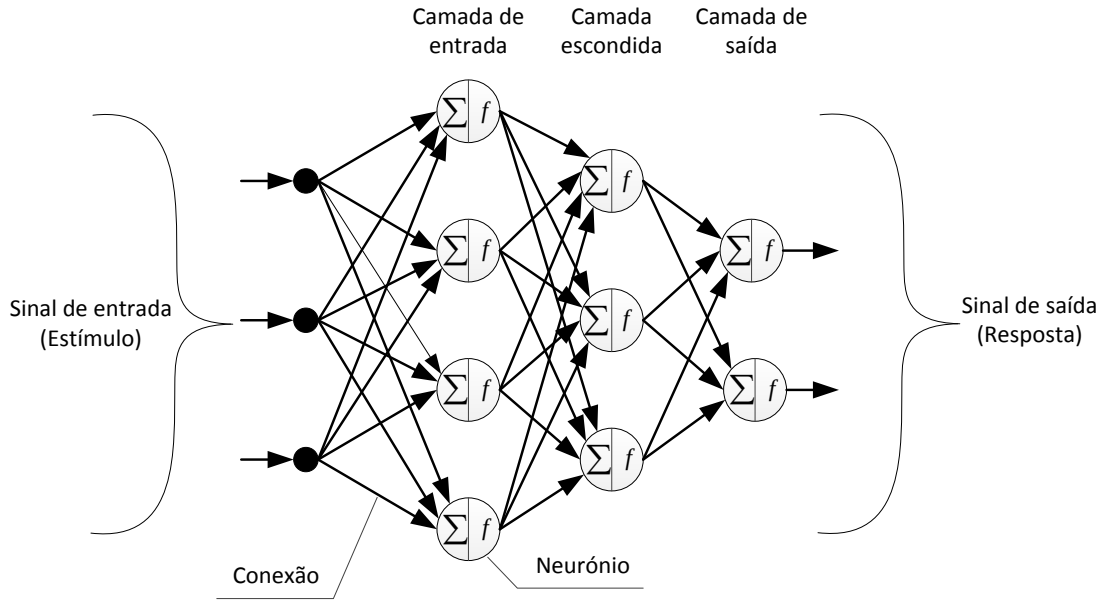


Figura 1 Rede neuronal artificial

2.1. COMPONENTES

Em geral pode-se considerar que uma rede neuronal é constituída por unidades de processamento ou nós (neurónios) e ligações entre esses nós, definindo a topologia ou arquitetura da rede. Outro aspeto determinante é o processo ou algoritmo, que define a forma de ajuste dos parâmetros da rede de forma a fornecer uma solução para um dado problema. Em geral pode dizer-se que existem várias alternativas em relação à configuração de uma rede neuronal com base nestes elementos, desde a escolha das funções que são

implementadas em cada nó, do efeito de cada uma das ligações entre os nós, à topologia e número de elementos utilizados. Ainda em relação aos algoritmos de treino, ou ajuste dos parâmetros da rede, também existem atualmente várias alternativas. Um neurónio artificial, ou nó, é a unidade fundamental de processamento de uma RNA, o qual recebe uma ou mais entradas, transformando-as em saídas. Cada entrada de um neurónio tem um peso associado, que determina sua intensidade. O esquema do neurónio artificial pode ser visualizado na Figura 2, onde u_i são as entradas na rede neuronal, $w_{1,i}$ são os pesos das conexões, b é a polarização, x é o valor de ativação e y é a saída real.

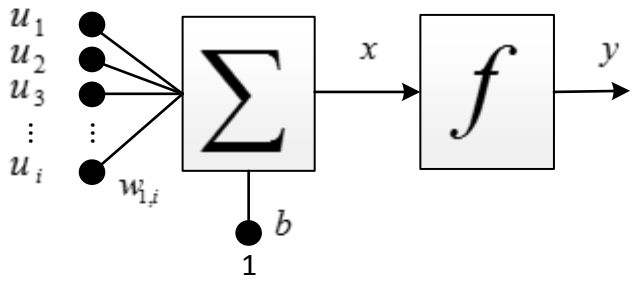


Figura 2 Neurónio artificial

Uma representação genérica de um neurónio humano é mostrada na Figura 3 de forma a estabelecer o paralelo entre os dois sistemas. Os dendritos são zonas recetivas, o corpo celular é onde se inicia a codificação da saída, os axónios constituem a linha de transmissão e as terminações do axónio transmitem o "aprendido" para outro neurónio.

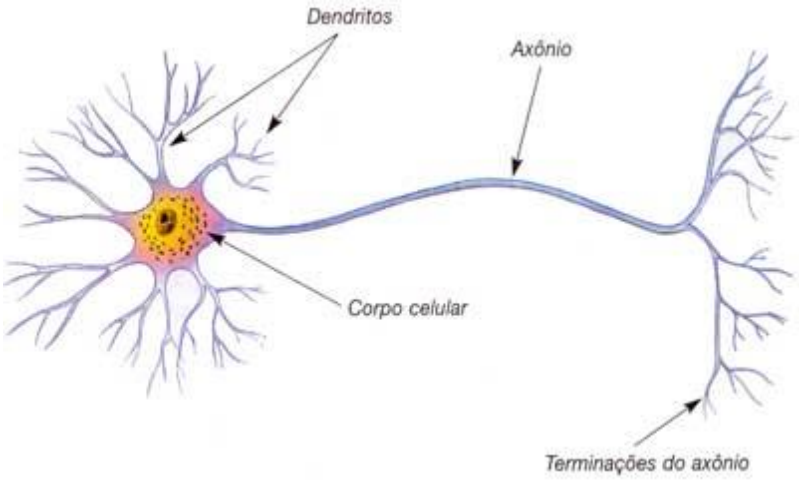


Figura 3 Neurónio humano

Cada neurónio artificial (nó) possui um estado interno chamado valor de ativação, que é modificado sempre que uma nova entrada é recebida. Essas entradas são combinadas e um novo valor de ativação é calculado que será o valor de entrada de uma função. O valor de

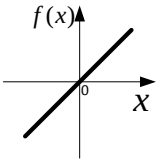
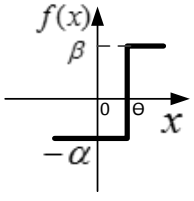
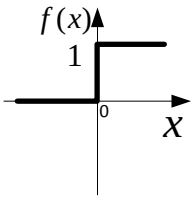
ativação também depende do valor da polarização do neurónio. Esta tem como função deslocar a função de ativação para a direita ou para a esquerda, o que permite que o treino de um neurónio convirja e tenha sucesso em menos iterações. A saída do neurónio é por sua vez calculada aplicando o valor de ativação a uma função de saída, por vezes também referida como função de ativação. Esta saída, por sua vez, serve de entrada para o neurónio seguinte e assim sucessivamente. A expressão que permite calcular o valor de saída de um neurónio artificial é apresentada na equação (1).

$$y = f\left(\sum_i w_{1,i} u_i + b\right) \tag{1}$$

2.2. FUNÇÕES DE ATIVAÇÃO

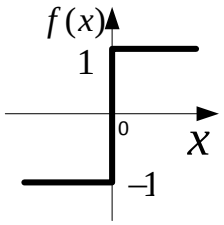
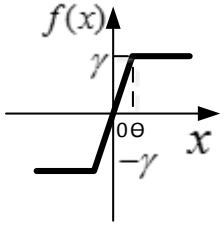
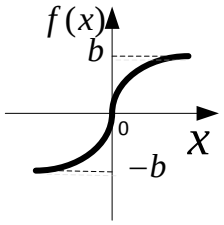
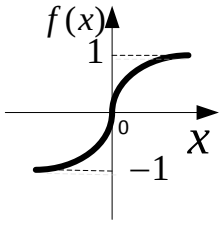
Um neurónio artificial é ativado segundo uma função de ativação que depende do valor obtido através da combinação linear das entradas com os pesos (estado interno do neurónio). A maioria das redes neuronais utiliza uma das funções de ativação básicas geralmente não lineares, que são apresentadas na Tabela 2, Tabela 3 e Tabela 4, para introduzirem um limiar na saída dentro de uma gama pré-fixada do estado interno. Os parâmetros *k*, α, β, Θ, γ e *a* são escalares positivos.

Tabela 2 Funções de ativação linear, degrau e binária

Designação	Função	Representação
Função de ativação linear	$f(x) = kx$	
Função de ativação degrau	$f(x) = \begin{cases} \beta, & x \geq \theta \\ -\alpha, & x < \theta \end{cases}$	
Função binária ou degrau unitário	$f(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$	

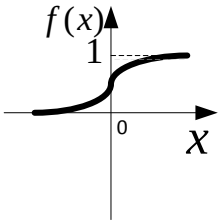
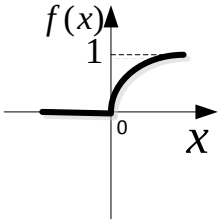
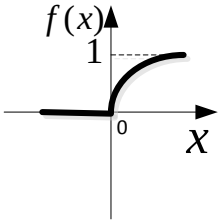
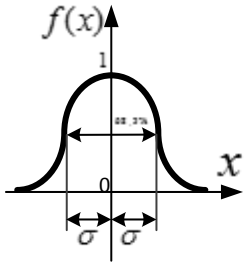
A função binária é a função de ativação por excelência no modelo do neurónio apresentado por McCulloch e Pitts e posteriormente utilizada no modelo do neurónio formal [2]. A função de ativação linear é posteriormente implementada para resolver o problema da separação linear e por excelência a função utilizada na regra delta (algoritmo de treino).

Tabela 3 Funções de ativação bipolar, rampa saturada e sigmoide

Designação	Função	Representação
Função de ativação bipolar	$f(x) = \begin{cases} 1, & x \geq 0 \\ -1, & x < 0 \end{cases}$	
Função de ativação rampa saturada	$f(x) = \begin{cases} \gamma, & x \geq \theta \\ kx, & \lvert x \rvert < \theta \\ -\gamma, & x \leq -\theta \end{cases}$	
Função sigmoide	$f(x) = b\frac{1 - e^{-ax}}{1 + e^{-ax}}$	
Função tangente hiperbólica	$f(x) = \frac{1 - e^{-x}}{1 + e^{-x}}$	

Várias das funções apresentadas na Tabela 2, Tabela 3 e Tabela 4 não são utilizadas na prática pois com a implementação das funções binária, linear, logística e/ou tangente hiperbólica, as redes neuronais atingem o funcionamento pretendido.

Tabela 4 Funções de logística, exponencial limitada e razão de quadrados

Designação	Função	Representação
Função logística	$f(x) = \frac{1}{1 + e^{-ax}}$	
Função exponencial limitada	$f(x) = \begin{cases} 1 - e^{-ax}, & x > 0 \\ 0, & x \leq 0 \end{cases}$	
Função razão de quadrados	$f(x) = \begin{cases} \frac{x^2}{1 + x^2}, & x > 0 \\ 0, & x \leq 0 \end{cases}$	
Função gaussiana	$f(x) = e^{-\frac{x^2}{2}}$	

A função de ativação de uma camada escondida de uma rede neuronal multicamada é habitualmente uma função logística enquanto a função de ativação da camada de saída da rede é uma função linear [2].

2.3. TOPOLOGIAS [2]

A forma de organização dos neurónios numa rede neuronal está intrinsecamente ligada ao problema que se quer solucionar e é um fator importante para a definição dos algoritmos de treino que serão utilizados. Estes algoritmos possuem regras específicas que influenciam a estrutura a ser adotada para atingir os objetivos da rede.

As redes neuronais podem ser classificadas pelo número de camadas dos elementos processadores:

- Rede neuronal monocamada (*single-layer*): neste caso existe somente uma única camada com os elementos processadores "completamente intraconetados" ou "parcialmente intraconetados". Foi o primeiro modelo concebido baseado no perceptrão, que revelou limitações de cálculo e veio a suscitar o desenvolvimento de sistemas com mais camadas interconectadas. Um exemplo deste tipo de rede neuronal é a apresentada na Figura 4;

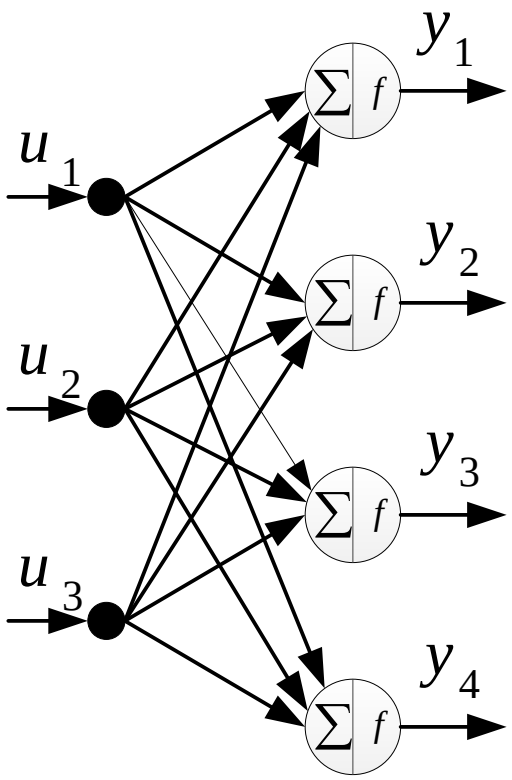


Figura 4 Rede neuronal do tipo monocamada

- Rede neuronal multicamada (*multi-layer*): este tipo de rede possui duas ou mais camadas de elementos processadores interconectados (podendo existir eventuais intraconexões), sendo uma camada de entrada situada num extremo da rede, uma ou várias camadas escondidas (a sua função é processar os sinais de entrada antes de enviá-los aos neurónios de saída), que se encontram no interior da rede, e uma camada de saída, colocada no outro extremo de rede neuronal. Apesar da maior complexidade, esta arquitetura possibilita uma melhor qualidade de treino pois há maior interação entre os neurónios. Este tipo de rede está ilustrado na Figura 5.

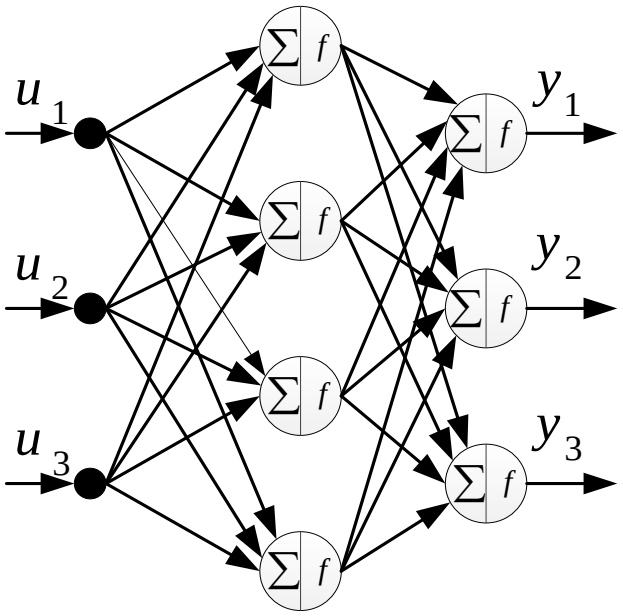


Figura 5 Rede neuronal do tipo multicamada

Além da classificação das redes neuronais através do número de camadas, existe outra classificação através da configuração das conexões entre elementos processadores nas camadas ou entre camadas sucessivas:

- Intraconexão: corresponde a uma conexão entre dois elementos processadores da mesma camada;
- Interconexão: refere-se a uma conexão entre dois elementos de processamento de camadas sucessivas. Geralmente as redes neuronais de múltiplas camadas são interconectadas.

Pode-se também classificar uma rede neuronal em relação aos sentidos de orientação das interconexões:

- Rede neuronal proactiva (*feedforwad*): as redes sem realimentação (*feedforward*) têm neurónios agrupados em camadas, o sinal percorre a rede numa única direção, da entrada para a saída e os neurónios da mesma camada não são conectados. Os valores das saídas dependem apenas das entradas e dos pesos atribuídos a essas conexões, consoante a função de ativação. Tratando-se de um controlo neuronal em malha aberta, resulta sempre uma rede estável, embora com saída imprevisível. Na Figura 6 é possível observar uma rede proactiva;

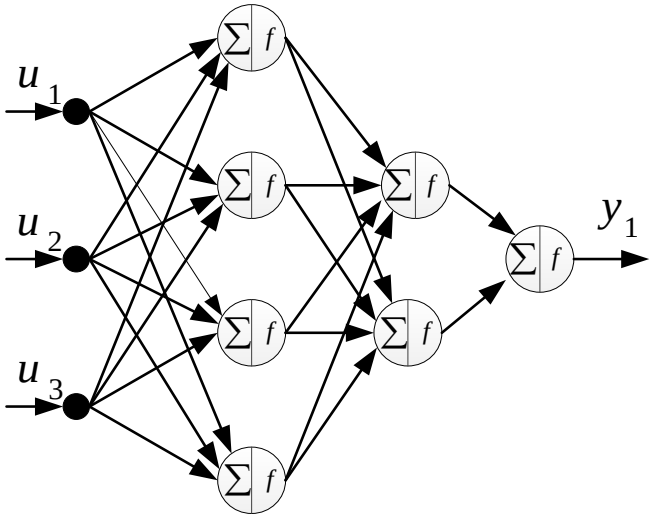


Figura 6 Rede neuronal do tipo proactiva.

- Rede neuronal retroativa (*feedback*): além das conexões proactivas, a rede neuronal contém também conexões retroativas, isto é, de um elemento processador de uma camada para os elementos processadores da camada anterior, donde resulta um controlo neuronal em malha fechada. Também designado de "rede neuronal recorrente" porque o processo é repetitivo. Nas redes com realimentação ou recorrentes, a saída de alguns neurónios são as entradas de outros neurónios da mesma camada (inclusive o próprio) ou de camadas anteriores. O sinal neste tipo de redes percorre a rede em duas direções. Outras características deste tipo de redes são terem memória dinâmica e capacidade de representar estados em sistemas dinâmicos. Um exemplo de uma rede neuronal retroativa é a rede de Hopfield, apresentada na Figura 7.

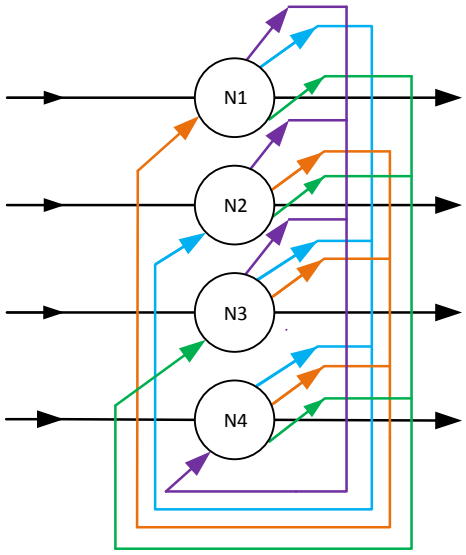


Figura 7 Rede de Hopfield de quatro neurónios

Este tipo de rede neuronal retroativa, portanto, pode funcionar recorrentemente e criar memórias associativas (usadas no reconhecimento de padrões ou de voz). Se as saídas ficarem em cálculo permanente diz-se que há instabilidade. Todavia, verifica-se que há estabilidade quando a matriz dos pesos for simétrica em relação à diagonal principal com todos os seus elementos nulos: segundo o teorema de Cohen-Grossberg, estabelecido em 1983, estas condições de estabilidade $w_{ij} = w_{ji}\, e\, w_{ii} = 0$ são suficientes mas não necessárias.

2.4. ALGORITMOS DE TREINO [3]

A característica principal de uma rede neuronal é a sua capacidade de aprender, levando a que normalmente se considerem duas fases no processo de utilização de uma rede: uma fase de treino, e uma de utilização. O treino consiste no ajuste dos parâmetros internos da rede, de maneira que a rede apresente um resultado esperado dada a apresentação de um conjunto de padrões específicos. Os padrões de treino da rede contêm as informações que se desejam que a rede aprenda. Os parâmetros a ajustar são os pesos das conexões que interligam os neurónios. Os diversos modelos de redes neuronais caracterizam-se pela utilização de diferentes técnicas de treino. O treino genericamente pode ser classificado como supervisionado ou não supervisionado.

O treino é supervisionado (Figura 8) quando o ajuste de parâmetros é feito a partir da apresentação de um conjunto de pares de entradas e saídas padrão. Neste processo uma entrada padrão é apresentada à rede e uma saída é calculada. A diferença existente entre a saída calculada e a saída padrão é o erro produzido, que se deseja minimizar.

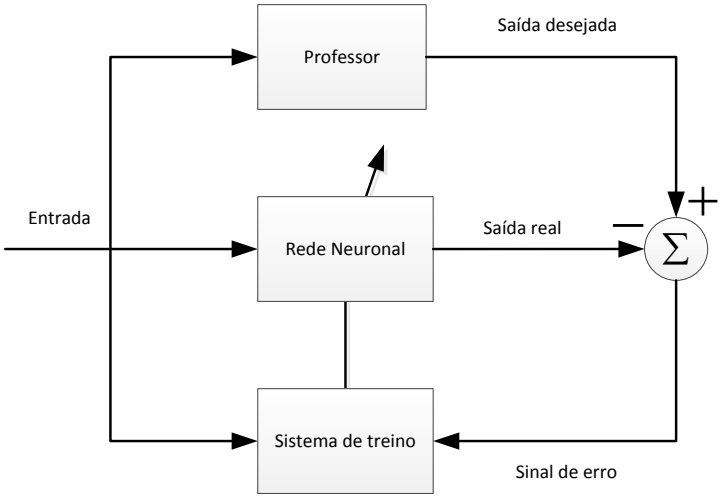


Figura 8 **Diagrama em blocos do treino supervisionado**

Neste tipo de aprendizagens são conhecidas inicialmente as respostas corretas correspondentes a um certo conjunto de dados de entrada. A referir entre outros os seguintes algoritmos de aprendizagem com supervisão [3]:

a) Regra de aprendizagem de Widrow-Hoff (ou método do gradiente aplicado em redes neuronais lineares);

b) Aprendizagem por retropropagação do erro (*error backpropagation*) que constitui uma generalização da anterior regra a redes com camadas intermédias, lineares ou não lineares;

c) Método do gradiente descendente e seus aperfeiçoamentos. De referir a existência de técnicas destinadas a melhorar a convergência destes métodos tais como a técnica do momento e do coeficiente de aprendizagem variável ou adaptativo;

d) A aprendizagem recorrendo aos métodos de aproximação, Método de Newton como por exemplo o de Levenberg-Marquardt;

e) A aprendizagem recorrendo a técnicas heurísticas, como por exemplo, os algoritmos evolutivos ou a aprendizagem recorrendo ao processo *simulated annealing*.

Normalmente o algoritmo utilizado classifica também a rede em que se aplica, sendo as redes mais divulgadas as redes *backpropagation* ou retropropagação. Estas são redes do tipo *feedforward*, com pelo menos uma camada intermédia e utilizam um algoritmo iterativo com base no gradiente descendente para ajuste dos pesos da rede de modo a minimizar o erro quadrático da camada de saída relativamente aos valores desejados.

O treino é "não supervisionado" quando o conjunto de padrões de treino possui somente entradas, ou seja não existe saída padrão, não sendo possível mostrar à rede neuronal um alvo para se alcançar. O processo utiliza a comparação entre sinais para a construção de grupos de similaridade.

A aprendizagem sem supervisão é aplicada em sistemas de memória associativa e essencialmente de reconhecimento de padrões. Nestas redes a aprendizagem é realizada sem se conhecer antecipadamente as respostas consideradas corretas. Podem ser utilizados diferentes algoritmos de aprendizagem sem supervisão, entre outros [3]:

a) Algoritmos de estimulação pela entrada (*reinforcement algorithms*), também designados (no contexto da aprendizagem sem supervisão) por algoritmos de aprendizagem associativa (*associative learning algorithms*). A regra de Hebb, as regras de Instar e Outstar [3], constituem alguns exemplos deste tipo de algoritmos;

b) Algoritmos de aprendizagem competitiva tais como a regra de Kohonen.

Considerando os diferentes algoritmos de treino atualmente disponíveis, pode-se considerar que os algoritmos do tipo *backpropagation*, desde a sua versão inicial até aos métodos que permitem evoluções na velocidade e generalização obtida através do processo de treino, continuam a ser dos mais utilizados.

2.4.1. TREINO DO PERCEPTRÃO [2]

O treino supervisionado de um elemento processador (neurónio) isolado pode ser realizado pelo algoritmo de aprendizagem do perceptrão. O perceptrão é constituído pelas entradas $\boldsymbol{u} = [u_1, u_2 \dots, u_i]^T$ e os respetivos pesos $\boldsymbol{w} = [w_{1,1}, w_{1,2} \dots, w_{1,i}]$, sendo b o peso referente à polaridade do elemento processador (Figura 2). A saída, *y*, é calculada através da função de ativação binária. Sendo a saída desejada, y^d, o erro da saída é dado por:

$$\delta = y^d - y \tag{2}$$

Para $\delta \neq 0$ o sistema apresenta erro. Para contrariar o erro, os pesos das entradas têm que ser alterados. Os pesos são alterados de forma proporcional ao erro dados pela expressão:

$$\Delta w_{1,i} = \eta \delta u_i \tag{3}$$

A variação do peso da conexão é igual ao erro obtido multiplicado pela entrada e por um fator de aprendizagem, η, que varia normalmente entre zero e um.

Assim o algoritmo de aprendizagem do perceptrão segue os seguintes passos:

1. Aplicam-se as entradas ao elemento processador.

2. Calcula-se o erro δ.

3. Reajustam-se os pesos segundo a equação (3).

Deste modo, após algumas iterações consegue-se convergir para o resultado desejado anulando o erro. Isto se o problema for linearmente separável[1]. Caso contrário o algoritmo do perceptrão não consegue convergir, mantendo-se sempre na primeira resposta sem melhoramentos.

No intuito de encontrar uma solução mais robusta de treino do perceptrão, Widrow e Hoff apresentaram, em 1960, uma nova regra denominada de regra de Widrow e Hoff [1]. Esta regra permite afastar a separação linear da classe de resposta binária. Este aperfeiçoamento deve-se ao uso da função linear $f(x) = x$ como função de activação que torna o sinal do erro na diferença entre o resultado desejado e a soma ponderada das entradas com linearidade, visto a função de ativação ser linear. O erro é calculado através da seguinte expressão (4):

$$\delta = y^d - \sum_{j=1}^{i} w_{1,j} u_j \tag{4}$$

Mantendo-se a variação do peso $\Delta w_{1,i} = \eta \delta u_i$.

Com esta regra consegue-se obter uma solução "aproximada" para problemas não linearmente separáveis. Além disso, torna mais robusta a solução de separação linear que exista, tendendo para um mínimo local de erro dado pela soma dos erros quadráticos do conjunto de amostras de treino, como apresentado pela expressão (5):

$$E = \sum_{k=1}^{q} \delta_k^2 \tag{5}$$

no caso de q iterações.

Desta forma o algoritmo de Widrow e Hoff realiza uma descida de gradiente quadrática, aproximando rapidamente o processo do erro mínimo.

[1] Um problema linearmente separável tem soluções que podem ser separadas por uma reta quando apresentadas num hiperplano.

2.4.2. TREINO EM REDES MONOCAMADA [2]

Numa rede monocamada as entradas excitam os vários neurónios e o número de saídas é igual ao número de elementos processadores (Figura 9).

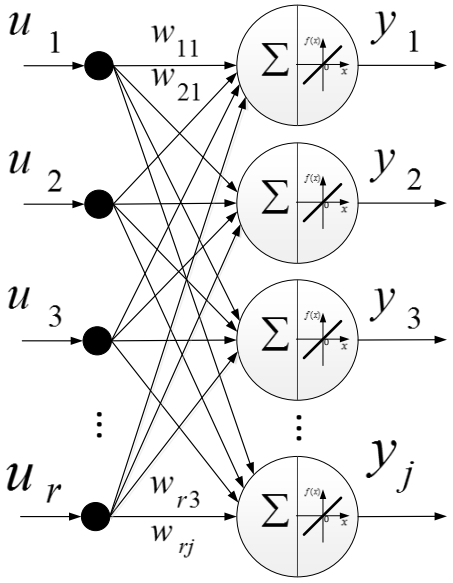


Figura 9 Arquitetura de uma rede neuronal monocamada onde é possível aplicar o algoritmo de treino - regra delta

Assim, sendo $\boldsymbol{u} = [u_1, u_2 \ldots, u_r]^T$ o vetor de entradas, $\boldsymbol{w_{rj}}$ os pesos de cada entrada, onde *j* é o número do elemento processador da entrada *r* a que o peso está associado.

Assim, numa rede de *j* elementos com *r* entradas, tem-se a matriz de pesos

$$\boldsymbol{w} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1\mathrm{j}} \\ w_{21} & w_{22} & \cdots & w_{2\mathrm{j}} \\ \vdots & \vdots & \vdots & \vdots \\ w_{\mathrm{r1}} & w_{\mathrm{r2}} & \cdots & w_{\mathrm{rj}} \end{bmatrix} \tag{6}$$

e o vetor de saída $y = \left[y_1, y_2 \ldots, y_j\right]^T$.

Para esta organização estrutural dos elementos processadores o algoritmo de treino Widrow e Hoff é insuficiente. Pelo que no treino de uma monocamada usa-se a regra delta, que não é mais do que uma generalização da regra de Widrow e Hoff. Esta regra é um algoritmo de treino supervisionado aplicável a redes neuronais proactivas monocamada com uma função de ativação linear $f(x) = x$.

O erro de um dado elemento da camada, j, para a saída desejada y_j^d é calculado da seguinte forma:

$$\delta_j = y_j^d - y_j \tag{7}$$

Sendo a variação do peso de cada entrada, para um elemento processador específico com o fator de aprendizagem de η, dada pela seguinte fórmula:

$$\Delta w_{ij} = \eta \delta_j u_i \tag{8}$$

A variação do peso calculado é assim adicionada ao peso existente através da seguinte expressão:

$$w_{ij}(k+1) = w_{ij}(k) + \Delta w_{ij}(k) \tag{9}$$

para cada iteração *k*.

Os pesos das entradas são ajustados sucessivamente para cada elemento processador, de modo a que a saída convirja para a resposta correta. A dinâmica deste algoritmo segue os passos seguintes:

1. Inicializar a rede com pesos aleatórios de valor reduzido.
2. Aplicar o vetor de entrada $\boldsymbol{u}$ ao conjunto de treino.
3. Calcular as saídas dos elementos processadores para a entrada aplicada.
4. Determinar o erro δ_j de cada saída.
5. Ajustar o peso das conexões segundo (9).
6. Repetir os passos para todos os vetores de entrada até se obter um erro aceitável.

Através deste algoritmo o treino das redes neuronais monocamada é facilmente realizado. No entanto as redes neuronais monocamada são limitadas. Tal como foi demonstrado por Minsky, que provou que uma rede monocamada não consegue modelar uma porta lógica XOR com sucesso. Assim para ultrapassar esta limitação foram desenvolvidas as redes neuronais multicamada.

2.4.3. TREINO EM REDES MULTICAMADA [2]

Uma rede multicamada é constituída por várias camadas consecutivas de elementos de processamento. Na Figura 10 é representado um esquema de uma rede multicamada.

Para treinar uma rede multicamada usa-se a técnica de retropropagação. O princípio fundamental desta técnica reside no uso de uma função de ativação derivável. O algoritmo serve-se das regras de derivação composta. O que lhe permite propagar os sinais no sentido proactivo, da entrada para a saída tal como no sentido retroativo, com a retropropagação do erro da saída para as camadas anteriores.

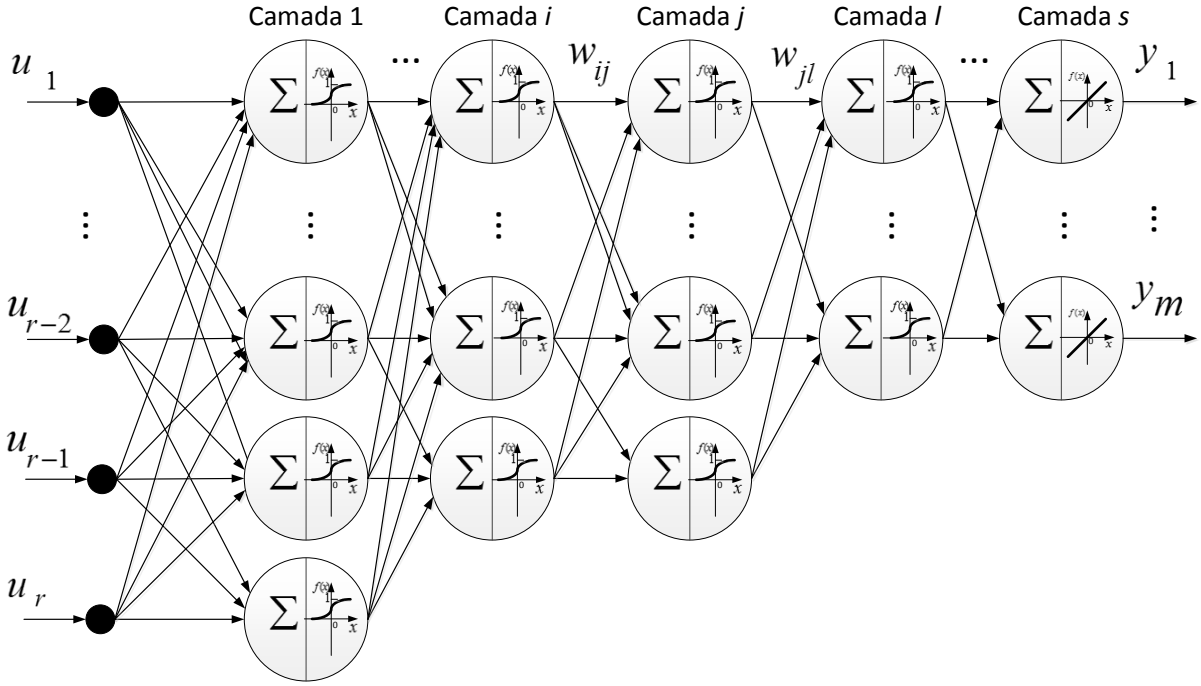


Figura 10 Arquitetura de um sistema neuronal multicamada

Através destes princípios é atingido o objetivo de minimizar o erro quadrático em relação a um dado peso por uma descida de gradiente. Para tal é aplicado o "princípio da descida de gradiente" onde a variação de cada peso é proporcional à derivada parcial do erro em relação a esse peso multiplicada pelo fator η:

$$\Delta w_{ij} = \eta \frac{d\delta_j}{dw_{ij}} \tag{10}$$

Normalmente, para este algoritmo escolhe-se a função sigmoide logística, visto esta ser derivável e de declive controlável. Assim, na saída de um elemento processador tem-se a função logística em que a sua derivada é a equação (11):

$$\frac{dy}{dx} = y(1-y) \tag{11}$$

Sendo o erro quadrático dado pela expressão (12):

$$\delta_j = (y_j^d - y_j)^2 \tag{12}$$

Define-se a função energética de Lyapunov, apresentada na equação (13), a qual se pretende minimizar a cada iteração efetuada do treino, até atingir um valor aceitável.

$$E_j = \frac{1}{2}(y_j^d - y_j)^2 \tag{13}$$

A determinação do erro faz-se pelo cálculo dos erros parciais em cada elemento do conjunto de treino, efetuando a soma das alterações parciais obtidas para cada entrada com fim à alteração final de cada peso. Interessa saber a variação do erro em função de um peso de uma entrada específica. O erro do elemento *j* varia em função da saída do elemento, que depende do valor do peso de cada entrada, podendo assim ser apresentado pela expressão $E_j[\mathrm{y}_j(w_{ij})]$. Assim a derivada $\frac{dE_j}{dw_{ij}}$ pode ser expandida resultando na expressão (14):

$$\frac{dE_j}{dw_{ij}} = \frac{dE_j}{dy_j} \times \frac{dy_j}{dw_{ij}} \tag{14}$$

A expansão de $\frac{dE_j}{dw_{ij}}$ permitirá o cálculo da variação dos pesos das conexões representadas pelo símbolo Δw_{ij}.

Como $\mathrm{y}_j = f(x_j)$, ou seja a saída é igual ao valor da função de ativação para o estado de excitação x_j e como o estado de excitação é dado pelo somatório do produto das entradas (saídas da camada anterior) do elemento processador pelos respetivos pesos, $x_j = \sum w_{ij} y_i$ tem-se:

$$\frac{dy_j}{dw_{ij}} = \frac{df(x_j)}{dx_j} \times \frac{dx_j}{dw_{ij}} = \frac{df(x_j)}{dx_j} y_i \tag{15}$$

Ou seja, a variação da saída y_j em relação aos pesos das conexões é igual à derivada da função de ativação $f'(x_j)$ a multiplicar pelas saídas da camada anterior y_i.

Resta o cálculo $\frac{dE_j}{dy_j}$, que faz intervir as sucessivas camadas, onde as alterações das saídas dos elementos processadores da camada *j* propagam-se às saídas dos elementos da camada l, ou seja, o erro de uma camada deve-se à saída dessa camada que depende das saídas da camada anterior visto estas serem entradas nesta, isto é $E_l[\mathrm{y}_l(y_j)]$. Assim, paralelamente à camada anterior, obtém-se a expressão (16):

$$\frac{dE_j}{dy_j} = \sum_l \frac{dE_l}{dy_l} \times \frac{dy_l}{dy_j} \tag{16}$$

Mas sendo $y_l = f(x_l)$ e $x_l = \sum_l w_{jl} y_j$ obtém-se analogamente:

$$\frac{dy_l}{dy_j} = \frac{df(x_l)}{dx_l} \times \frac{dx_l}{dy_j} = \frac{df(x_l)}{dx_l} w_{jl} \tag{17}$$

Assim a variação dos pesos, atendendo ao cálculo efetuado para $\frac{dE_j}{dw_{ij}}$, é dada por:

$$\Delta w_{ij} = \eta \frac{dE_j}{dy_j} \times \frac{df(x_j)}{dx_j} y_i \tag{18}$$

Que para uma função de ativação sigmoide logística resulta:

$$\Delta w_{ij} = \eta B_j \times\ y_j(1 - y_j) y_i \tag{19}$$

Onde $B_j = \frac{dE_j}{dy_j}$, é definido como o "benefício". A equação do "benefício" depende da localização do elemento processador na rede. Assim, para elementos processadores intermédios vem:

$$B_j = \frac{dE_j}{dy_j} = \sum_l \frac{dE_l}{dy_l} \times \frac{df(x_l)}{dx_l} w_{jl} = \sum_l B_l \times y_l(1 - y_l) w_{jl} \tag{20}$$

E para elementos processadores finais $B_s = \frac{dE_s}{dy_s}$, onde $s = 1, \ldots, m$ (Figura 10) vem:

$$B_s = y_s^d - y_s \tag{21}$$

Supondo que o fator η anule o -2 de $\frac{dE_s}{dy_s} = -2(y_s^d - y_s)$.

Após a análise matemática do algoritmo descrevem-se os passos da sua implementação:

1. Aplicar o vetor de entrada de um par de treino à rede.
2. Calcular a saída do sistema.
3. Calcular o "benefício" para os elementos processadores finais.
4. Calcular o "benefício" para os elementos processadores intermédios.
5. Calcular a variação de cada peso.
6. Redefinir os pesos segundo $w_{ij}(k+1) = w_{ij}(k) + \Delta w_{ij}(k)$, para a iteração *k*.
7. Repetir todos os passos para cada vetor do conjunto de treino até se obter um erro admissível.

O desempenho deste algoritmo melhora significativamente quando à função de ativação usada nos elementos processadores é adicionada uma polaridade *b*. Com esta técnica a convergência do treino para o erro desejado é mais rápida. Outro método de acelerar o algoritmo de treino é pela inserção de um fator α, que varia normalmente de 0 até 1, designado de "momento". Este fator adiciona ao ajuste dos pesos uma parcela proporcional à variação do peso na iteração anterior, isto é:

$$w_{ij}(k+1) = w_{ij}(k) + \Delta w_i(k) + \alpha \Delta w_{ij}(k) \qquad (22)$$

Este fator corresponde à adição de um momento de inércia, que reforça a tendência e evita grandes oscilações nos pesos. Isto é vantajoso quando o declive do erro é muito baixo ou muito elevado.

Apesar da grande eficiência do algoritmo de retropropagação, este pode apresentar dificuldades para o fenómeno de "paralisia da rede". Este fenómeno acontece quando no treino se atingem pesos elevados que colocam os elementos processadores a operar em zonas da função de ativação de derivada muito baixa. Este efeito é evitado reduzindo o fator de aprendizagem, η, aumentando deste modo o tempo de treino da rede.

Atualmente existem diversas variantes deste algoritmo que usam vários métodos de melhoria, com o objetivo da otimização dos tempos de aprendizagem e do número de camadas escondidas das diferentes topologias multicamada.

2.5. ÁREAS DE APLICAÇÃO DAS REDES NEURONAIS [4]

As redes neuronais apresentam várias aplicações em diferentes áreas entre as quais se encontram as aplicações na área aeroespacial, automóvel, eletrónica, fabrico, telecomunicações e robótica.

Na área aeroespacial as redes neuronais são implementadas em pilotos automáticos para aviões de alto desempenho, simulação de trajeto de voos, sistemas de controlo do avião, melhoramentos no piloto automático, simulação de componentes de aviões e detetores de falhas em componentes dos aviões.

Já na área automóvel e dos transportes, os sistemas de navegação automática automóvel e analisadores de garantia de atividade podem conter redes neuronais. Estas também são aplicadas em sistemas de diagnóstico de travões dos camiões, de calendarização de veículos e de sistemas de traçado de rotas.

Na área bancária e financeira as redes neuronais existem em leitores de cheques e outros documentos e são também utilizadas como avaliadores de crédito. Também efetuam avaliação imobiliária, aconselhamentos em empréstimos, triagem de hipotecas, avaliação de títulos corporativos, análise do uso de linha de crédito, análise financeira corporativa e previsão do preço da moeda.

Na área da defesa militar as redes neuronais são implementadas com o objetivo de executarem a orientação de armas, seguimento de alvos, distinção de objetos e reconhecimento facial. Também fazem parte de novos tipos de sensores, sonares, radares e sistemas de processamentos de sinais de imagem, incluindo compressão de dados, extração de características, supressão de ruído e identificação de sinal/imagem.

Na área da eletrónica as redes neuronais são utilizadas como sistemas de previsão da sequência de execução de código, controlo de processos, análise de falhas nos circuitos integrados, visão artificial, síntese de voz e modelação não-linear.

Já na área do fabrico as redes neuronais implementam o controlo do processo de fabrico, projeto e análise de produtos, diagnóstico de processos e máquinas, identificação de partículas em tempo real, sistemas de visão para inspeção de qualidade, teste de cervejas, análise da qualidade da soldadura, previsão da qualidade do papel, análise da qualidade de integrados para computadores, análise de operações de moagem, análise de produtos químicos, análise de manutenção de máquinas, licitação de projetos, planeamento e gestão e modelação dinâmica de sistemas de processamento químico.

Na área médica as redes neuronais são implementadas para efetuarem a análise de células do cancro da mama, análise de eletroencefalograma (EEG) e eletrocardiograma (ECG), projeto de próteses, otimização dos tempos de transplante, a redução das despesas hospitalares, o melhoramento da qualidade hospitalar e o aconselhamento de testes médicos em salas de emergência.

Na área da robótica as redes neuronais são aplicadas no controlo de trajetórias, em robôs empilhadores, no controlo de manipuladores e em sistemas de visão.

Na área da fala as redes neuronais permitem efetuar o reconhecimento da fala, a compressão da fala, a classificação de vogais e a síntese de fala a partir de texto.

Nas telecomunicações as redes neuronais são utilizadas na compressão de imagem e de dados, nos serviços automáticos de informação, em sistemas de tradução em tempo real de linguagem falada e sistemas de processamento de pagamentos de clientes.

2.5.1. ROBÔ ASPIRABOT [5]

Nesta subsecção é apresentado um exemplo o mais detalhado possível da aplicação de redes neuronais nas áreas em que esta tese se enquadra, controlo e robótica.

O aspeto do robô AspiraBot, onde a rede neuronal foi implementada, está apresentado na Figura 11.

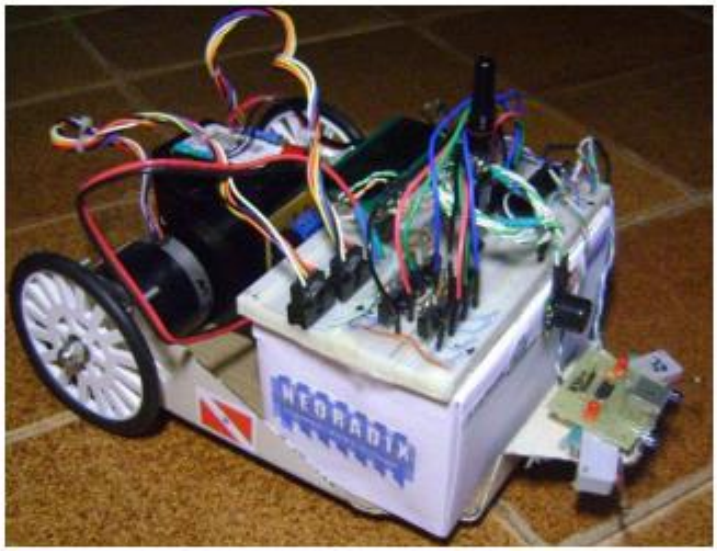


Figura 11 Robô AGV/ROVER Aspirabot

O AspiraBot é um AGV que se desvia de obstáculos que se encontrem na direção em que se desloca. Para detetar o ambiente, o AspiraBot é constituído por um sonar e dois detetores/sensores de fim de curso colocados na parte frontal do robô, como apresentado na Figura 12.

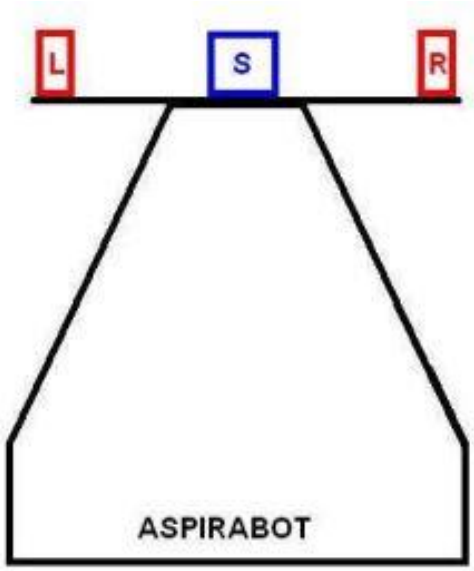


Figura 12 Esquema da disposição dos sensores no AspiraBot

Para o controlo do robô foi implementada uma rede neuronal com duas saídas e quatro entradas, incluindo a polarização. Esta rede neuronal efetua a decisão da ação que o robô

deve tomar dependendo dos valores lidos pelos sensores. A interpretação dos valores de saída da rede neuronal é mostrada na Tabela 5.

Tabela 5 Relação de saída da rede neuronal e sua interpretação

Saída da rede neuronal	Interpretação
11	Obstáculo à frente
10	Obstáculo à esquerda
01	Obstáculo à direita
00	Sem obstáculos

Como apresentado na Tabela 5 se o valor nas saídas da rede neuronal for '11' significa que tem um obstáculo à frente logo a decisão a tomar deve ser desviar-se. Todas as combinações possíveis entre as entradas e as saídas da rede neuronal são apresentadas na Tabela 6.

Tabela 6 Tabela de todas as entradas e saídas possíveis da rede neuronal do AspiraBot

Sonar	Fim de Curso Esquerda	Fim de Curso Direita	Neurónio de Saída 1	Neurónio de Saída 2
1	1	1	1	1
1	1	0	1	0
1	0	1	0	1
1	0	0	1	1
0	1	1	1	1
0	1	0	1	0
0	0	1	0	1
0	0	0	0	0

Como apresentado na Tabela 6, se os valores lidos dos sensores (sonar e fins de curso) for '111', ou seja todos os sensores detetaram o obstáculo a saída da rede neuronal deverá ser '11', obstáculo à frente. Por outro lado, se os valores de entrada da rede neuronal forem '010', a saída da rede neuronal deve ser '10', obstáculo à esquerda. Devido à redundância das entradas em que o sonar tem valor 1 e os fins de curso também são ativados estas entradas não são consideradas na versão final da rede. Assim, seguindo o raciocínio anterior, as relações entradas/saídas utilizadas no treino são as apresentadas na Tabela 7.

Tabela 7 Tabela das entradas e saídas da rede neuronal implementada no AspiraBot

Sonar	Fim de Curso Esquerda	Fim de Curso Direita	Neurónio de Saída 1	Neurónio de Saída 2
1	0	0	1	1
0	1	0	1	0
0	0	1	0	1
0	1	1	1	0

Deste modo, a rede neuronal deverá responder exatamente como previsto às entradas apresentadas na Tabela 7 com os valores das saídas correspondentes.

O diagrama da rede neuronal implementada é apresentado na Figura 13, onde se pode verificar que existem 3 entradas, além da polarização (*b*) que são o sonar (*S*), o fim de curso esquerdo (*L*) e o fim de curso direito (*R*). A rede neuronal é uma rede monocamada com 2 neurónios, logo duas saídas.

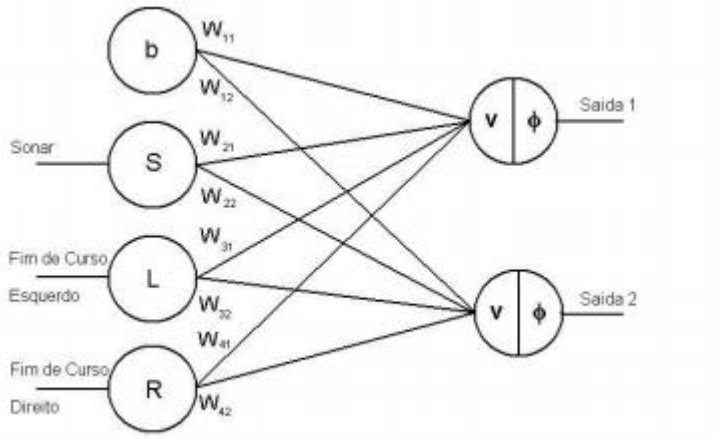


Figura 13 Representação da rede neuronal implementada no AspiraBot [5]

O algoritmo de controlo da rede neuronal implementada, apresentado na Figura 14, efetua a leitura dos sensores, entradas da rede neuronal, calcula o valor de entrada na função de ativação de cada neurónio através do somatório pesado das entradas desse neurónios e dos seus pesos. Posteriormente verifica se esse valor de entrada ativa ou não a função de ativação. A função de ativação usada no AspiraBot, para cada um dos neurónios, é uma função binária que toma o valor 1 se a entrada for superior ou igual a 0 e 0 se a entrada for inferior a 0. Após o cálculo dos valores de saída de cada neurónio é chamada a função de evasão do obstáculo dependendo dos valores da saída da rede neuronal. Por exemplo, se o

valor de saída da rede neuronal for ‘01’ então é chamada a função de evasão da direita, ou seja vira para a esquerda.

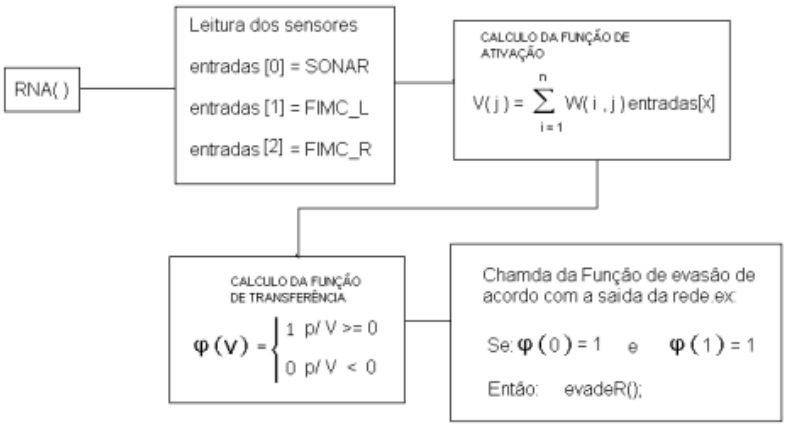


Figura 14 Fluxograma da rede neuronal implementada no AspiraBot [5]

Na Figura 15 é apresentado o diagrama do algoritmo de treino da rede neuronal implementada no AspiraBot. O algoritmo calcula o valor de saída para um conjunto de entradas e enquanto o valor da saída da rede neuronal não for igual ao valor da saída desejada para essa entrada, faz o ajuste dos pesos das entradas através da regra delta.

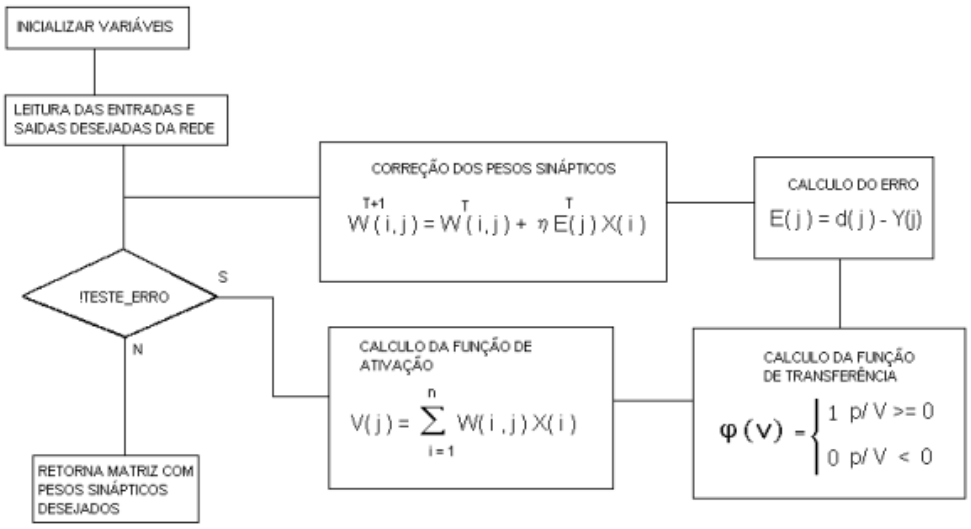


Figura 15 Diagrama do algoritmo para o treino da rede neuronal do AspiraBot [5]

A Figura 16 mostra que o algoritmo de treino aplicado na rede neuronal obteve na iteração 15 um erro nulo como pretendido.

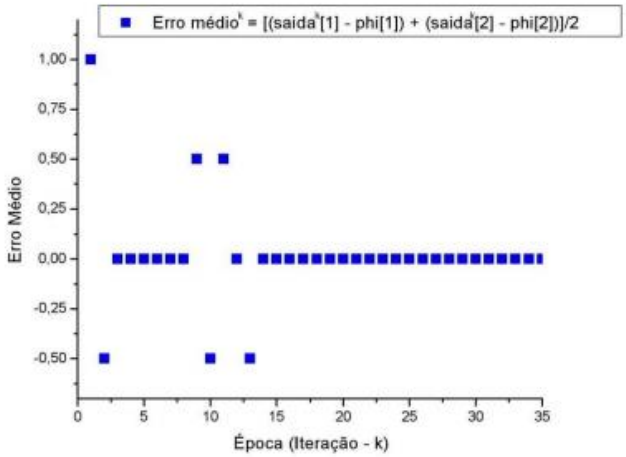


Figura 16 Erro médio gerado em cada iteração da rede neuronal implementada no AspiraBot [5]

Os pesos resultantes do algoritmo de treino são os apresentados na matriz $W_{i,j}$ (Figura 17) sendo cada coluna um neurónio e a primeira linha o valor do peso da polarização e as outras linhas são os valores dos pesos de cada uma das entradas desse neurónio.

$$W_{i,j} = \begin{bmatrix} -1.0 & 2.0 \\ 2.0 & 1.0 \\ 2.0 & -2.0 \\ 1.0 & 0.0 \end{bmatrix}$$

Figura 17 Matriz dos pesos sinápticos utilizada no Aspirabot [5]

A posterior implementação desta rede neuronal no Aspirabot obteve um bom desempenho, pois gerava as respostas necessárias aos estímulos que recebia do ambiente, ou seja, desviava-se de obstáculos.

O robô Aspirabot (Figura 11) foi o resultado de um projeto de uma *startup* no Brasil que obteve notoriedade pela sua versatilidade, sendo notícia em emissoras de TV local e jornais impressos. Atualmente a página de internet da emprese referida neste projeto já não refere nenhuma informação sobre este projeto, mas continua com o objetivo de desenvolver sistemas autónomos para utilizar em casa.

3. PLATAFORMA DO AGV E SUAS FUNCIONALIDADES

Neste capítulo é apresentado o AGV que foi utilizado para obter os dados de treino para a rede neuronal que irá ser criada. No robô será implementada a rede neuronal e posteriormente feitos testes experimentais ao seu funcionamento.

O robô que foi utilizado foi desenvolvido na tese "Controlo Difuso de um AGV" efetuada em 2010 por Dário Osório nesta mesma instituição [1]. O robô desenvolvido nesta tese tinha como funcionalidades gerais o seguimento de paredes e desviar-se de obstáculos. Como o título indica foi implementado um controlo difuso o qual irá ser substituído por um controlo através de uma RNA.

3.1. ARQUITETURA DO AGV

O robô é constituído por três microcontroladores PIC18F4585 interligados através de uma rede CAN (Figura 18). Cada um dos microcontroladores tem uma tarefa diferente mas que se integra com as tarefas de cada um dos outros microcontroladores, diretamente ou indiretamente. A arquitetura detalhada do AGV é apresentada na Figura 18.

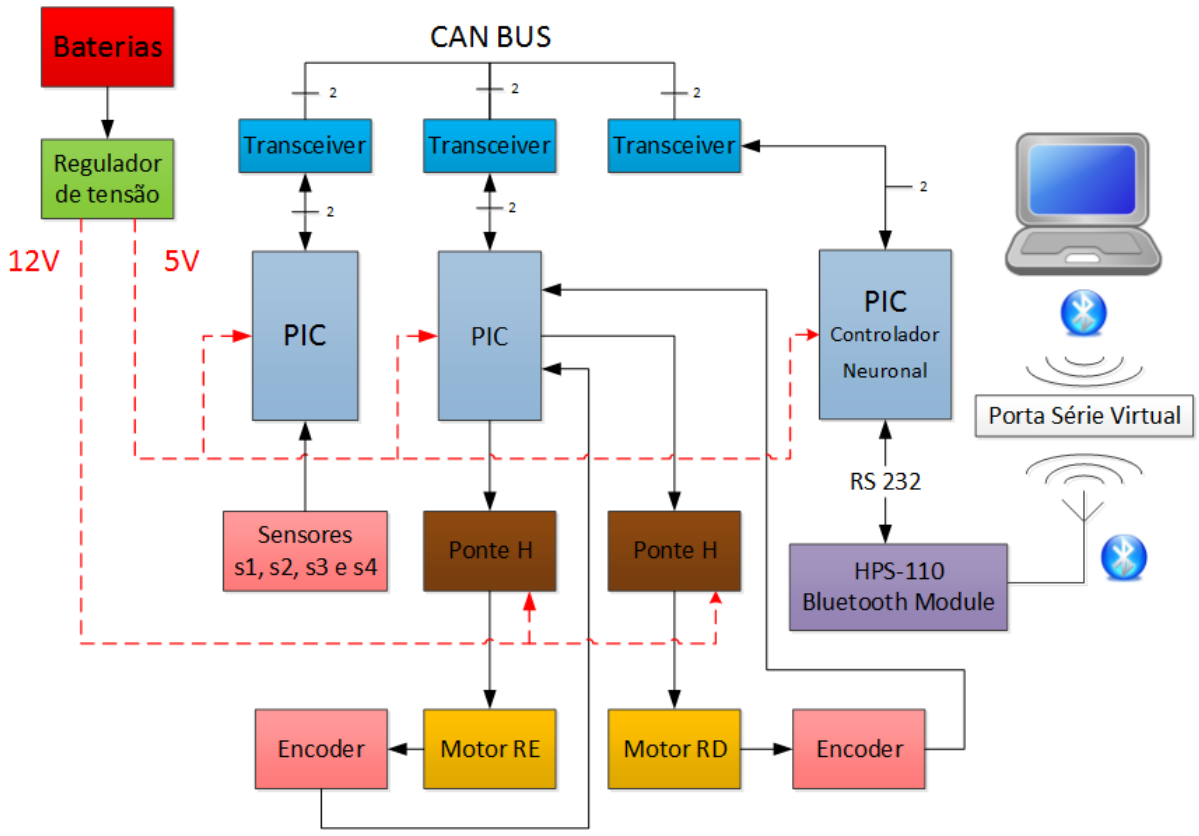


Figura 18 Arquitetura do AGV

Um dos microcontroladores efetua o envio dos comandos e leitura das medições dos sensores de ultrassons (vulgarmente designados de sonares). Posteriormente, os valores medidos são enviados via CAN para o microcontrolador que contém o controlo através da rede neuronal. Outro dos microcontroladores efetua o controlo dos motores do robô através de um controlador PI que recebe os valores de referência via CAN enviados pelo controlador difuso/neuronal. Por fim o microcontrolador que contém o controlador neuronal recebe as medições efetuadas e a partir desses dados são calculados os valores da velocidade linear e velocidade angular a que os motores do robô terão de rodar. O módulo que contém o controlador neuronal também tem anexado um módulo Bluetooth, o modelo HPS-110 da HandyWave, que permite enviar e receber dados remotamente. A comunicação entre o módulo Bluetooth e o módulo do controlador difuso/neuronal é efetuado através de RS-232. Neste trabalho esta funcionalidade não constava do AGV inicial, sendo adicionada para permitir uma melhor interação com o AGV.

A tração implementada no robô é a tração diferencial por possuir um modelo cinemático simples e um funcionamento satisfatório.

3.2. SISTEMA GLOBAL

O sistema é composto por três módulos distintos responsáveis por efetuar o controlo de velocidade dos motores, aquisição de dados provenientes dos sensores e efetuar o controlo por uma RNA do sistema onde estão implementados os vários comportamento**s:** seguir paredes, desviar de obstáculo e paragem de emergência.

É apresentado na Figura 19 um esquema representativo da distribuição modular e da interação entre os vários módulos do sistema. Analisando o esquema, notam-se as várias dependências entre os módulos; por exemplo, o módulo de controlo de velocidade do sistema necessita das saídas do módulo que implementa o controlador neuronal, ou seja, as velocidades de referência dos motores direito e esquerdo. Por sua vez, o controlador difuso/neuronal necessita na sua entrada dos valores medidos pelos sensores, sendo estes obtidos do módulo de aquisição de dados dos sensores e enviados para o módulo do controlador neuronal.

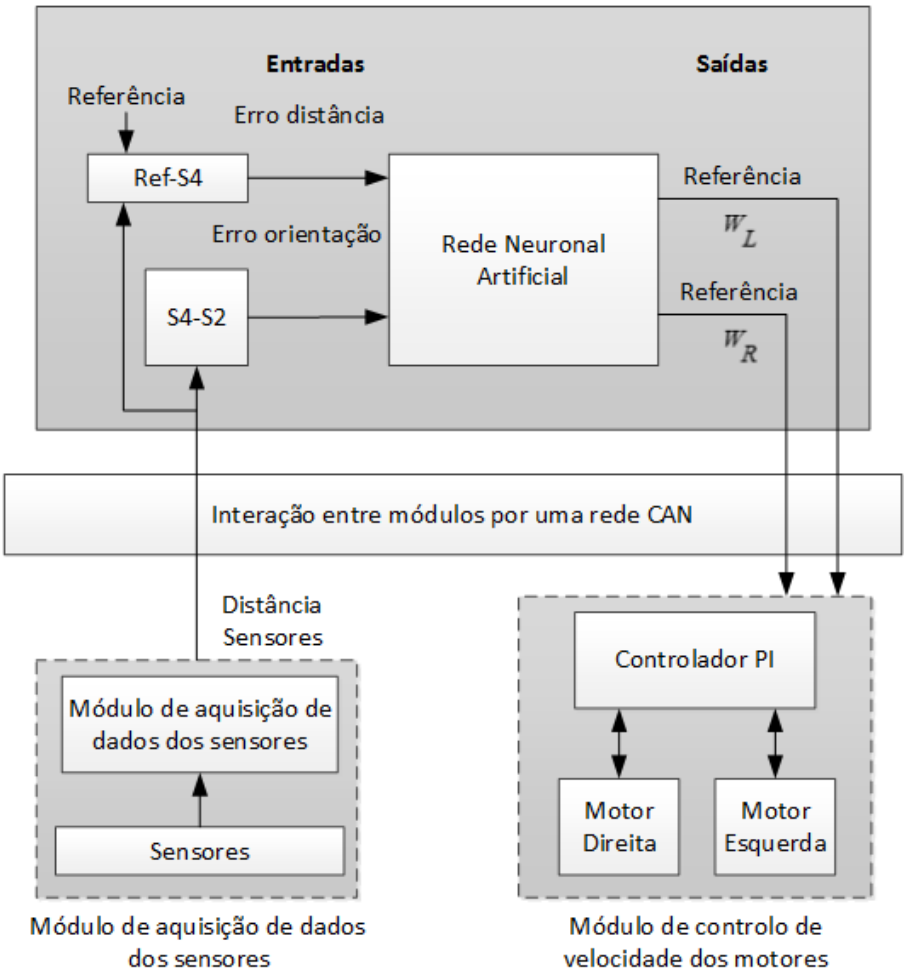


Figura 19 Esquema representativo da interação entre os vários módulos do sistema

O desenvolvimento de um sistema modular permite uma maior flexibilidade do sistema, permitindo implementar futuras evoluções e melhorias com relativa facilidade. A divisão das tarefas por diferentes módulos permite também simplificar a implementação e desenvolvimento de código para o sistema, visto que cada módulo executa uma tarefa específica.

Na Figura 20 é apresentado um esquema que representa o sistema a ser controlado pela rede neuronal a ser implementada através dos quatro sonares e por dois motores que acionam as rodas do AGV. Inicialmente pretende-se que a rede neuronal faça o AGV comportar-se da mesma forma que se comportava quando estava a ser controlado por lógica difusa.

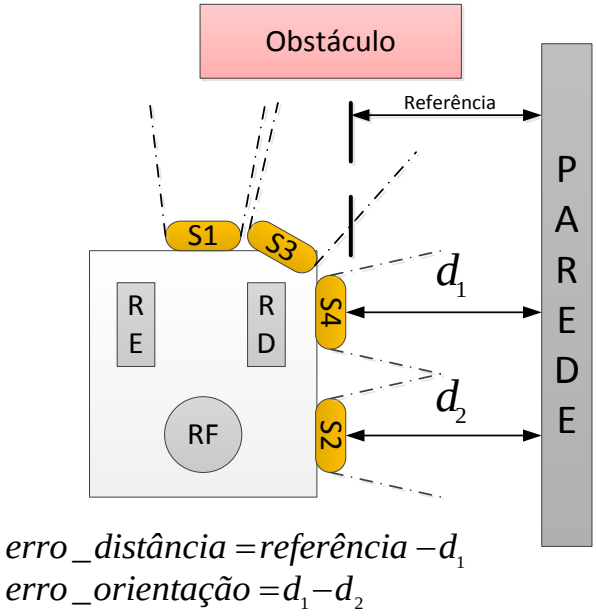


Figura 20 Estrutura do AGV testado

O módulo responsável pela aquisição de dados provenientes dos sensores irá enviar os dados referentes a cada sensor, assim que terminar a leitura dos quatro sensores, através de uma mensagem com o formato previsto no protocolo CAN. O módulo do controlador neuronal, ao verificar que existe uma mensagem com o ID 5, sabe que esta é proveniente do módulo dos sensores e que a informação contida na mensagem é necessária e por isso recebe-a.

Após se terem efetuado todas as tarefas do controlador neuronal são obtidos os valores da velocidade angular para cada motor a partir dos valores de saída do controlador. Estas velocidades são os valores de referência utilizados pelos controladores PI; deste modo é necessário pegar nesta informação e colocá-la numa mensagem CAN com o ID2, e enviá-la para a rede. O módulo de controlo de velocidade reconhece o identificador e recebe a

mensagem e, em seguida, atualiza as variáveis de modo a serem utilizadas no próximo ciclo de controlo.

3.3. MÓDULOS DO AGV

Nesta secção são apresentados brevemente os vários componentes do AGV utilizado e que nas secções seguintes irão ser detalhadamente apresentados.

O AGV é apresentado na Figura 21, onde se destacam as três *Printed Circuit Board* (PCB) existentes que o constituem (cada uma alojando um microcontrolador), quatro sonares, dois na parte lateral e dois na parte frontal e ainda as rodas que impulsionam o AGV.

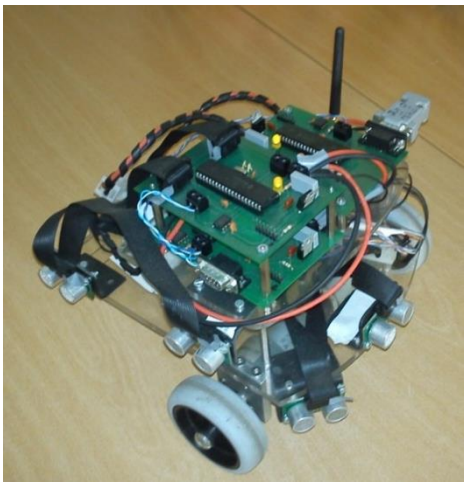


Figura 21 Fotografia do AGV utilizado

Na Figura 22 é apresentado o módulo Bluetooth que permite a comunicação remota com o AGV. O módulo Bluetooth está conectado à PCB que aloja o controlo central do robô através de uma comunicação série RS-232.

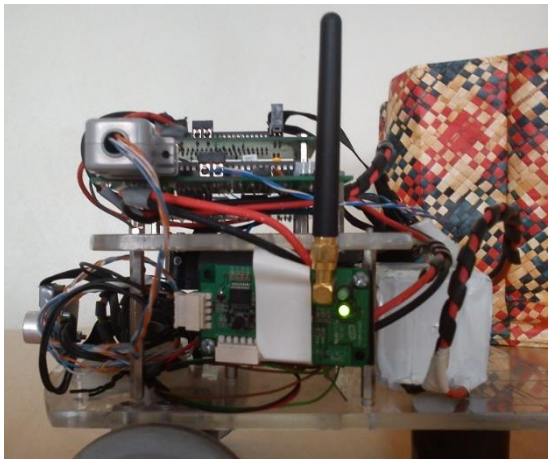


Figura 22 Fotografia do módulo Bluetooth utilizado

Na Figura 23 é apresentada uma fotografia do módulo de controlo central onde é implementado o controlador difuso ou o controlador neuronal. Este módulo está ligado ao barramento CAN permitindo assim comunicar com os outros módulos integrantes do AGV e pode efetuar comunicações remotas através de um módulo Bluetooth.

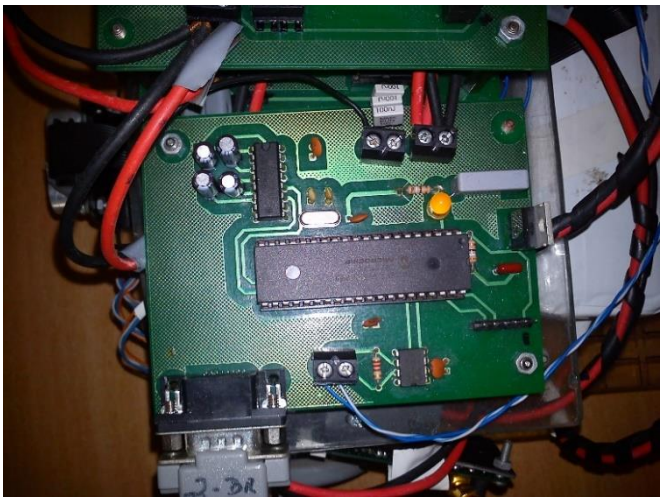


Figura 23 Módulo de controlo central do AGV

Na Figura 24 é apresentada uma fotografia que mostra a localização dos sonares frontais e dos motores utilizados para impulsionar as rodas e, consequentemente, o AGV em si. Como se pode observar na fotografia, um dos sonares está situado no centro da parte frontal do AGV permitindo detetar obstáculos que se situem frontalmente ao AGV. O outro sonar situado frontalmente está situado na zona lateral direita e inclinado para permitir a deteção de obstáculos que não obstruem completamente o trajeto do AGV.

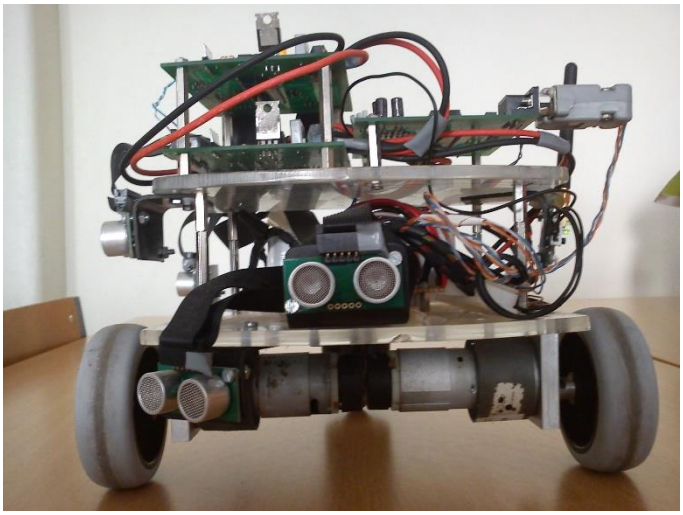


Figura 24 Localização dos motores e sonares frontais do AGV

Na Figura 25 é apresentada uma fotografia que mostra o módulo conectado aos sonares que efetua o comando e leitura das medições efetuadas. Como se pode verificar pela fotografia, o módulo tem como conexões os fios da alimentação proveniente da bateria, os fios do barramento CAN e dois *flat cables* que interligam o microcontrolador e os sonares.

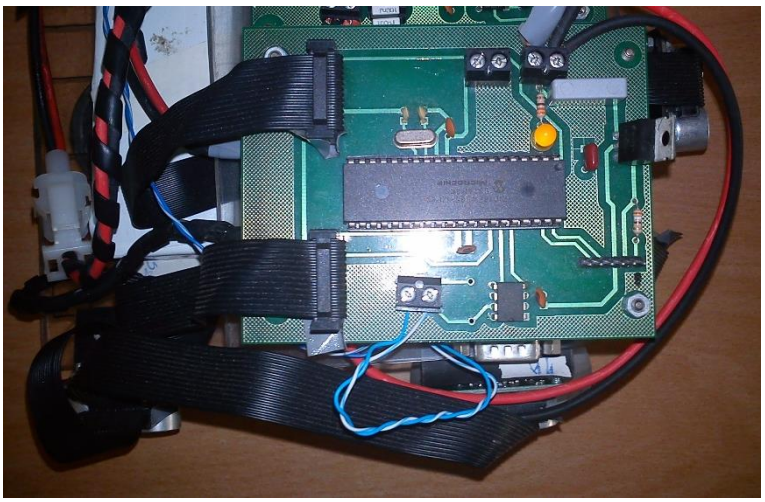


Figura 25 Módulo conectado aos sonares

3.4. REDE CAN

A rede CAN foi implementada com o intuito de interligar os vários módulos que constituem o AGV, mas também pela flexibilidade que esta rede traz ao sistema, pois facilita a introdução de futuros módulos no sistema. A rede implementada tem como função promover a interação entre os vários nós, sendo responsável pela difusão das mensagens.

Uma vez que não existe um número elevado de identificadores, ou seja de mensagens, não seria uma grande preocupação a taxa de comunicação. De qualquer modo optou-se por uma taxa de comunicação de 444 kbps.

Cada nó envia e recebe apenas as variáveis que lhe interessam, podendo-se utilizar os 8 bytes disponíveis na mensagem CAN para o envio das variáveis que se pretende, desde que se tenha em atenção o formato de cada uma delas.

De seguida são referidos os formatos das mensagens que cada nó envia, ficando assim definido o formato das mesmas como informação para futuras evoluções do sistema.

O nó dos sensores envia mensagens com o identificador número 5 (ID5), em que os 8 bytes da mensagem CAN estão divididos da forma representada na Figura 26.

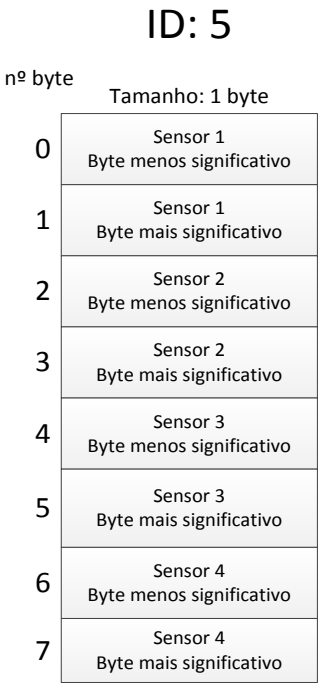


Figura 26 Mensagem enviada pelo nó que efetua a leitura dos sonares

O nó do controlador neuronal envia mensagens com o identificador número 2 (ID2), em que os 8 bytes da mensagem CAN estão divididos da forma representada na Figura 27.

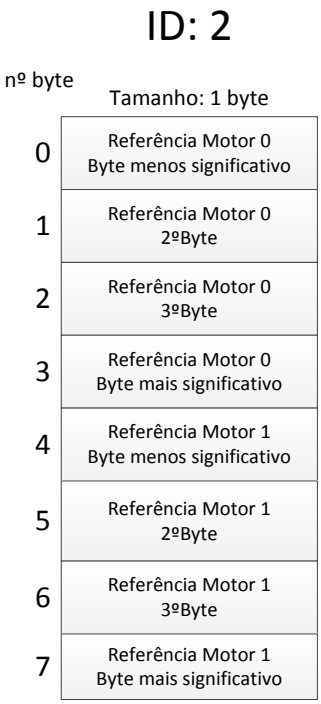


Figura 27 Mensagem enviada pelo nó do controlador difuso/neuronal para controlo dos motores

O controlador neuronal recebe as mensagens provenientes dos outros módulos, utilizando esses dados para calcular os valores de controlo para o sistema. De seguida, envia uma mensagem para o módulo de controlo dos motores com os valores de referência das velocidades para cada motor.

3.5. LEITURA DOS SONARES

Como é apresentado na arquitetura de *hardware* do sistema (Figura 18), um dos microcontroladores é responsável pela leitura dos sensores e envio desses dados através da rede CAN.

Os sinais elétricos dos sonares possuem o diagrama temporal apresentado na Figura 28, onde é visível o sinal de controlo a gerar (*trigger pulse*) e o sinal que indicará as distâncias medidas (*echo pulse).*

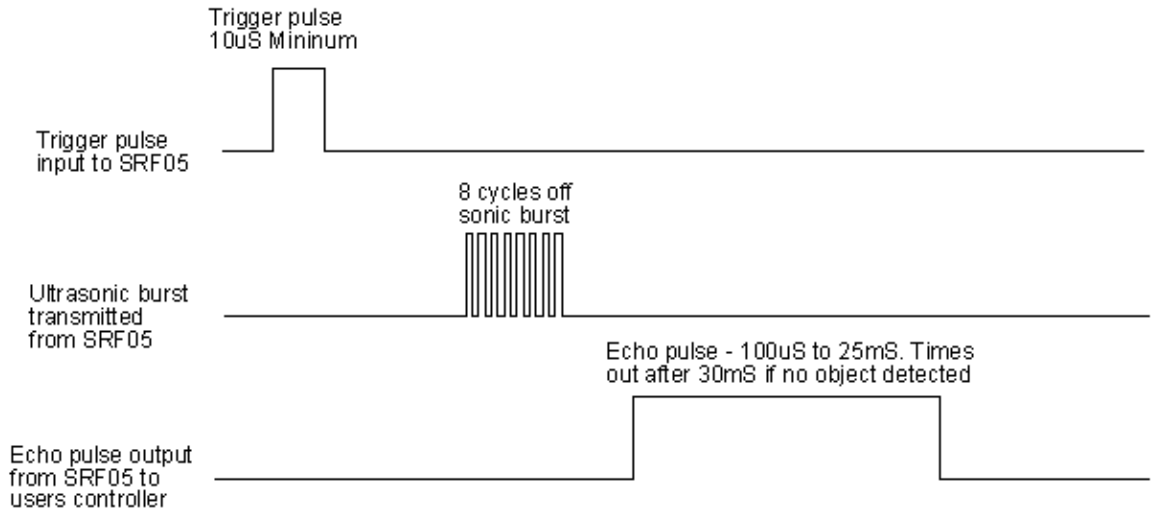


Figura 28 Sinais para interação com o sonar SRF05 [6]

Para o sonar efetuar uma medição é necessário fornecer um sinal de disparo com uma largura mínima de 10 µs. Depois de o sonar emitir a sequência de pulsos, deve-se esperar que a linha de eco vá ao nível lógico "1". O tempo que este sinal permanecer em "1" vai ser proporcional a uma distância; neste caso, a relação distância = tempo (µs) /58 indica diretamente a distância em centímetros.

A implementação da aquisição dos valores das leituras dos quatro sonares passa por uma reprodução dos passos apresentados no fluxograma da Figura 29.

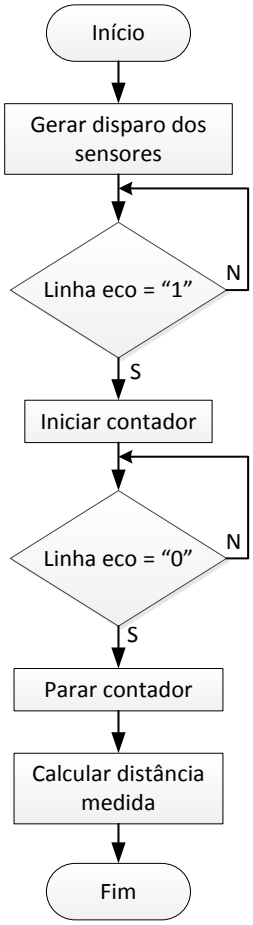


Figura 29 Fluxograma representativo da aquisição dos dados de um sonar

Como existem quatro sensores, decidiu-se dispará-los dois a dois. Os sensores disparados simultaneamente encontram-se em locais opostos para não existirem interferências entre eles. Deste modo, pode-se diminuir o tempo de aquisição dos valores em comparação à situação em que se disparavam os sensores individualmente. Nessa situação, cada aquisição poderia ultrapassar a duração de 30 ms; com esta solução o tempo é reduzido sensivelmente para metade.

A deteção do momento em que a linha de eco é colocada a “0” é efetuada através da verificação das *flags* das interrupções externas que ocorrem no flanco descendente do eco para dois dos sonares e para dois dos outros sonares é através das *flags* do modo de captura dos temporizadores configurando o seu funcionamento para o flanco descendente.

3.6. MEDIÇÃO DE VELOCIDADE E CONTROLADOR PI

Um dos microcontroladores que constituem o AGV tem como objetivo implementar um controlador PI para cada um dos motores que acionam as rodas do sistema móvel. Para isso é necessário implementar um sistema de medição da velocidade a que os motores estão atualmente a rodar. Isto é proporcionado pelo *encoder* interno que cada um dos motores tem e que fornece 3 impulsos por rotação. Os motores utilizados têm também uma caixa redutora com a relação de 94,73:1 entre o veio interno e o veio externo a que estão conectadas as rodas. Assim para cada rotação de uma roda do AGV o *encoder* fornece 283 impulsos. Logo através destes impulsos consegue-se medir a velocidade a que as rodas do AGV estão a rodar efetivamente.

O método escolhido para efetuar a medição da velocidade foi o Método de Contagem de Tempo [1], no qual a frequência de contagem do contador é de 1,25 MHz. A velocidade instantânea do motor é calculada através da divisão do ângulo de rotação do veio externo do motor pelo valor contado entre impulsos, multiplicado pelo período de contagem, tal como apresentado na equação (23).

$$w = \frac{2\frac{\pi}{3} \times \frac{1}{94{,}37}}{n_{contador\ impulsos} \times \left(\frac{1}{1{,}25} \times 10^{-6}\right)} = \frac{0{,}0222}{n_{contador\ impulsos} \times 0{,}0000008}\ (rad/s) \tag{23}$$

A cada sinal gerado pelo *encoder* é gerada uma interrupção que efetua a leitura do valor contado e reinicia o contador, sendo posteriormente calculado o valor da velocidade. Um fluxograma explicativo é apresentado na Figura 30.

A equação de controlo que foi implementada no microcontrolador é apresentada na equação (24) em que K=0,0472, α=0,2869 e $K.\alpha$=0,01235.

$$c\ (k) = Ke(k) - K\alpha e(k-1) + \ c(k-1) \tag{24}$$

Sendo $x_1(k) = c(k) - K\alpha e(k)$ a equação resultante simplificada é apresentada na equação (25).

$$c\ (k) = Ke(k) + x_1(k-1) \tag{25}$$

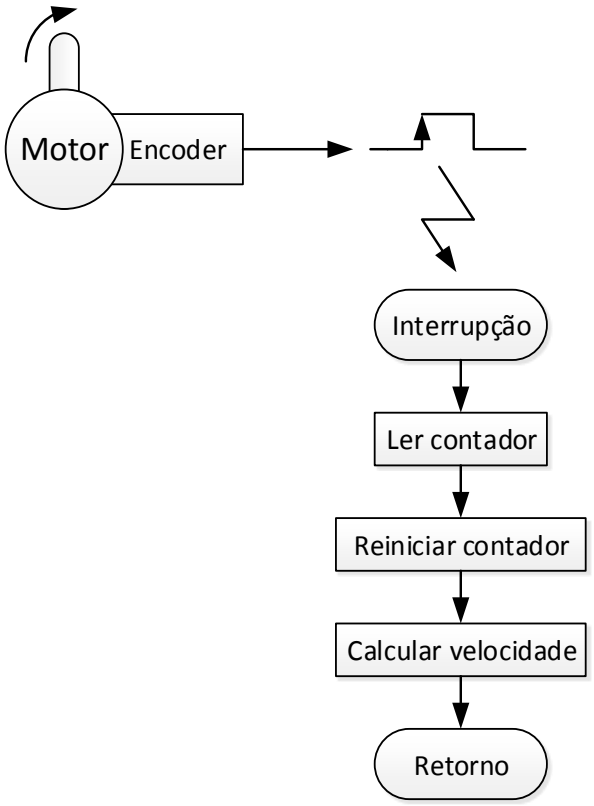


Figura 30 Fluxograma do processo para cálculo da velocidade

Os sinais de controlo são fornecidos através de dois módulos *Pulse-Width Modulation* (PWM) para cada um dos motores que são limitados ao intervalo [0,1] com uma resolução de 10 bits cada, ou seja, consegue-se um controlo com uma variação mínima de 0,978 mV. O controlo dos motores é efetuado por intermédio de uma interrupção que ocorre em intervalos de 10 ms e que implementa a equação (24).

3.7. CONTROLO CENTRAL

Por fim, um dos microcontroladores tem o propósito de calcular a velocidade linear e a velocidade angular do sistema global, isto é do AGV em si, a partir das distâncias medidas através dos sonares. Posteriormente, através das fórmulas de um sistema de tração diferencial, é efetuado o cálculo da velocidade angular de referência para cada um dos motores. Sendo a velocidade angular do sistema dada por:

$$w_{sistema} = V_{direita} - V_{esquerda} \tag{26}$$

a diferença entre as velocidades lineares dos dois motores é:

$$\Delta V = w_{sistema} L \tag{27}$$

em que L é a distância entre os eixos das rodas.

Uma vez que a velocidade linear pretendida já é conhecida, pois resulta da saída do controlador, considera-se que as velocidades a aplicar a cada motor são:

$$V_{direita} = V_{sistema} - \frac{\Delta V}{2} \tag{28}$$

$$V_{esquerda} = V_{sistema} + \frac{\Delta V}{2} \tag{29}$$

Após a obtenção das velocidades lineares a aplicar a cada motor, apenas é necessário dividir o valor obtido pelo raio da roda, de modo a obter-se uma velocidade angular (em rad/s) de referência para os motores.

Em seguida as referências de controlo são enviadas para o microcontrolador que efetua o controlo das velocidades dos motores, o qual as usa como variáveis de entrada do sistema, ou referências de controlo.

3.8. COMUNICAÇÃO BLUETOOTH PARA CONTROLO REMOTO DO AGV

De forma a se obter um controlo remoto do sistema desenvolvido o AGV é dotado de um módulo Bluetooth que se interliga com o sistema através de uma porta série. No sistema remoto que controla o AGV e também o monitoriza é emulada uma porta série sobre o Bluetooth permitindo assim um certo nível de abstração das caraterísticas e funcionamento da rede Bluetooth através da utilização deste tipo de módulo. Para a utilização deste módulo apenas é necessário efetuar a configuração de alguns parâmetros, tais como, a configuração da taxa de comunicação da porta série entre o microcontrolador e este módulo, a definição do papel do módulo e a definição do endereço físico do módulo.

3.8.1. MÓDULO BLUETOOTH HPS-110 [7]

O módulo de porta série sem fios, HPS-110 (Figura 31), é um dispositivo que permite estabelecer uma comunicação *wireless* entre duas portas série externas. O adaptador é vendido aos pares permitindo uma rápida integração e desenvolvimento - na prática vai permitir emular a existência de um cabo série "virtual".

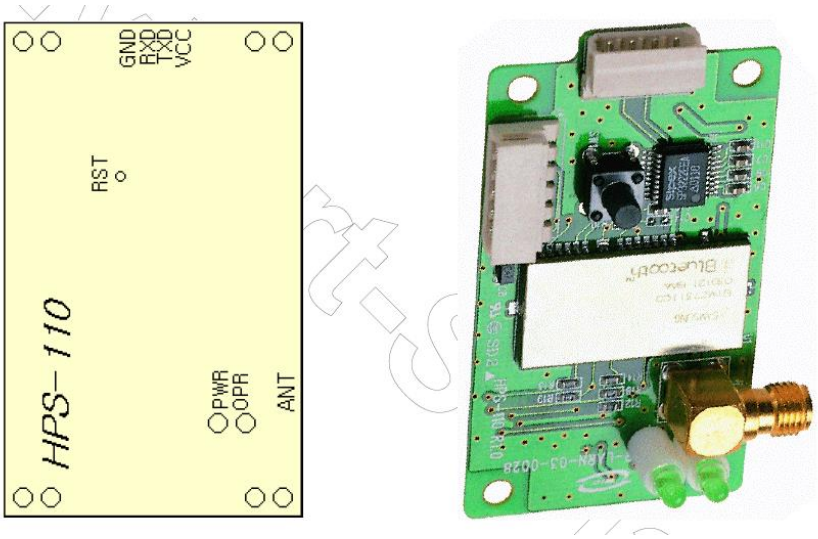


Figura 31 Módulo HPS-110 da HandyWave

3.8.2. ESPECIFICAÇÕES DO MÓDULO BLUETOOTH

O módulo Bluetooth HPS-110 da HandyWave permite comunicação *Full-Duplex*, apresenta taxas de comunicação entre 1,2 e 115,2 kbps e possui um consumo energético de 110 mA no máximo de corrente consumida [7]. Na Tabela 8 é apresentado uma lista de algumas das especificações do módulo Bluetooth.

Tabela 8 Especificações do módulo HPS-110

Especificações	
Standard	Bluetooth v1.1
Frequência rádio	2,402 ~ 2,480 GHz
Largura de banda	1 Mbps
Potência de saída	Tipicamente de 16 dBm
Sensibilidade de receção	-84 dBm
Saltos de frequência	1600/s, cada canal tem 1 MHz de espaço
Velocidades de transmissão	1,2 ~ 115,2 kbps

3.8.3. INTERFACE DE COMUNICAÇÃO DO MÓDULO BLUETOOTH

Este módulo possui quatro pinos, como é visível na Figura 31, dois deles de alimentação (5 V) e os restantes dois pinos (Rx e Tx) para a ligação entre as portas série. Na Tabela 9 é apresentada uma descrição dos pinos do módulo Bluetooth HPS-110.

Tabela 9 Tabela descritiva dos pinos existentes no módulo HPS-110

Número do pino	Notação	Direção	Descrição
1	VCC	Entrada	Vcc 3,3 V ~ 16 V
2	TxD	Saída	Informação transmitida RS-232
3	RxD	Entrada	Informação recebida RS-232
4	GND	N/A	Sinal terra

Se se ligar este módulo a um computador e se utilizar um programa de comunicação série (por exemplo, o Tera Term) pode-se efetuar a configuração dos parâmetros do módulo. Na Figura 32 é apresentado o menu de configuração do módulo. Como este possui ainda um considerável número de parâmetros, estes não são abordados neste documento, mas esta informação está descrita com maior detalhe no manual técnico do módulo em questão [7].

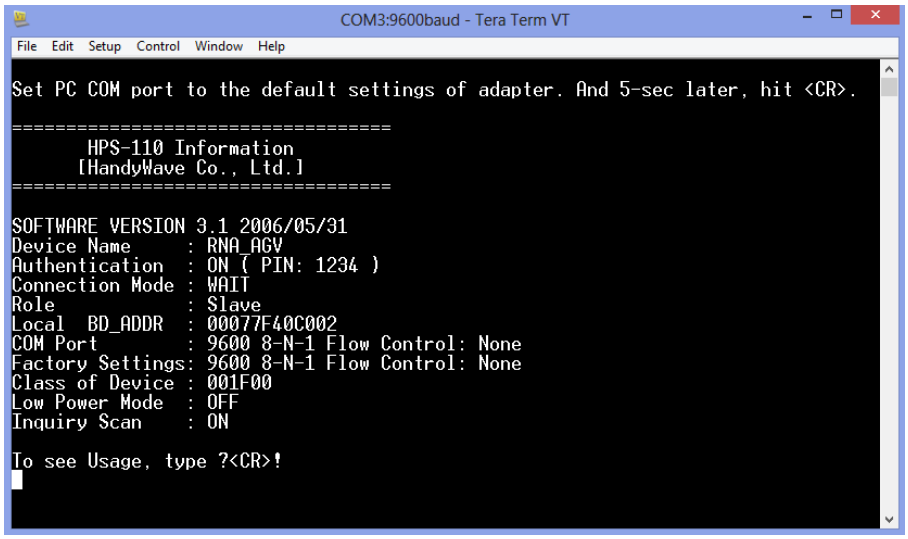


Figura 32 Menu apresentado ao se efetuar a configuração do módulo

3.8.4. APLICAÇÃO DE COMUNICAÇÃO DESENVOLVIDA EM ANDROID

Inicialmente no projeto utilizou-se o programa de comunicação Tera Term num computador para efetuar a monitorização e controlo. Na segunda fase do projeto, em que se treina o robô através de entradas e saídas obtidas através do controlo remoto implementado no robô, foi desenvolvida uma aplicação JAVA para o sistema operativo Android para efetuar o controlo remoto e receber os dados (pares de valores dos sensores e velocidades angulares de cada um dos motores) que são utilizados para efetuar o treino da rede neuronal no MATLAB.

A razão de se ter criado uma aplicação para o *smartphone* Android foi pela sua difusão ser grande atualmente, estava disponível um *smartphone* com o sistema operativo Android que

poderia ser utilizado como interface de controlo e monitorização com melhor mobilidade do que um computador portátil e esta ser uma área que atualmente está em grande expansão e evolução.

O *Integrated Development Environment* (IDE) utilizado para desenvolver a aplicação foi o Eclipse [8]. Este é um IDE multilinguístico que permite o desenvolvimento de projetos para Android através do Android SDK (*Software Development Kit*) [9].

Dado que hoje em dia a maioria dos telemóveis já possuem módulo Bluetooth, optou-se por continuar a utilizar a comunicação Bluetooth entre o sistema e o Android, tal como era utilizado com a comunicação entre o sistema e o computador.

Para a aplicação suportar estes requisitos é necessário recorrer à *Application Programming Interface* (API) de Bluetooth do Android. Devido à falta de conhecimento sobre o sistema operativo Android e Bluetooth recorreu-se à aplicação designada de BluetoothChat, disponibilizada como exemplo no IDE. A partir desta, e do estudo da API, conseguiu-se realizar os requisitos desejados. A aplicação que foi utilizada de base possibilita a comunicação entre dois dispositivos através de um *chat* de texto, apresentado na Figura 33, onde se podem ver os vários *layouts* e funcionalidades disponíveis na aplicação.

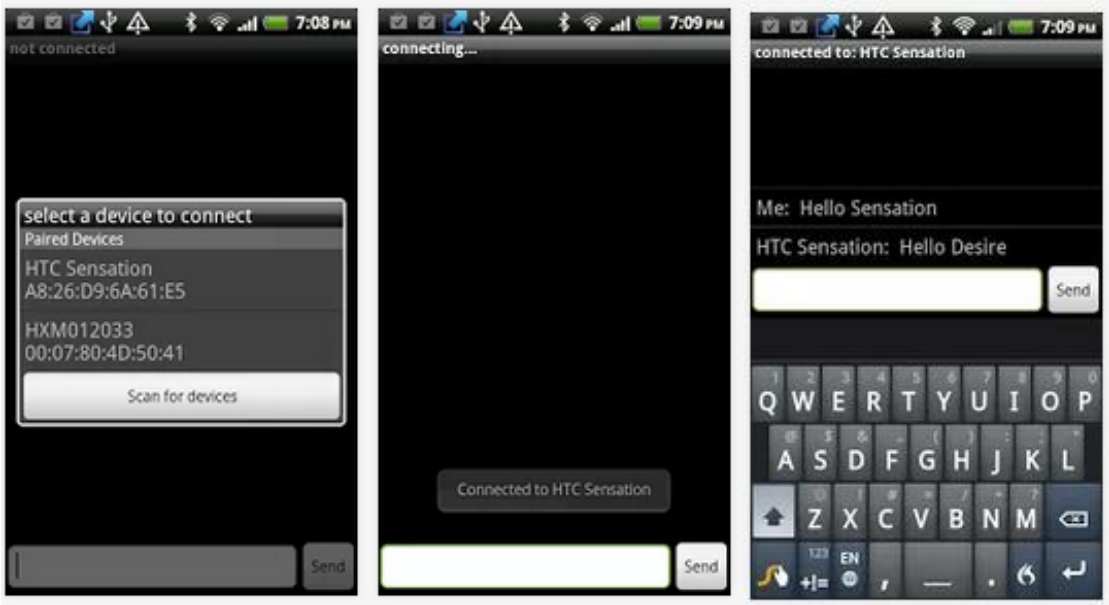


Figura 33 Imagens da aplicação base utilizada para desenvolver a aplicação Android pretendida

Assim, tendo como base esta aplicação foi criada a aplicação que será em seguida apresentada através das suas interfaces com o utilizador e das suas principais funcionalidades.

No ecrã inicial da aplicação gráfica, apresentado na Figura 34a), destacam-se os seguintes elementos:

- A vermelho, a área da interface gráfica onde se apresentam várias informações sobre a execução do sistema, como o modo de funcionamento, o número do último conjunto de amostras recebido via Bluetooth, o tempo de ciclo do sistema e os valores medidos pelos sonares (em cm);

- A verde, a área onde se situam os botões que permitem controlar remotamente o sistema via Bluetooth através dos comandos apresentados na Tabela 10;

- A laranja, as áreas que apresentam a progressão dos processos de amostragem dos valores de entrada e de saída da rede neuronal e de envio dos valores amostrados via Bluetooth do sistema para o telemóvel, situadas à esquerda e direita respetivamente.

- A azul, a área onde é apresentada uma imagem que é alterada mostrando o estado atual de ativação ou desativação das amostras;

- A castanho, a área onde é apresentada uma imagem que mostra a ocorrência de um erro, como a tentativa de modificação do modo do funcionamento do sistema enquanto ainda se estão a receber os dados via Bluetooth.

Na Figura 34b) destaca-se a amarelo o menu de opções. Neste menu encontram-se as seguintes opções:

- Conectar Dispositivo: permite selecionar o dispositivo a que se pretende conectar;

- Modo Autónomo: permite alternar o modo de funcionamento do sistema para o modo em que o controlo é realizado através da rede neuronal artificial implementada. Quando o sistema se encontra no modo Autónomo, esta opção é modificada para Modo Remoto, permitindo assim alternar o controlo do sistema para o modo de controlo quando pretendido. O modo de funcionamento do sistema é inicialmente o modo de controlo remoto;

- Ativar Amostras: permite alternar o modo de operação do sistema para o modo onde são retiradas amostras dos valores de entrada e de saída do sistema de controlo implementado (RNA). Quando o sistema se encontra no modo de operação em que as amostras estão ativadas, esta opção é modificada para Desativar Amostras, permitindo assim alternar o modo de operação do sistema para o modo sem amostras quando pretendido. O modo de operação do sistema é inicialmente o modo de operação sem amostras;

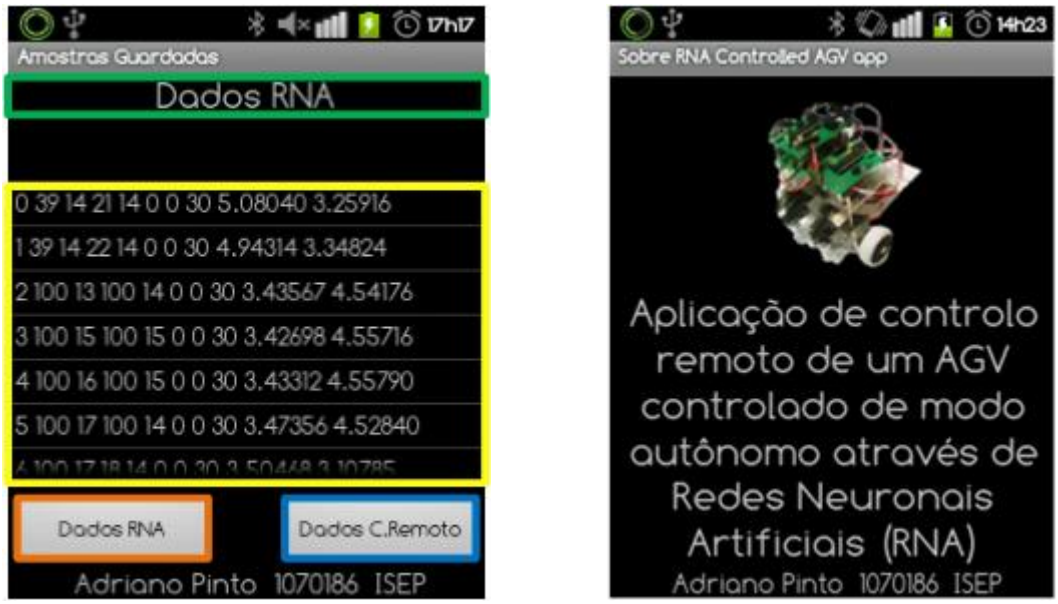
- Apagar Amostras: permite apagar as amostras do sistema que foram guardadas e que ainda não foram enviadas via Bluetooth para o telemóvel;

- Ver Dados: permite aceder ao ecrã apresentado na Figura 35a) onde se verificam os dados amostrados em cada um dos modos de funcionamento;

- Sobre: permite aceder ao ecrã apresentado na Figura 35b) que apresenta informação sobre o objetivo da aplicação desenvolvida.

a) **b)**

Figura 34 Interface gráfica da aplicação Android: a) ecrã inicial e b) menu de opções



a) **b)**

Figura 35 Interface gráfica da aplicação Android: a) ecrã para verificação das amostras retiradas do sistema e b) ecrã com informação sobre a aplicação

Na Figura 35a) ainda se podem destacar os seguintes elementos:

- A verde, a área onde é apresentada a identificação dos dados apresentados nesse momento neste ecrã;
- A amarelo, a área onde são apresentados os dados amostrados do sistema que são lidos de ficheiros guardados no cartão de memória do telemóvel;
- A laranja, o botão que permite selecionar os dados amostrados do sistema no modo de funcionamento com controlo através de uma rede neuronal artificial;
- A azul, o botão que permite selecionar os dados amostrados do sistema no modo de funcionamento com controlo remoto.

O controlo remoto do sistema consiste no envio de um comando (um carater numérico entre 1 e 9) via Bluetooth que é posteriormente descodificado no microcontrolador e que origina os seguintes pares de comando/resultado do sistema apresentados na Tabela 10.

Tabela 10 Tabela descritiva das ações para cada tecla no *smartphone*

Tecla	Ação	Motor Esquerda (rad/s)	Motor Direita (rad/s)
↖	Diagonal Esquerda Frente	4,0	5,0
↑	Frente	5,0	5,0
↗	Diagonal Direita Frente	5,0	4,0
←	Esquerda	0,0	4,0
STOP	Parar	0,0	0,0
→	Direita	4,0	0,0
↙	Diagonal Esquerda Trás	-4,0	-5,0
↓	Trás	-5,0	-5,0
↘	Diagonal Direita Trás	-5,0	-4,0

3.8.5. IMPLEMENTAÇÃO DA APLICAÇÃO

Nesta subsecção é descrito como é implementada a aplicação do Android e as suas funções chave [10][11][12].

A comunicação é implementada recorrendo à API de Bluetooth do Android, como já referido. A aplicação não atende pedidos de ligação, desta forma, não é implementada a classe para ser servidor. As classes implementadas são:

- `BluetoothAdapter` – representa o módulo de Bluetooth do dispositivo Android no qual a aplicação está a correr;
- `BluetoothDevice` – representa cada dispositivo remoto com o qual se deseja comunicar;

- `BluetoothSocket` – cria uma *socket* de Bluetooth através do método `createRFcommSocketToServiceRecord`; a *socket* permite realizar o pedido de ligação ao dispositivo remoto, e realizar a comunicação.

Para realizar a conexão e comunicação foram criadas duas *threads*, designadas de `ConnectThread` e `ConnectedThread`. Estas têm as funções de:

- `ConnectThread` – esta *thread* corre enquanto se está a estabelecer a ligação com um dispositivo;

- `ConnectedThread` – esta *thread* corre durante a ligação e gere todas as transmissões.

Para gerir estas *threads* foram criados os métodos:

- `Connect` – este método cancela qualquer *thread* que esteja a tentar fazer uma ligação (`ConnectThread`) e cancela qualquer *thread* que esteja a gerir uma ligação efetuada (`ConnectedThread`). Após estes passos inicia a `ConnectThread` com o dispositivo indicado;

- `Connected` – este método cancela a *thread* que completou a ligação e inicia a *thread* `ConnectedThread` que gere as transmissões da nova ligação.

Quando a aplicação é inicializada obtém-se o `BluetoothAdapter`, o qual é utilizado para verificar se o dispositivo móvel possui módulo Bluetooth; caso não tenha, a aplicação é terminada. O fluxograma da Figura 36 representa o processo referido.

Quando o utilizador seleciona o dispositivo a que se irá conectar, o objeto `BluetoothAdapter` é utilizado para obter o `BluetoothDevice` que representa o dispositivo selecionado. Este objeto é utilizado para realizar o pedido de conexão ao dispositivo. Após a seleção do dispositivo, o `connect` é executado. O método `connect` tem as funcionalidades já indicadas, sendo estas:

- Cancela qualquer *thread* que esteja a tentar realizar uma conexão, ou seja, cancela qualquer `ConnectThread`;

- Cancela qualquer *thread* que esteja com uma ligação ativa, ou seja, cancela qualquer `ConnectedThread`;

- Por fim inicia uma nova *thread* que se liga ao dispositivo indicado, designada de `ConnectThread`.

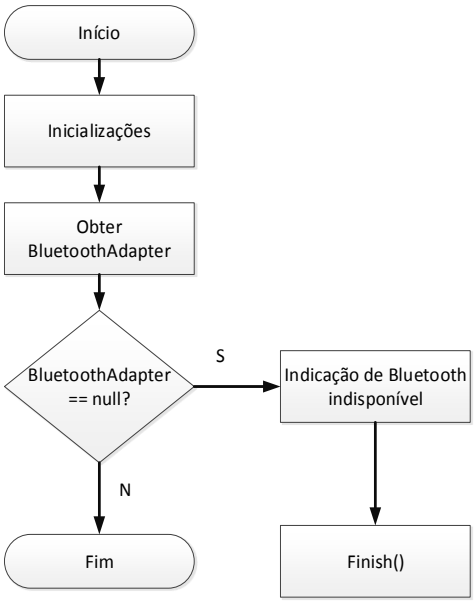


Figura 36 Fluxograma da inicialização da aplicação

A `ConnectThread` executa o método `createRfcommSocketToServiceRecord` que cria a `BluetoothSocket` para a comunicação com o `BluetoothDevice` indicado. Este passo é realizado utilizando um determinado *Universally Unique Identifier* (UUID). O UUID é um identificador único que permite identificar o serviço de Bluetooth da aplicação, sendo assim diferente para as várias aplicações. Esta regra não se aplica para aplicações que comuniquem com módulos Bluetooth, como é o caso deste projeto. O UUID para estes fins é sempre o mesmo:

00001101-0000-1000-8000-00805F9B34FB

Após se obter a *socket* é executada a conexão, sendo esta bloqueante, ou seja, a *thread* fica bloqueada até a conexão ser realizada. Quando isso acontece o programa desbloqueia e inicia o método `connected`. O método `connected` tem as funcionalidades já indicadas, sendo estas:

- Cancela a `ConnectThread` que efetuou a ligação;
- Inicia a `ConnectedThread` que gere a ligação, ou seja, recebe e envia os dados.

Estes processos são representados pelo fluxograma da Figura 37.

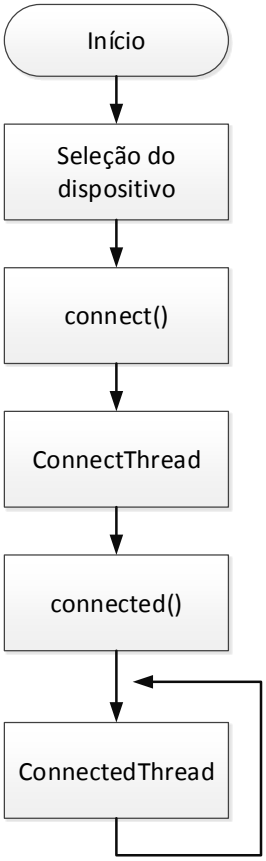


Figura 37 Fluxo de conexão ao dispositivo remoto

Ao iniciar a `ConnectedThread` é passada para esta *thread* a *socket* para permitir a troca de dados, sendo esta *thread* que corre durante toda a ligação e gere os fluxos de dados. Para realizar a troca de dados obtém-se numa fase inicial da `ConnectedThread` os objetos de entrada e saída de dados através da *socket*, como representado no seguinte excerto de código.

```
InputStream tmpIn = null;
OutputStream tmpOut = null;
// Get the BluetoothSocket input and output
streams
try {
     tmpIn = socket.getInputStream();
     tmpOut = socket.getOutputStream();
    }
catch (IOException e) {
       Log.e(TAG, "temp sockets not created", e);
     }
mmInStream = tmpIn;
```

```
mmOutStream = tmpOut;
```

A `ConnectedThread` recebe e envia os fluxos de dados em forma de *bytes*. O método de leitura recebe um *byte* de cada vez que é armazenado num *buffer*. No seguinte excerto de código está apresentado o código da função de leitura que utiliza este procedimento. A função de leitura termina de adicionar os dados no *buffer* quando é recebido o carater '\r', sendo posteriormente estes dados enviados para a *activity*[2] de interface com o utilizador.

```
int inputdata;
bytes[0] = 0;
while((inputdata = mmInStream.read()) > -1)
      {
        char inputchar = (char)inputdata;
        buffer[bytes[0]++] = (byte)inputchar;
        if (inputchar == '\r')
           {
              return;
           }
      }
    }
```

O envio de dados é efetuado através do método de escrita de dados existentes num *buffer* no *stream* de saída de dados do *socket* do Bluetooth, como é representado no seguinte excerto de código.

```
mmOutStream.write(buffer);
```

Quando a aplicação recebe dados, a `ConnectedThread` envia-os para o `mHandler` da interface gráfica (mostrado no extrato de código apresentado em seguida), que os manipula através do método `handleMessage`. Na receção dos dados enviados pela `ConnectedThread`, este método, que manipula todo o tipo de mensagens recebidas, verifica o tipo de mensagem recebida. Se o tipo de mensagem recebida for MESSAGE READ, ou seja, mensagem de leitura, o método verifica as marcas que foram definidas neste tipo de mensagens que permitem à aplicação identificar a funcionalidade despoletada pela receção desta mensagem, entre as quais se encontram a gravação em memória, num ficheiro, dos dados recebidos ou a apresentação dos dados numa das `TextView` existentes na interface com o utilizador, entre outras ações possíveis.

```
private final Handler mHandler = new Handler() {
   (…)
  public void handleMessage(Message msg) {
     (…)
```

[2] Uma *activity* é um componente de uma aplicação que disponibiliza um ecrã com que o utilizador pode interagir em ordem a efetuar alguma ação. A cada *activity* é dada uma janela em que é permitido desenhar a sua interface com o utilizador.

```
    switch (msg.what) {
        (…)
       //Verificação do tipo de mensagem recebida
       case MESSAGE_READ:
         //leitura da mensagem para um buffer
         local
         byte[] readBuf = (byte[]) msg.obj;
         // construct a string from the valid
        bytes in the buffer
         String readMessage = new String(readBuf,
         0, msg.arg1);

/*Verificação do conteúdo da mensagem recebida e
dependendo deste apresentação dos dados ou
escrita dos dados num ficheiro ou outra ação*/
}}};
```

Na Tabela 11 apresentam-se as marcas adicionadas nas mensagens transmitidas via Bluetooth, que permitem à aplicação decidir o que deve efetuar com os dados recebidos.

Tabela 11 Marcas implementadas nas mensagens recebidas via Bluetooth

Mensagem	Marcas
Apresentação dos valores dos sensores	"S"
Identificação do conjunto de dados recebidos	"F"
Tempo de ciclo do sistema	"T"
Amostragem em curso	"*"
O sistema fez *reset*	"#"

No caso de uma mensagem recebida não conter nenhuma destas marcas, a aplicação guarda em memória os dados recebidos, num ficheiro que se encontra na memória externa do telemóvel e que posteriormente pode ser acedido e os seus dados analisados através do MATLAB.

4. PROJETO E SIMULAÇÃO DA RNA

Neste capítulo são abordados todos os passos relevantes efetuados durante o projeto e desenvolvimento das redes neuronais artificiais para o controlo do AGV. Para isso é descrita a tarefa de obtenção de dados do AGV para treinar a rede. Posteriormente é efetuada a descrição do treino de uma rede neuronal através do MATLAB que efetua o seguimento de uma parede e a utilização de um programa em C para simulação e teste da rede criada. Também é relatado o treino de redes neuronais que efetuam comportamentos mais complexos como o seguimento de uma parede com descontinuidades ou o desviar-se de obstáculos.

4.1. ROTINA PARA GUARDAR E ENVIAR OS DADOS PARA UTILIZAR NO TREINO DA RNA

Como foi referido no Capítulo 2, uma das vantagens da implementação das redes neuronais é a sua capacidade de aprendizagem. Assim tira-se vantagem dessa característica para implementar uma rede neuronal que tem como objetivo possuir o mesmo comportamento que o controlador difuso desenvolvido no trabalho de tese anterior para controlo do AGV [1]. Para isso é necessário obterem-se dados de entrada e as respetivas saídas desejadas que

a rede neuronal terá de apresentar para se efetuar o treino da rede neuronal e assim esta aprender a funcionar da forma pretendida.

Com este propósito em mente, foi alterado o código fonte do controlador difuso, adicionando-se a funcionalidade de durante a sua execução serem guardados os valores de entrada e de saída do controlador. A rotina implementada para guardar e posteriormente enviar os dados é apresentada no fluxograma da Figura 38.

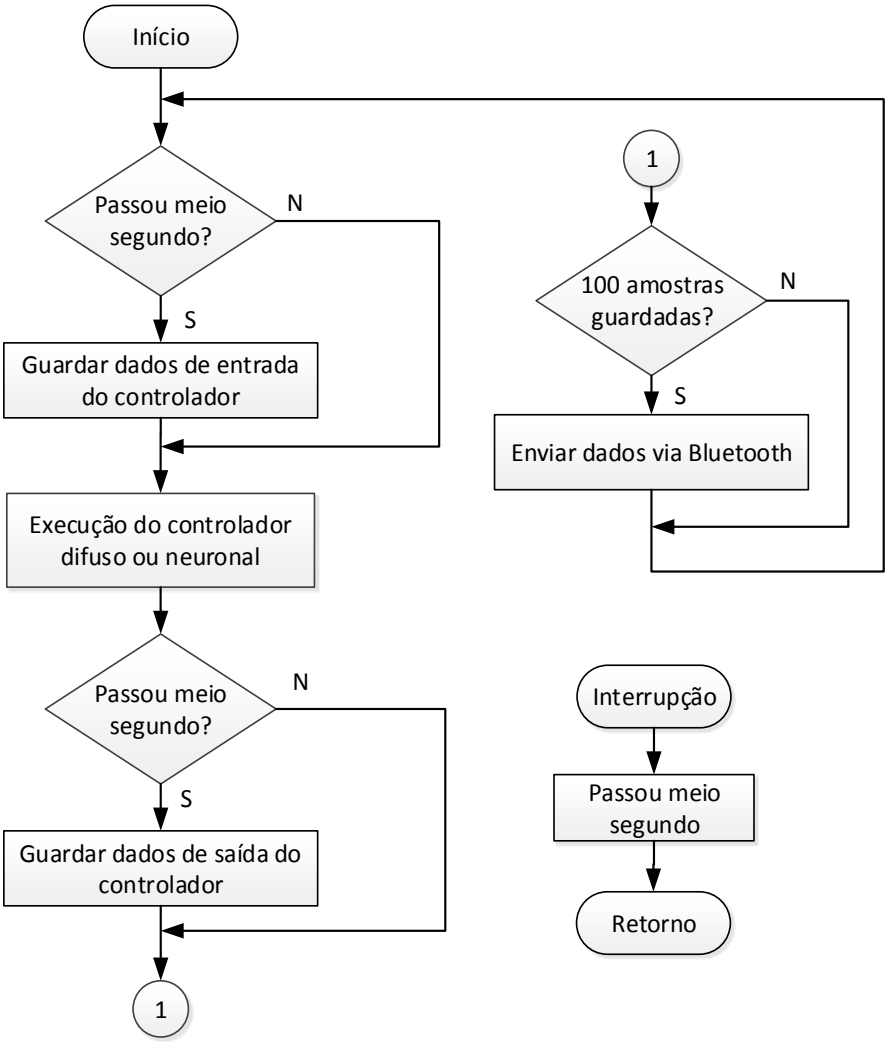


Figura 38 Rotina para guardar e enviar os dados via Bluetooth

Os dados são guardados em 9 vetores, em que cada vetor é relativo a uma das variáveis de entrada ou de saída do controlador difuso, com um tamanho de 100 posições e com uma frequência de amostragem de 2 Hz. Os dados guardados são as medições feitas pelos sensores (*S*1, *S*2, *S*3 e *S*4 em cm), o erro de orientação (*EO* - diferença entre a medição do sensor *S*4 e o sensor *S*2), o erro de distância (*ED* - diferença entre o valor de referência da distância entre o AGV e a parede e a medição do sensor *S*4), o valor de referência da distância

entre o AGV e a parede (*Ref*), a velocidade linear do sistema (*Vfinal*) e a velocidade angular do sistema (*Wfinal*).

Para enviar os dados guardados (a cada 100 amostras) foi utilizado o módulo Bluetooth existente no AGV que comunica com o microcontrolador via USART.

Na Figura 39 é apresentado um extrato dos dados recebidos no PC via Bluetooth simulando uma comunicação série. Como se pode verificar pela figura, cada amostra (linha) é composta por 10 valores (colunas), sendo o primeiro o número da amostra, enquanto as outras colunas são os dados guardados nos vetores durante a execução do programa e pela seguinte ordem: *S*1, *S*2, *S*3, *S*4, *EO*, *ED, Ref, Vfinal e Wfinal*.

```
0 40 15 30 15 0 15 30 0.10429 0.14784
1 40 15 30 15 0 15 30 0.10429 0.14784
2 40 15 30 16 1 14 30 0.10429 0.11725
3 40 15 30 17 2 13 30 0.10429 0.11725
4 40 15 26 19 4 11 30 0.10429 0.06627
5 40 17 28 21 4 9 30 0.10429 -0.00509
6 40 18 30 22 4 8 30 0.10429 -0.03568
7 40 20 30 24 4 6 30 0.10429 -0.05607
8 40 22 30 25 3 5 30 0.10429 -0.10705
9 40 24 30 27 3 3 30 0.10429 -0.10705
```

Figura 39 Exemplo dos dados recebidos via comunicação série sobre Bluetooth

Estes dados foram guardados num ficheiro de texto para posteriormente serem utilizados no treino da rede neuronal.

Como já referido anteriormente, o controlador difuso utilizado como base para o treino da rede neuronal implementada é composto por 3 comportamentos: seguir paredes, desviar de obstáculos e paragem de emergência. O desenvolvimento da RNA foi efetuada por etapas com um grau crescente de complexidade dos comportamentos que o AGV teria de apresentar, sendo estas as seguintes:

- Seguir uma parede sem descontinuidades posicionando-se à distância de referência;
- Efetuar o seguimento de uma parede com descontinuidades;

- Efetuar o seguimento de uma parede com descontinuidades e desviando-se de obstáculos.

4.2. CRIAÇÃO E TREINO DE UMA RNA QUE IMPLEMENTA O COMPORTAMENTO DE SEGUIR A PAREDE

Para a criação da rede neuronal que efetua este comportamento foram utilizados dados recebidos através da rotina do fluxograma da Figura 38 apresentado na secção 4.1.

Os dados de entrada utilizados para a rede neuronal são os valores do erro de orientação e do erro de distância que estão apresentados na Figura 40, onde se verifica a evolução das variáveis de entrada, *EO* e *ED*, ao longo do tempo amostrado.

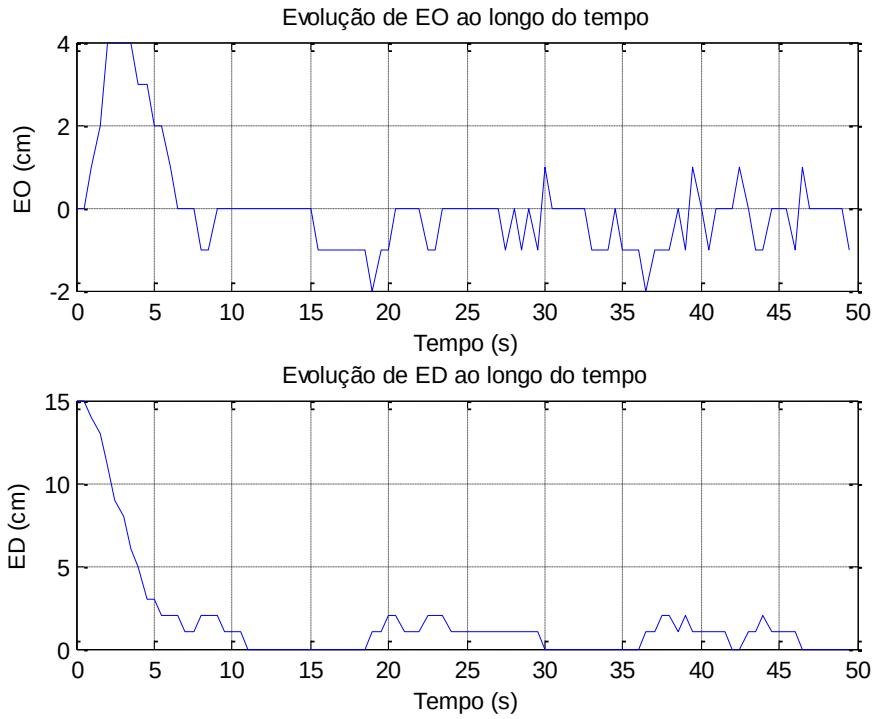


Figura 40 Dados de entrada para o treino da RNA que implementa o comportamento de seguir paredes

Como se pode verificar pela análise dos gráficos, o percurso a que este dados se referem é iniciado com o robô a 15 cm da parede, posteriormente este evolui para se colocar à distância de referência de 30 cm em relação à parede que está a seguir. Na Figura 41 é apresentada uma fotografia do percurso utilizado para obter amostras do controlador difuso, para treinar a rede neuronal de forma a esta se comportar como pretendido.

Figura 41 Percurso amostrado através do controlador difuso para o comportamento de seguir uma parede

A realização deste percurso estabelece os dados de saída, que são a velocidade linear e a velocidade angular, sendo caracterizado por uma velocidade linear constante diferente de zero, e velocidade angular onde existem variações que refletem as correções no trajeto que o AGV tem de efetuar para se colocar à distância de referência em relação à parede que está a seguir. O que foi anteriormente referido pode ser verificado na Figura 42 que apresenta os gráficos da evolução das variáveis de saída do controlador difuso ao longo do tempo amostrado.

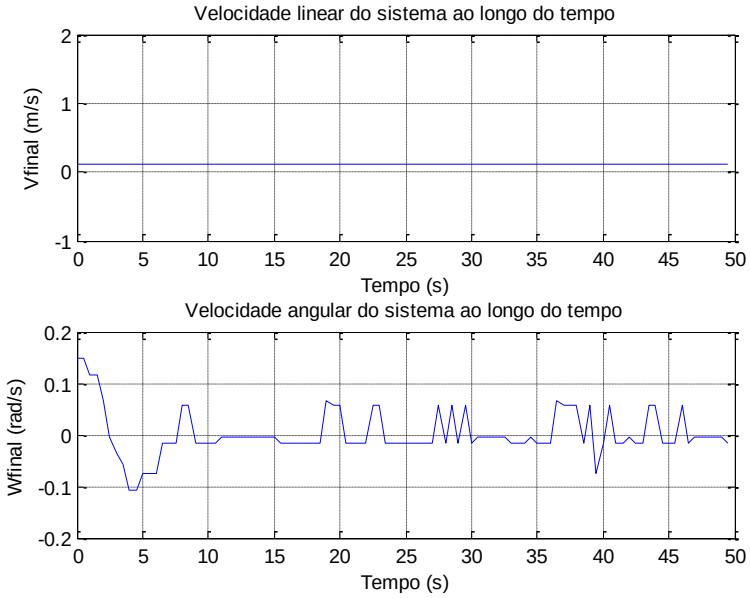


Figura 42 Dados de saída do controlador difuso para o treino da RNA que implementa o comportamento de seguir paredes

O código seguinte mostra os comandos MATLAB utilizados para o efeito.

```
%% Loading training data
data=load('data1.txt');
T=0.5;

%% Defining values to the Network
entradas=[data(:,6) data(:,7)];
saidas_desejadas=[data(:,9) data(:,10)];
inputs=entradas';
targets=saidas_desejadas';

%% Create a Neural Network
rede = newff(minmax(inputs),[10 2],{'logsig'
'purelin'});

%% Initialize Weights
rede=init(rede)
pesosIL = rede.IW{1}
pesosLW = rede.LW{2}
polaridade1 = rede.b{1}
polaridade2 = rede.b{2}

%% Train network
rede.trainParam.showCommandLine = 1;
rede.trainParam.show = 10;
rede.trainParam.lr = 0.01;
rede.trainParam.mc = 0.9;
rede.trainParam.epochs = 1000;
rede.trainParam.goal = 0;
[rede treino] = train(rede,inputs,targets);
treino.perf(end)
pesosIL = rede.IW{1}
pesosLW = rede.LW{2}
polaridade1 = rede.b{1}
polaridade2 = rede.b{2}
pause
nntraintool('close')

%% Simulate the Network
data_sim=load('dados_sim_codigoC.txt');
saidas=sim(rede,inputs);
subplot(211);
plot(data(:,1)'*T,targets(1,:),'b');
grid on;hold on;
plot(data(:,1)'*T,saidas(1,:),'r');
plot(data(:,1)'*T,data_sim(:,3)','g')
xlabel('Tempo(s)');ylabel('rad/s');
title('Velocidade Linear do Sistema');
subplot(212);
plot(data(:,1)'*T,targets(2,:),'b');grid on;
hold on;plot(data(:,1)'*T,saidas(2,:),'r');
plot(data(:,1)'*T,data_sim(:,5)','g')
xlabel('Tempo(s)');ylabel('rad/s');
title('Velocidade Angular do Sistema');
pause
```

```
close all
```

A rotina implementada no MATLAB efetua o carregamento dos dados a serem utilizados para treinar a rede através do comando `load`, que tem como parâmetros de entrada o nome do ficheiro a carregar e que devolve os dados existentes no ficheiro. A partir da variável resultante do carregamento do ficheiro são criadas duas matrizes que serão utilizadas no momento de se efetuar o treino da rede. Uma das matrizes (chamada de *inputs*), correspondente às entradas (*EO* e *ED*) e a outra matriz (chamada de *targets*), corresponde às saídas desejadas (*Vfinal e Wfinal*).

Posteriormente, através do comando `newff` é criada uma rede neuronal do tipo multicamada *feedforward* com 2 camadas. A primeira camada, a camada escondida, é constituída por 10 neurónios com uma função de ativação sigmoide logarítmica. A segunda camada, a camada de saída, é constituída por 2 neurónios, logo duas variáveis de saída, com uma função de ativação linear. Na Figura 43 é apresentado o diagrama desta rede neuronal.

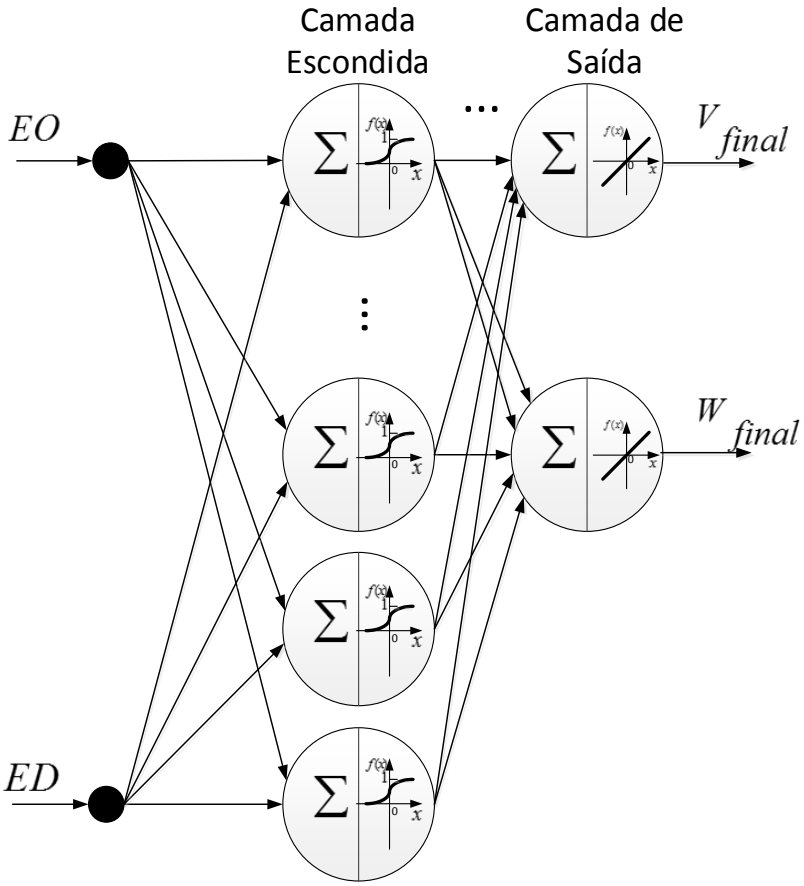


Figura 43 Diagrama da rede neuronal implementada que realiza o comportamento seguir paredes

Após se ter efetuado a criação da rede neuronal é necessário efetuar a definição inicial dos pesos de cada uma das ligações existentes na rede neuronal. Para isso é utilizado o comando `init` que tem como parâmetro de entrada a variável que identifica a rede neuronal anteriormente criada [13]. Nesta variável existem campos que identificam os algoritmos que este comando utilizará para inicializar os pesos e as polarizações. Por predefinição, estes campos configuram a rede neuronal para ser inicializada através do algoritmo Nguyen-Widrow [14]. Este algoritmo escolhe valores a fim de distribuir a região ativa de cada neurónio na camada de uma forma aproximadamente uniforme ao longo do espaço de entrada da camada. Os valores resultantes contêm um grau de aleatoriedade, pelo que eles não são os mesmos cada vez que este comando é efetuado.

Com os passos anteriores realizados pode-se então efetuar o treino da rede neuronal, que sendo efetuado com sucesso permitirá obter uma rede neuronal que se comportará da forma pretendida. Por predefinição, o algoritmo de retropropagação de Levenberg-Marquardt é o algoritmo de treino com que a rede neuronal é configurada com o comando `newff` [15]. Este algoritmo é uma otimização do algoritmo de retropropagação original, convergindo de uma forma mais rápida e eficaz que este. Tem a desvantagem de possuir processamento mais pesado em comparação com o algoritmo de retropropagação simples. Neste caso, sendo o treino efetuado *offline* e através do *software* MATLAB, optou-se pelo algoritmo de retropropagação de Levenberg-Marquardt, pois são obtidos resultados mais satisfatórios que o algoritmo de retropropagação simples como referido em seguida.

Na Figura 44 e Figura 45 são apresentados os gráficos que apresentam a comparação entre as saídas das redes neuronais que pretendem simular o comportamento de seguir uma parede, apenas tendo como diferença entre si o número de neurónios na camada escondida. Foram implementadas redes neuronais com 10, 15 e 25 neurónios para se obter as respostas apresentadas a azul, magenta e vermelho, respetivamente.

Como se pode observar pela análise dos gráficos nenhuma das redes obteve uma resposta semelhante aos valores das variáveis de saída utilizadas no treino, apresentadas a traço preto, considerando-se assim um treino sem sucesso. A consideração que o treino foi completado sem sucesso também é apoiada por não se ter atingido nenhuma das condições de fim de treino com sucesso, pois este foi completado por se ter atingido o número máximo de iterações (1000) permitido.

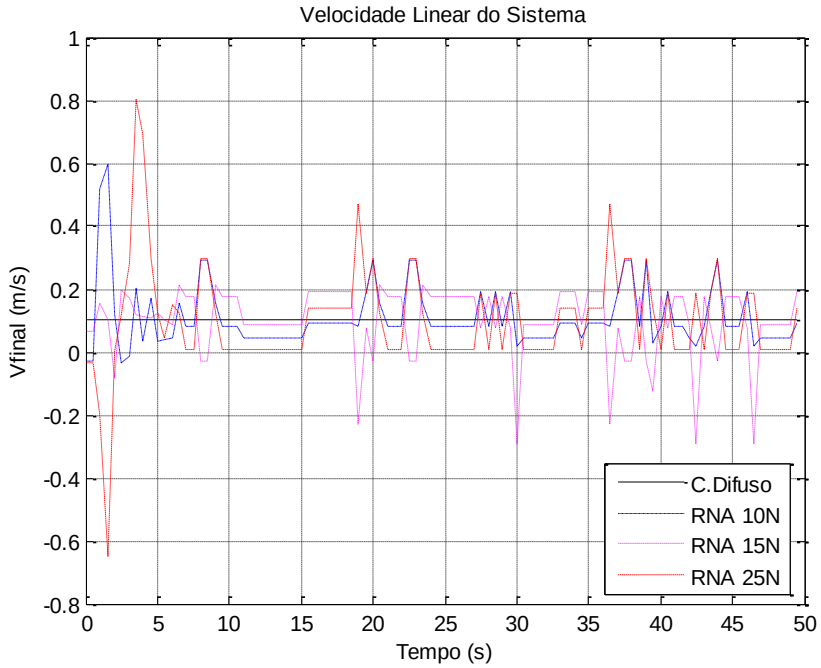


Figura 44 Comparação dos resultados do treino de uma rede neuronal com o algoritmo retropropagação variando o número de neurónios na camada escondida: velocidade linear

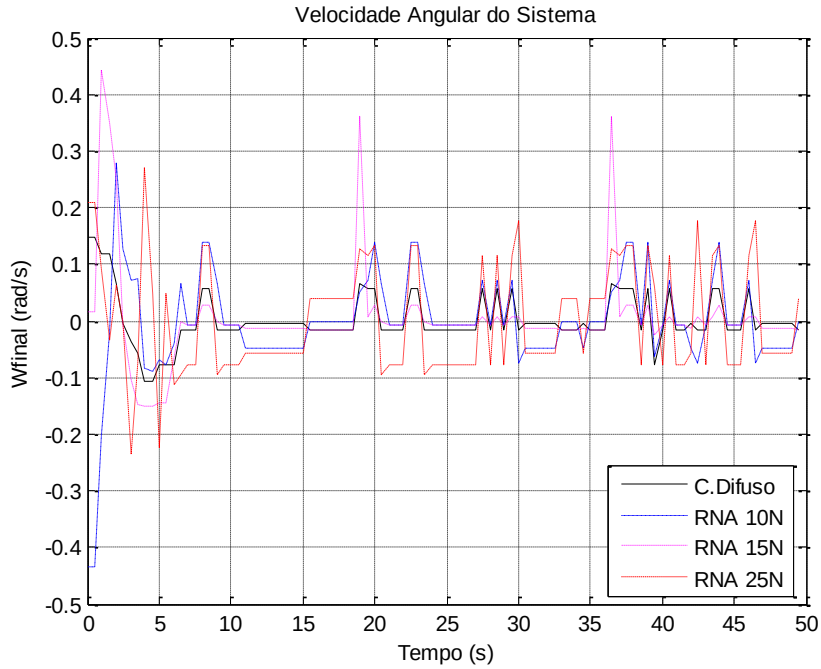


Figura 45 Comparação dos resultados do treino de uma rede neuronal com o algoritmo retropropagação variando o número de neurónios na camada escondida: velocidade angular

Na Figura 46 e Figura 47 são apresentadas comparações da utilização do algoritmo de retropropagação para o treino de uma rede neuronal para efetuar o comportamento desejado

variando o fator de aprendizagem. Os valores de saída apresentados a azul, magenta e vermelho referem-se às respostas das redes neuronais treinadas com 0,01, 0,05 e 0,1 como fator de aprendizagem, respetivamente.

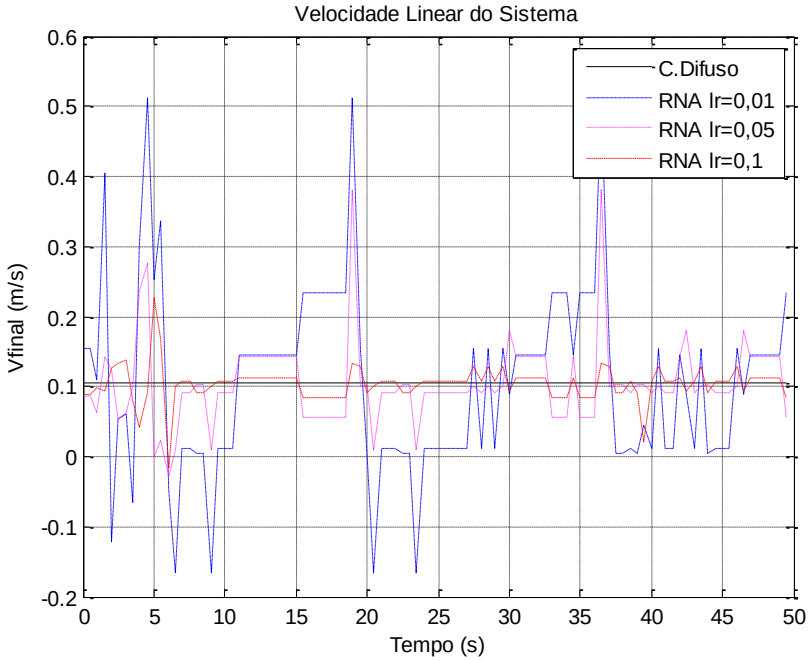


Figura 46 Comparação dos resultados do treino de uma rede neuronal com o algoritmo retropropagação variando o fator de aprendizagem: velocidade linear

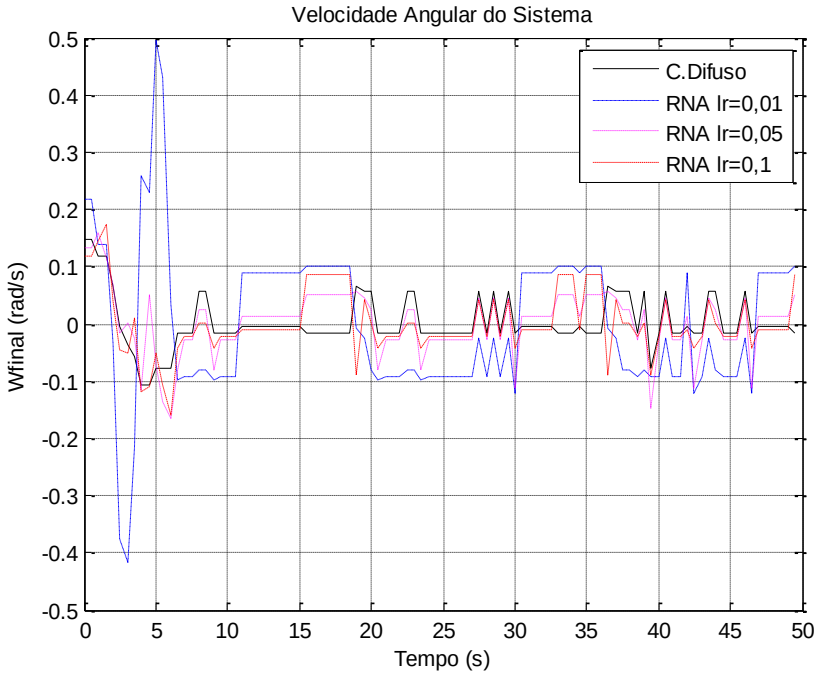


Figura 47 Comparação dos resultados do treino de uma rede neuronal com o algoritmo retropropagação variando o fator de aprendizagem: velocidade angular

Como se pode verificar pela análise dos gráficos da Figura 46 e Figura 47, as respostas obtidas pela simulação das redes neuronais treinadas não se assemelham à resposta utilizada como treino da rede, apresentada a traço preto.

Através dos testes efetuados e apresentados anteriormente, pode-se verificar que com o algoritmo retropropagação, ou algumas das suas outras variações, como o algoritmo de retropropagação com momento, não se conseguiriam obter os resultados pretendidos. Assim, optou-se por utilizar o algoritmo de Levenberg-Marquardt, por se terem obtidos os resultados desejados, como será apresentado a seguir.

Para efetuar o treino da rede é apenas necessário utilizar o comando `train` que tem como parâmetros de entrada a variável de identificação da rede e as matrizes que contêm as entradas e as saídas para efetuar o treino. Ao ser executado o treino é apresentada a interface gráfica (Figura 48) que permite analisar o progresso do treino durante a sua execução.

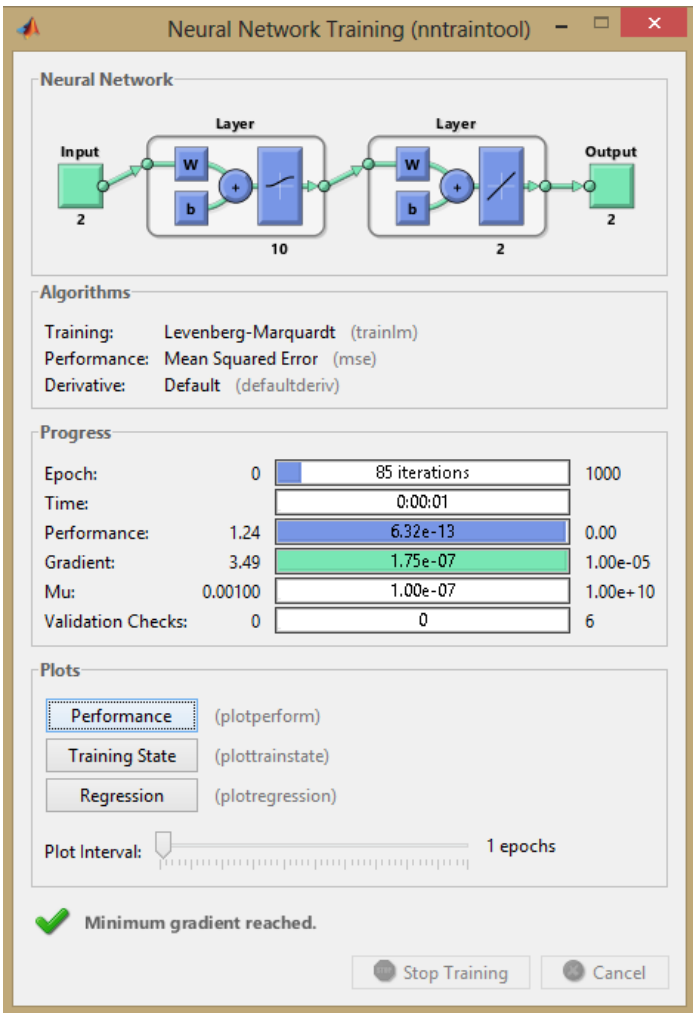


Figura 48 Interface gráfica apresentada durante a execução do treino

Como se pode verificar na Figura 48, o treino foi efetuado com sucesso ao fim de 85 iterações. O sucesso foi verificado por se ter atingido o gradiente mínimo aceitável, ou seja, a variação dos pesos entre iterações é pequena, podendo assim concluir-se que o treino convergiu para o valor mínimo de erro aceitável, logo foi atingido um desempenho satisfatório.

Para verificar se a rede neuronal foi treinada com sucesso utiliza-se o comando `sim` que permite simular a rede neuronal. Este comando tem como parâmetros de entrada a variável identificadora da rede neuronal a simular e a matriz que contém os dados de entrada para simular a resposta da rede neuronal a essas mesmas entradas. Utilizando como entradas para a simulação da rede neuronal os dados de entrada utilizados para treinar a rede obtiveram-se os gráficos apresentados na Figura 49. O traço azul corresponde aos resultados experimentais do controlador difuso e o magenta corresponde aos resultados da simulação da rede neuronal.

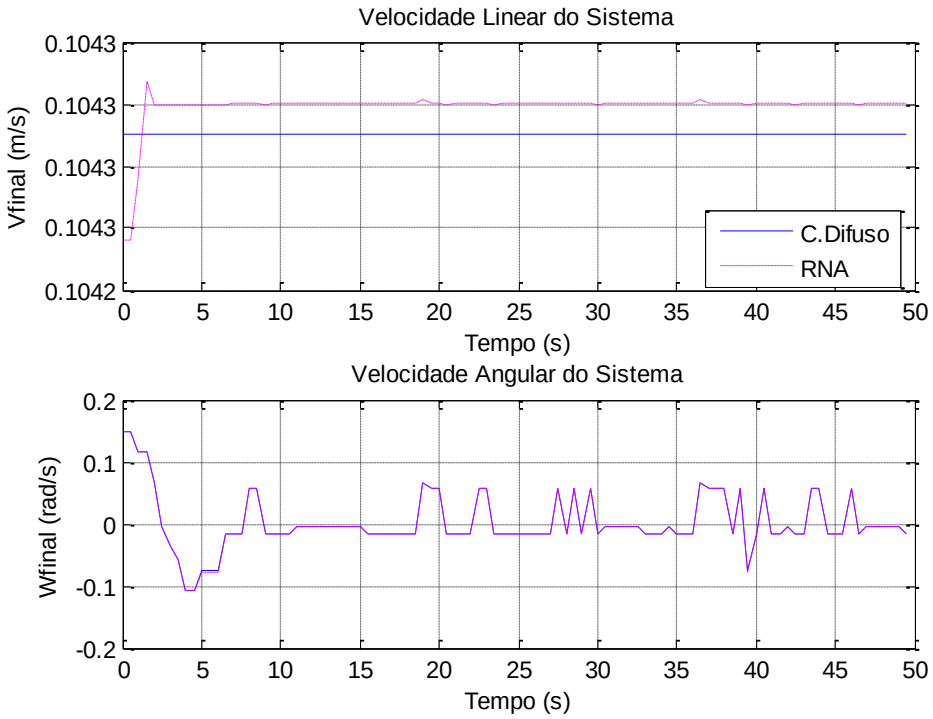


Figura 49 Resposta apresentada pela simulação da rede neuronal utilizando os dados de entrada do treino no MATLAB

Como se pode verificar pela análise dos gráficos apresentados na Figura 49, a rede neuronal apresenta uma resposta perfeitamente semelhante ao controlador difuso, notando-se apenas mínimas diferenças na velocidade linear do sistema. Esta discrepância é originada pela quase

inexistência de variação dos valores amostrados dessa variável quando foi efetuado o treino, logo durante a execução deste os pesos do neurónio relativos a esta variável de saída não foram atualizados de forma tão eficaz como o que se verificou para o neurónio da variável da velocidade angular.

Com os resultados obtidos considerou-se que tinha sido criada uma rede neuronal que satisfazia o comportamento pretendido, passando-se assim ao passo seguinte - a sua implementação em linguagem C.

4.3. IMPLEMENTAÇÃO DE UMA REDE NEURONAL EM LINGUAGEM C

A implementação em linguagem C de uma rede neuronal teve como objetivo obter uma plataforma para facilitar a implementação e posteriores testes de funcionamento das várias redes neuronais que foram desenvolvidas através do uso de ficheiros para a interface de comunicação de dados entre o programa em linguagem C criado para implementar a rede neuronal e o MATLAB para análise gráfica dos resultados, e também para a comparação entre a implementação em linguagem C e a implementação utilizando o MATLAB. Na Figura 50 é apresentado o fluxograma do programa criado para implementar uma rede neuronal. Nesse fluxograma é mostrado o fluxo de execução do programa quando se pretende simular o funcionamento da RNA.

O programa criado tem como principal funcionalidade a implementação de uma rede neuronal multicamada do tipo *feedforward*, como a apresentada na Figura 43. Para isso, a aplicação efetua a leitura de ficheiros que contêm os pesos a serem utlizados na rede e as entradas a que a rede terá de responder. Depois são calculados os valores de saída da rede neuronal que são escritos num ficheiro, para posteriormente serem analisados no MATLAB.

Um exemplo da implementação em linguagem C de uma rede neuronal, disponibilizado na Internet, foi utilizado como referência para a criação deste programa [16]. Este programa tinha como objetivo efetuar o treino, através do algoritmo de retropropagação, e a simulação de uma RNA utilizando *threads*. Assim, como o objetivo desta tese é bastante diferente da tese que criou o pograma de referência, apenas foi utilizado o código que implementava algumas das funções, como o cálculo do somatório das entradas de um neurónio ou o cálculo do valor de saída das funções de ativação, necessárias para implementar uma RNA em linguagem de programação C. Durante o desenvolvimento deste programa foram desenvolvidas várias versões, que tinham como diferença o objetivo da rede neuronal. Uma

das versões criava uma rede neuronal multicamada proactiva que implementava uma porta lógica XOR, efetuando o treino e simulação da rede.

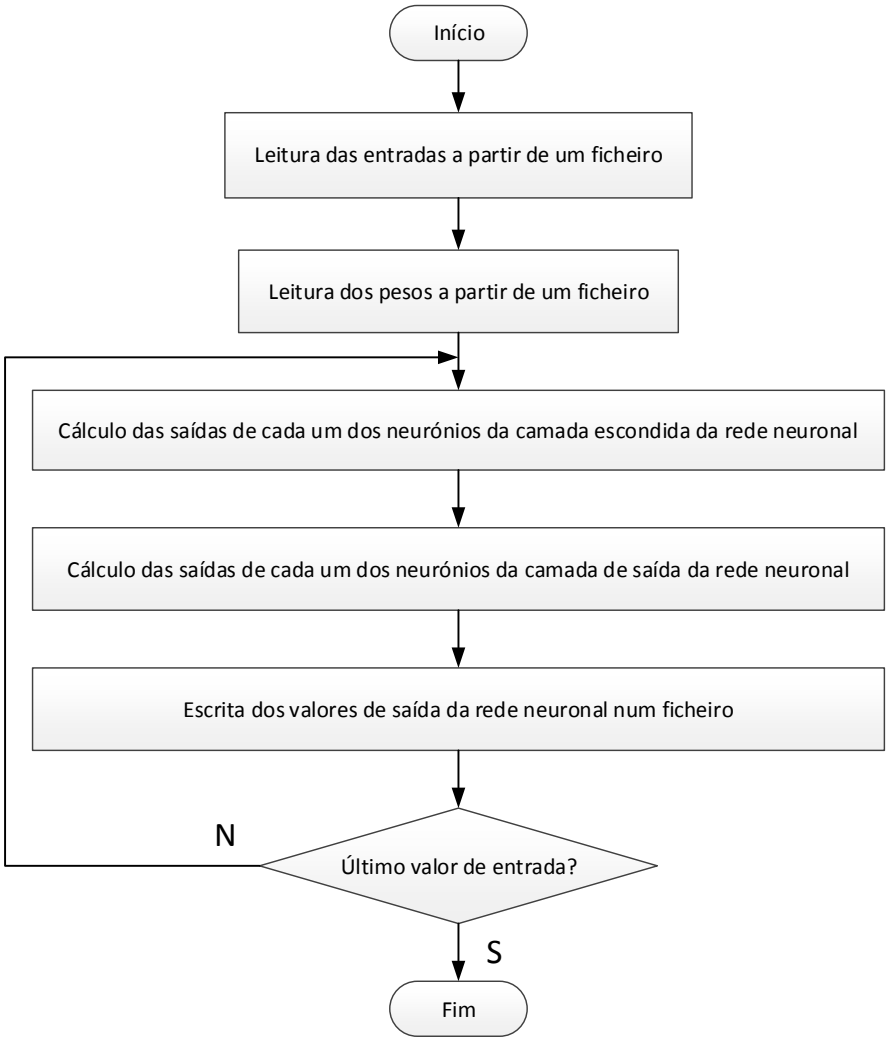


Figura 50 Fluxograma da implementação em linguagem C da rede neuronal

No Anexo A é apresentada a versão final do programa em linguagem C desenvolvido para a implementação da rede neuronal *feedforward* (proactiva).

O programa compilador de linguagem C utilizado foi o `cl.exe` que é disponibilizado no *software* Microsoft Visual Studio, podendo assim ser utlizado sobre um sistema operativo Windows. A execução deste compilador apenas pode ser efetuada através da linha de comandos disponibilizada pelo Visual Studio. Na Figura 51 é apresentado um exemplo da utilização do compilador e da linha de comandos.

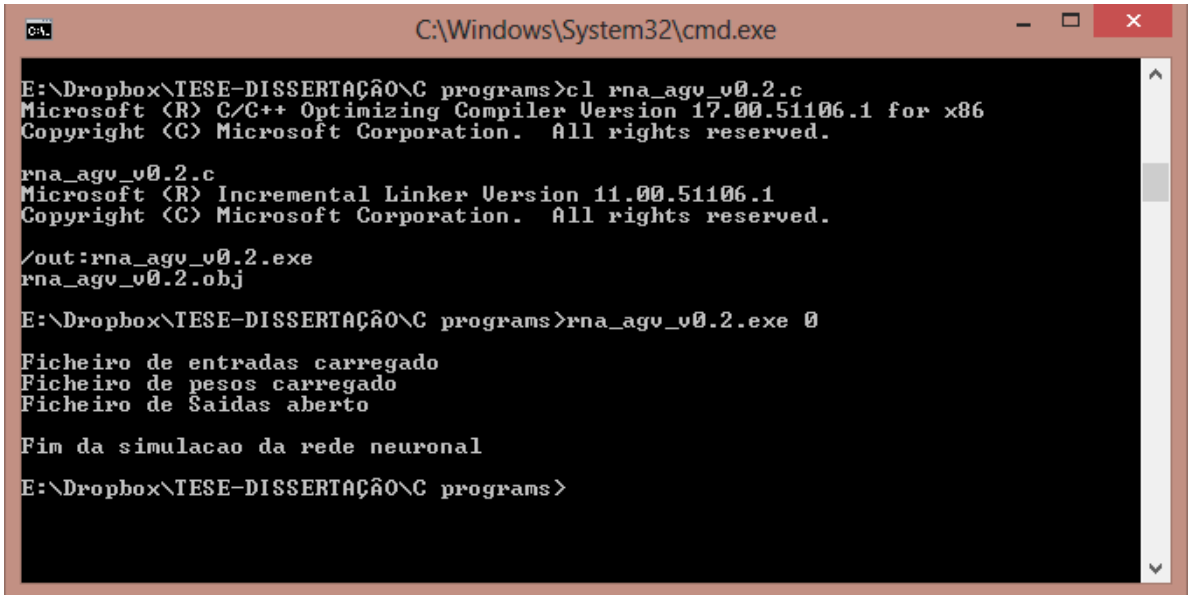


Figura 51 Exemplo da utilização do compilador de linguagem C e da linha de comandos do Microsoft Visual Studio

A versão final do programa tem um parâmetro de entrada aquando da sua execução que indica ao programa o modo de funcionamento pretendido. Este pode ser um de dois: 0 – indica que se pretende apenas simular a rede neuronal; 1 - indica que se pretende treinar a rede neuronal através do algoritmo de retropropagação. No exemplo apresentado na Figura 51 o parâmetro de entrada é 0, pois apenas se pretendia efetuar a simulação da RNA para posteriormente se comparar com a simulação efetuada no MATLAB.

Executando o programa criado, com os pesos e os valores de entradas da rede iguais aos utilizados no MATLAB, simulou-se a rede neuronal que efetua o comportamento de seguir paredes, obtendo-se os resultados apresentados na Figura 52. São apresentados a traço azul os dados experimentais do controlador difuso, a traço magenta os resultados da simulação no MATLAB e o traço a vermelho os resultados da rede neuronal implementada em linguagem C.

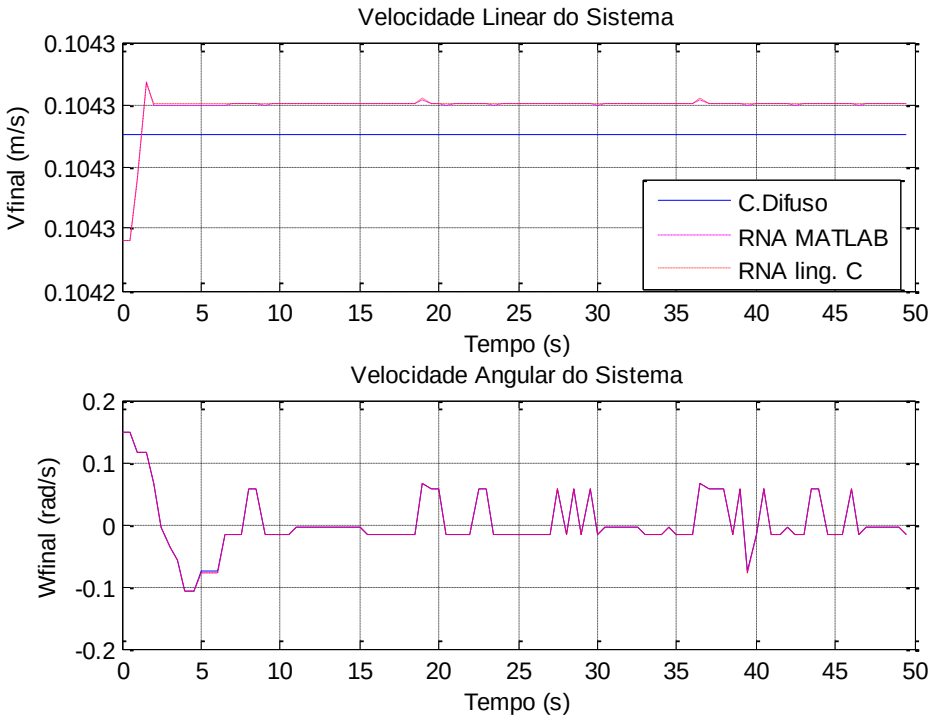


Figura 52 Resposta apresentada pela simulação da rede neuronal utilizando os dados de entrada do treino no programa em linguagem C

Analisando os gráficos da Figura 52 pode-se verificar que os resultados obtidos através da simulação em linguagem C são semelhantes aos resultados obtidos pela simulação no MATLAB, podendo-se concluir que o código do programa executa uma rede neuronal bem implementada e que este pode ser adaptado para a implementação no AGV.

4.4. CRIAÇÃO E TREINO DE UMA RNA QUE IMPLEMENTA O COMPORTAMENTO DE SEGUIR UMA PAREDE COM DESCONTINUIDADES

Após se ter verificado que a rede neuronal que implementava o comportamento mais simples (seguimento de uma parede) funcionava de forma satisfatória, tanto no MATLAB como na implementação em linguagem C, avançou-se para a implementação das redes neuronais que controlam o AGV de forma a este efetuar comportamentos mais complexos, como o seguimento de uma parede com descontinuidades.

Com este objetivo, de forma paralela ao efetuado no caso do comportamento anterior, foram retiradas amostras dos dados de entradas e saída do controlador difuso ao efetuar um trajeto que se enquadrasse no comportamento que se pretende ensinar à rede neuronal e que será

implementada no AGV. O trajeto a ser utilizado para retirar as amostras do controlador neuronal, para posteriormente se efetuar o treino, é apresentado na Figura 53.

Figura 53 Percurso de amostragem do comportamento seguimento de uma parede com descontinuidades

Como se pode verificar pela fotografia, o percurso que o AGV tem de percorrer neste comportamento é mais complexo que no comportamento anterior, pois tem de detetar a parede frontal e desviar-se. Na Figura 54 é apresentado o diagrama exemplificativo da rede neuronal a ser implementada.

No caso deste comportamento é necessário adicionar mais variáveis de entrada para a rede neuronal, tal como foi necessário no controlador difuso. Estas novas variáveis são os valores medidos pelos sensores $S1$ e $S3$, os sonares posicionados na parte frontal do AGV e que permitem ao sistema detetar obstáculos no seu percurso. Assim, a rede neuronal a ser implementada difere da anterior pelo aumento do número de entradas, quatro. O número de neurónios foi mantido o mesmo em ambas as camadas, dez neurónios na camada escondida e dois na de saída.

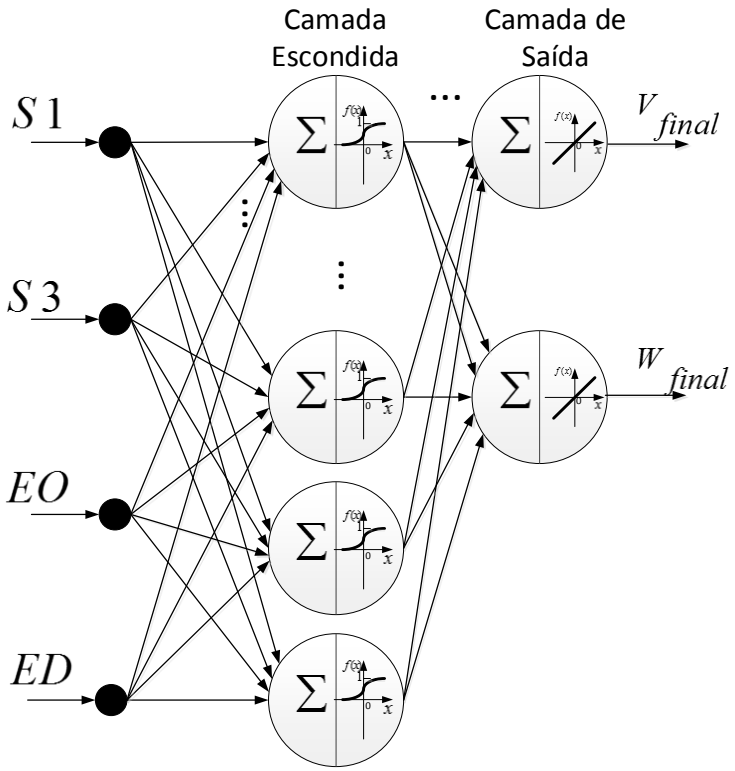


Figura 54 Diagrama da rede neuronal que implementa o comportamento seguir paredes com descontinuidades

O trajeto utilizado para retirar as amostras do controlador difuso é o seguimento de uma parede que faz 90 graus com outra e que na posição inicial tem uma coluna, como se pode ver na Figura 53. Assim, o AGV tem de se aproximar da parede, após o fim da coluna, e posteriormente na proximidade da curva de 90º tem de se direcionar para seguir a nova parede (Figura 53). Os dados amostrados dos sonares frontais do AGV ao ser efetuado este trajeto são apresentados na Figura 55, enquanto na Figura 56 são apresentados os dados de entrada do erro de orientação e erro de distância amostrados durante a execução do trajeto. Os dados de saída do controlador difuso, correspondentes às entradas anteriormente mostradas (Figura 55 e Figura 56), ao percorrer o trajeto (Figura 53), e que são utilizados para treinar a rede neuronal para que esta implemente o mesmo comportamento, são apresentados na Figura 57.

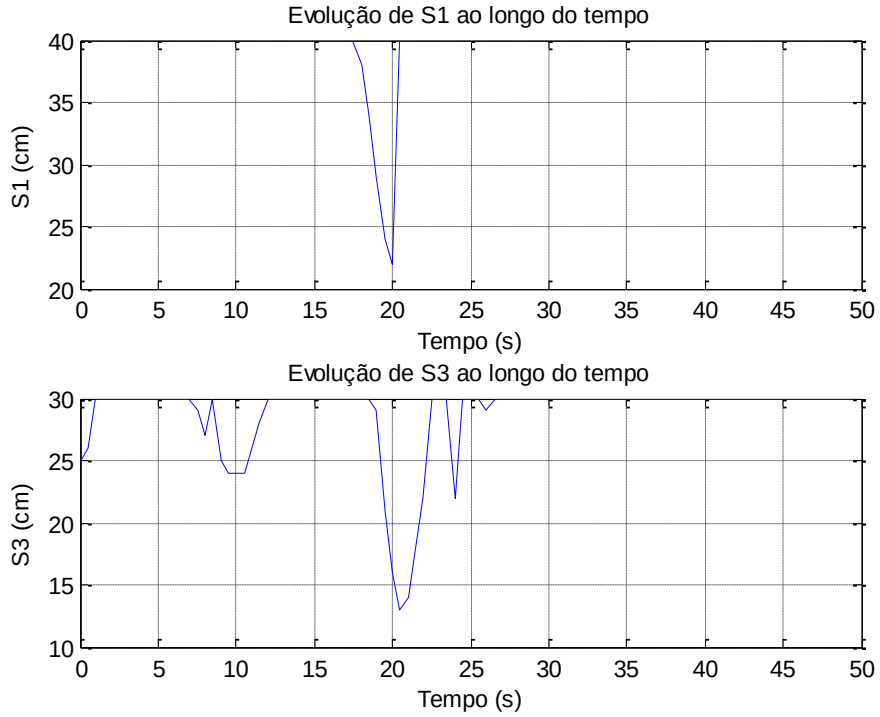


Figura 55 Dados de entrada dos sonares frontais para treinar a rede neuronal a implementar o comportamento seguir paredes com descontinuidades

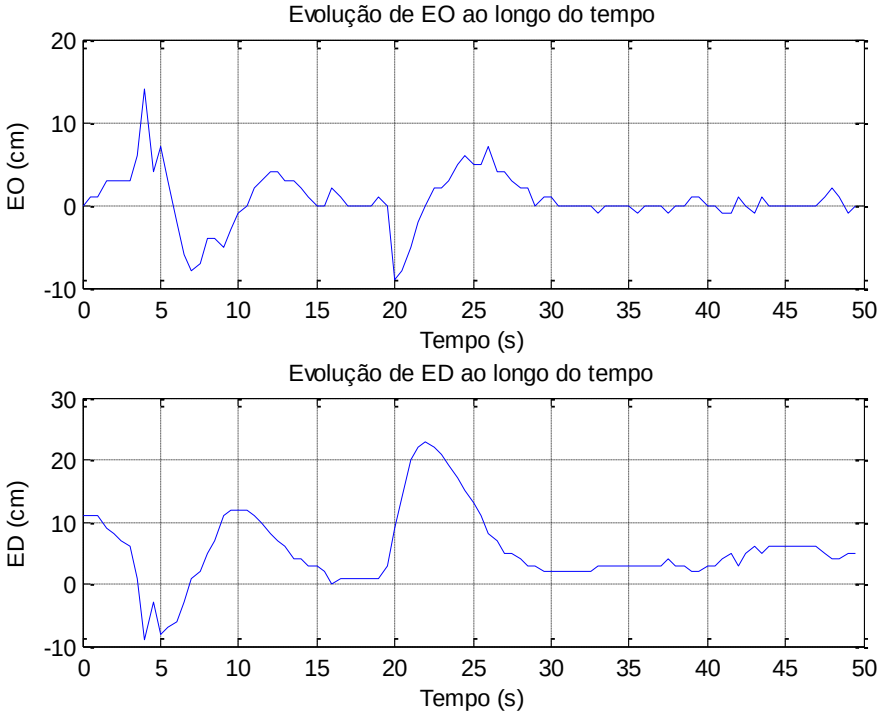


Figura 56 Dados de entrada do controlador difuso para treinar a rede neuronal a implementar o comportamento seguir paredes com descontinuidades

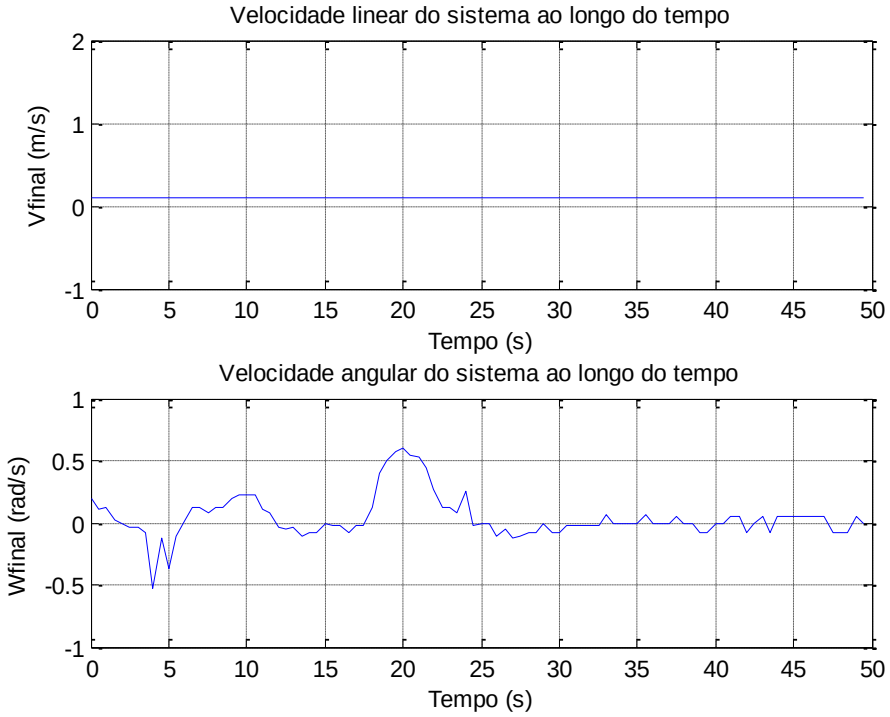


Figura 57 Dados de saída do controlador difuso para treinar a rede neuronal a implementar o comportamento seguir paredes com descontinuidades

Para se efetuar o treino foi novamente utilizado o MATLAB, modificando-se a rotina anteriormente criada apenas adicionando os dois novos vetores das amostras dos dados dos sensores frontais à matriz que será utilizada como dados de entrada para o treino da rede neuronal. O código seguinte mostra os comandos MATLAB utilizados para o efeito.

```
%% Loading training data
data=load('data_curva1.txt');
T=0.5;

%% Defining values to the Network
entradas=[data(:,2) data(:,4) data(:,6)
data(:,7)];
saidas_desejadas=[data(:,9) data(:,10)];

inputs=entradas';
targets=saidas_desejadas';

%% Create a Neural Network
rede = newff(minmax(inputs),[10 2],{'logsig'
'purelin'});

%% Initialize weights
rede=init(rede)
```

```
pesosIL = rede.IW{1}
pesosLW = rede.LW{2}
polaridade1 = rede.b{1}
polaridade2 = rede.b{2}

%%Train network
rede.trainParam.showCommandLine = 1;
rede.trainParam.show = 10;
rede.trainParam.lr = 0.01;
rede.trainParam.mc = 0.9;
rede.trainParam.epochs = 1000;
rede.trainParam.goal = 0;
[rede treino] = train(rede,inputs,targets);
treino.perf(end)
pesosIL = rede.IW{1}
pesosLW = rede.LW{2}
polaridade1 = rede.b{1}
polaridade2 = rede.b{2}
pause
nntraintool('close')

%% Simulate the Network
data_sim=load('dados_sim_curva_codigoC.txt');
saidas=sim(rede,inputs);
subplot(211);
plot(data(:,1)'*T,targets(1,:),'b');
grid on;holdon;
plot(data(:,1)'*T,saidas(1,:),'r');
plot(data(:,1)'*T,data_sim(:,3)','g')
xlabel('Tempo(s)');ylabel('rad/s');
title('Velocidade Linear do Sistema');
subplot(212);
plot(data(:,1)'*T,targets(2,:),'b');
grid on;holdon;
plot(data(:,1)'*T,saidas(2,:),'r');
plot(data(:,1)'*T,data_sim(:,5)','g')
xlabel('Tempo(s)');ylabel('rad/s');
title('Velocidade Angular do Sistema');
pause
close all
```

Esta matriz fica composta pelos dados dos sensores *S*1 e *S*3, erro de orientação e erro de distância. O treino da rede neuronal foi completado com sucesso ao fim de 254 iterações por se ter atingido o mínimo gradiente, como apresentado na Figura 58.

Na Figura 59 são apresentados os resultados da simulação da rede neuronal para o caso utilizado no treino. O traço azul corresponde aos dados utilizados no treino, o magenta aos resultados do MATLAB e o vermelho aos resultados da implementação em C.

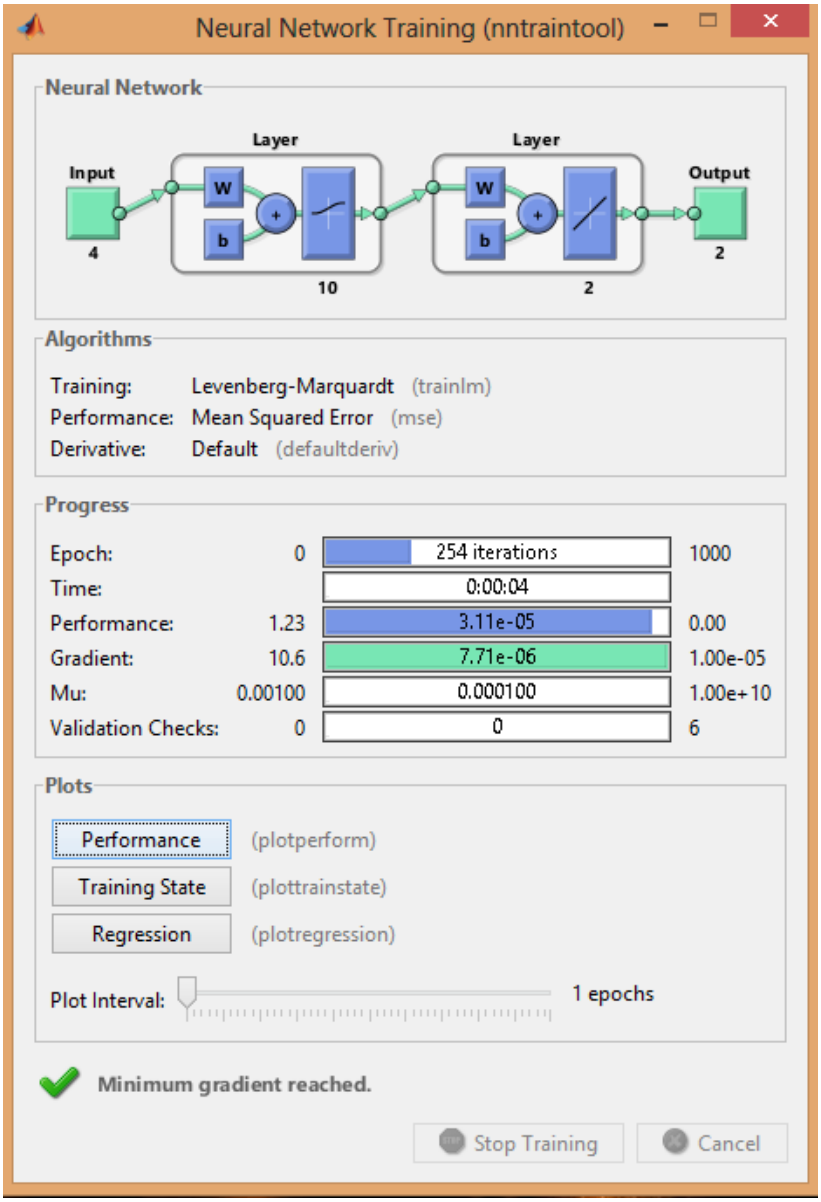


Figura 58 Informação relativa ao treino da rede neuronal

Como se pode verificar pela análise destes gráficos (Figura 59), os resultados obtidos pela implementação de uma rede neuronal em C e no MATLAB são semelhantes à implementação do controlador difuso, apenas se notando uma diferença mínima entre a resposta da velocidade linear do sistema da rede neuronal (o traço a cor magenta e o traço a cor vermelha estão sobrepostos) e a do controlador difuso (traço azul no fundo do primeiro gráfico).

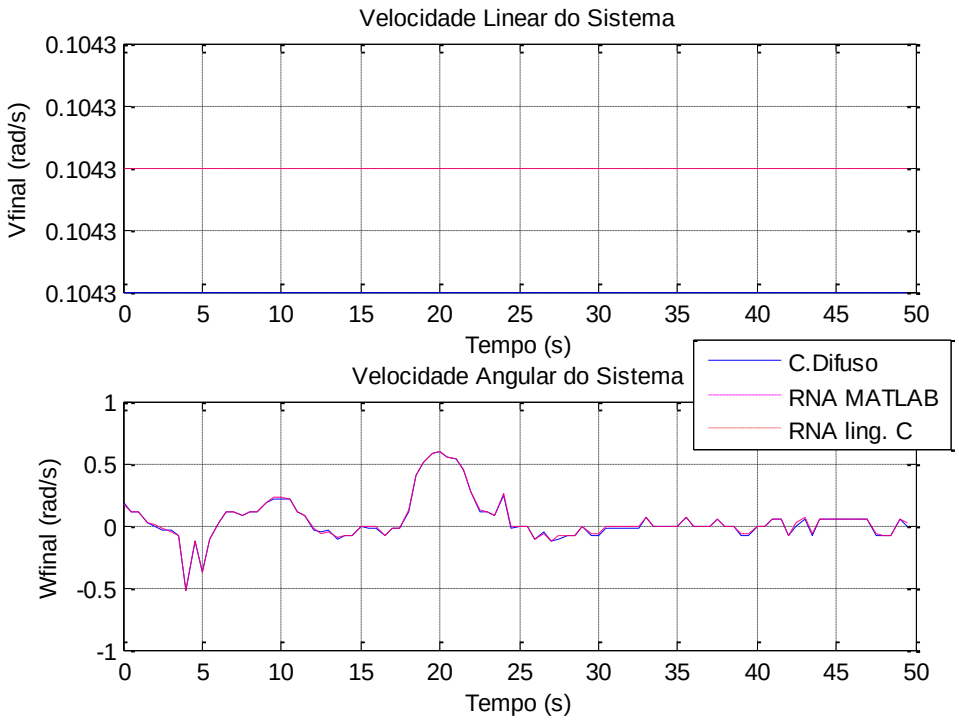


Figura 59 Resposta apresentada pela simulação da rede neuronal para implementar o comportamento seguir paredes com descontinuidades utilizando os dados de entrada do treino no programa em C

4.5. CRIAÇÃO E TREINO DE UMA RNA QUE IMPLEMENTA O SEGUIMENTO DE UMA PAREDE COM DESCONTINUIDADES E O DESVIO DE OBSTÁCULOS

Como referido anteriormente, o desenvolvimento de uma rede neuronal através dos dados amostrados a partir do controlador difuso foi efetuado por passos, que se diferenciavam pelo aumento gradual da complexidade comportamental exigida ao funcionamento do robô. Assim, nesta terceira fase foi escolhido um trajeto bastante mais complexo que o anterior, obrigando o AGV, controlado pelo controlador de lógica difusa a desviar-se de um obstáculo no seu percurso e seguir paredes descontínuas. O percurso a ser efetuado pelo AGV é caraterizado por ter secções onde este terá de inverter a marcha para conseguir seguir o percurso desejado. As secções onde isto se verifica é a zona posterior ao caixote do lixo e a zona posterior ao armário. Na Figura 60 é apresentado uma fotografia que apresenta o AGV na posição inicial antes de percorrer o percurso de treino.

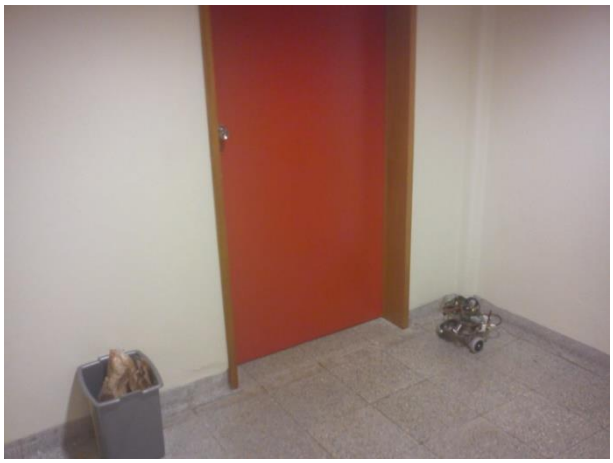


Figura 60 Posição inicial do AGV ao efetuar o percurso de treino

Na Figura 61 são apresentadas fotografias que ilustram as duas primeiras secções do percurso efetuado pelo AGV, controlado através de lógica difusa, para se retirar os dados utilizados no treino da rede neuronal. Como se pode verificar pelas imagens, o AGV tem de se desviar do caixote do lixo e posteriormente, como se tenta aproximar novamente da parede que seguia, ao detetar a parede frontal, o robô recua e desloca-se para seguir essa mesma parede frontal.

a) **b)**

Figura 61 Primeira secção do percurso (a) e segunda secção do percurso (b)

Na Figura 62 é apresentado o resto do percurso (mais duas secções). Na terceira secção do percurso, após o AGV seguir o armário, acontece uma situação semelhante ao referido na primeira secção devido a um espaçamento entre o armário e a parede que se encontra a seguir. Isto é visível na fotografia que apresenta a quarta secção do percurso na Figura 62b).

a) **b)**

Figura 62 Terceira secção do percurso (a) e quarta secção do percurso (b)

Os dados amostrados das medições dos sonares frontais *S*1 e *S*3 do AGV, que são entradas do controlador difuso, e que foram recolhidos do percurso e recebidos via Bluetooth, são apresentados na Figura 63. Na Figura 64 são apresentados os valores das entradas, erro de orientação e erro de distância, que também foram recolhidas da mesma forma, e ao mesmo tempo que as anteriores. Os valores de saída do controlador difuso às entradas anteriormente apresentadas e recolhidos de uma forma idêntica aos valores de entrada são mostrados na Figura 65.

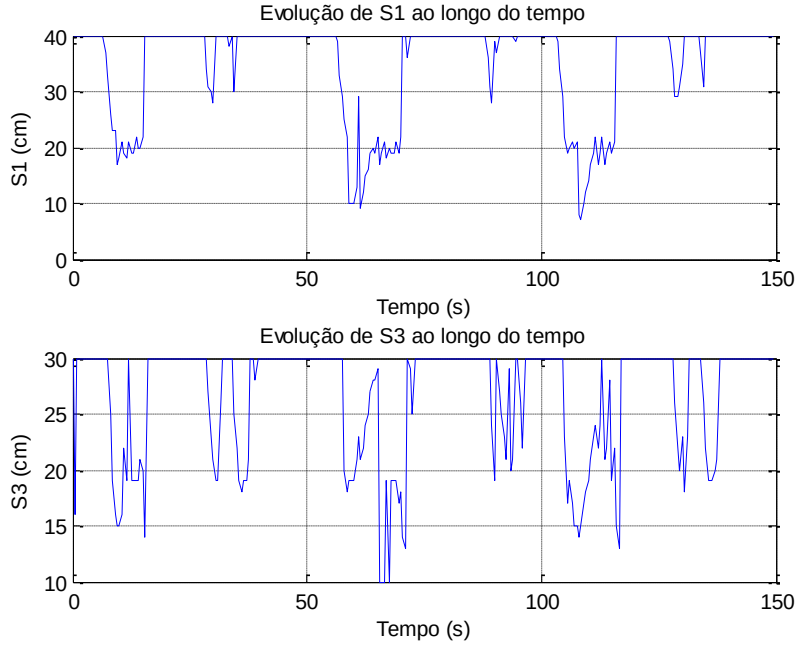


Figura 63 Dados de entrada, *S*1 e *S*3, do controlador difuso para treinar a rede neuronal a implementar o comportamento seguir paredes com descontinuidades e desviar-se de obstáculos

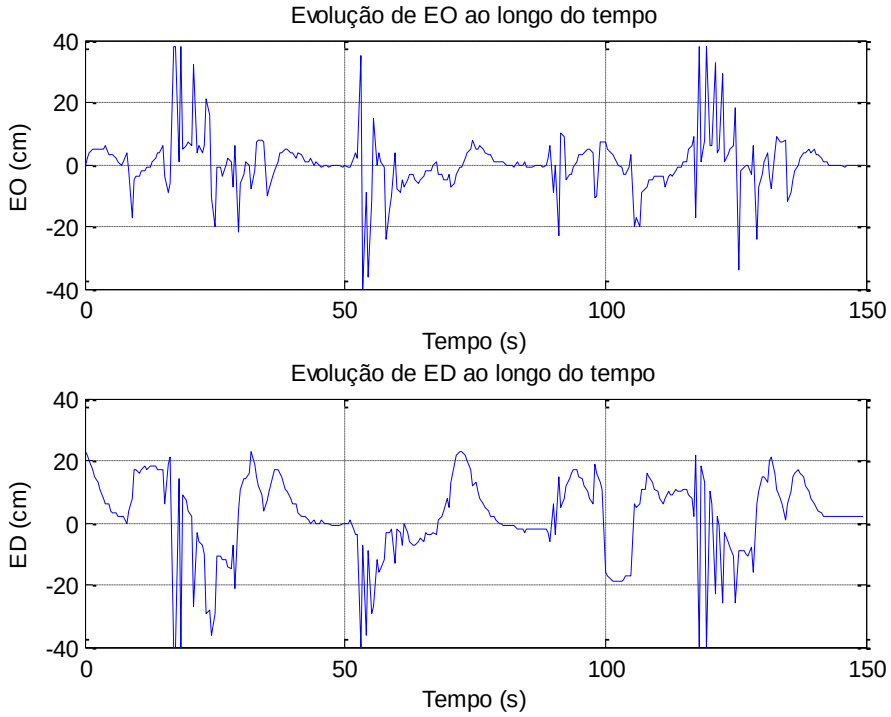


Figura 64 Dados de entrada, *EO* e *ED*, do controlador difuso para treinar a rede neuronal a implementar o comportamento seguir paredes com descontinuidades e desviar-se de obstáculos

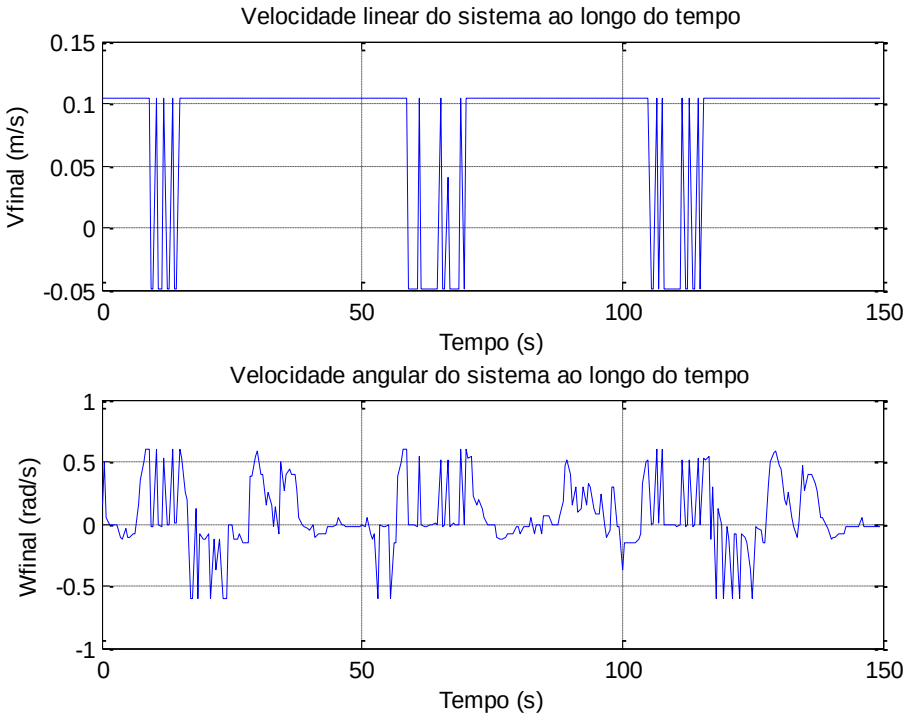


Figura 65 Dados de saída do controlador difuso para treinar a rede neuronal a efetuar o comportamento seguir paredes com descontinuidades e desviar-se de obstáculos

Com a obtenção destas amostras pode-se efetuar o treino de uma rede neuronal tal como foi efetuado anteriormente para os comportamentos menos complexos. A rede neuronal implementada é do tipo multicamada *feedforward*, semelhante às redes neuronais anteriormente implementadas em todos os outros aspetos, como se pode verificar pelo código seguinte utilizado para o efeito.

```
 %% Loading training data
data=load('data_emergencia.txt');
T=0.5;

%% Defining values to the Network
entradas=[data(:,2) data(:,4) data(:,6)
data(:,7)];
saidas_desejadas=[data(:,9) data(:,10)];

inputs=entradas';
targets=saidas_desejadas';

%% Create a Neural Network
rede = newff(minmax(inputs),[10 2],{'logsig'
'purelin'});

%% Initialize weights
rede=init(rede)
pesosIL = rede.IW{1}
pesosLW = rede.LW{2}
polaridade1 = rede.b{1}
polaridade2 = rede.b{2}

%% Train network
rede.trainParam.showCommandLine = 1;
rede.trainParam.show = 10;
rede.trainParam.lr = 0.01;
rede.trainParam.mc = 0.9;
rede.trainParam.epochs = 1000;
rede.trainParam.goal = 0;
[rede treino] = train(rede,inputs,targets);
treino.perf(end)
pesosIL = rede.IW{1}
pesosLW = rede.LW{2}
polaridade1 = rede.b{1}
polaridade2 = rede.b{2}
pause
nntraintool('close')

%% Simulate the Network
data_sim=load('dados_sim_emergencia_codigoC.txt')
;
saidas=sim(rede,inputs);
subplot(211);
plot(data(:,1)'*T,targets(1,:),'k-');
grid on;hold on;
plot(data(:,1)'*T,saidas(1,:),'k--');
```

```
plot(data(:,1)'*T,data_sim(:,3)','K-.')
xlabel('Tempo(s)');ylabel('rad/s');
title('Velocidade Linear do Sistema');
subplot(212);
plot(data(:,1)'*T,targets(2,:),'k-');
grid on;hold on;
plot(data(:,1)'*T,saidas(2,:),'k--');
plot(data(:,1)'*T,data_sim(:,5)','k-.')
xlabel('Tempo(s)');ylabel('rad/s');
title('Velocidade Angular do Sistema');
pause
close all
```

Para melhorar o desempenho da rede neuronal optou-se por aumentar o número de neurónios existentes na camada escondida. Assim, aumentou-se o número de neurónios dessa camada de 10 para 12 neurónios. Treinando a rede após esta modificação obtiveram-se os resultados para as variáveis de saída, velocidade linear e velocidade angular do sistema, apresentados na Figura 66 e Figura 67, sendo o traço azul a resposta do controlador difuso, o traço vermelho a resposta da rede neuronal com 12 neurónios na camada escondida e o traço magenta a resposta da rede neuronal com 10 neurónios.

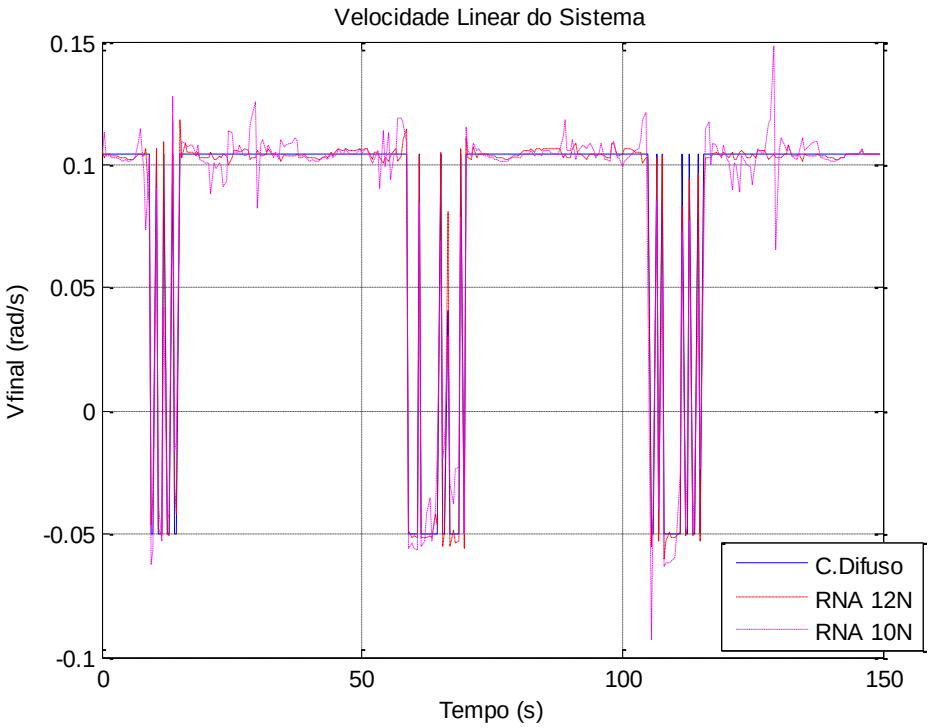


Figura 66 Resultados da simulação da rede neuronal que implementa o comportamento seguir paredes com descontinuidades e desviar-se de obstáculos no MATLAB: velocidade linear

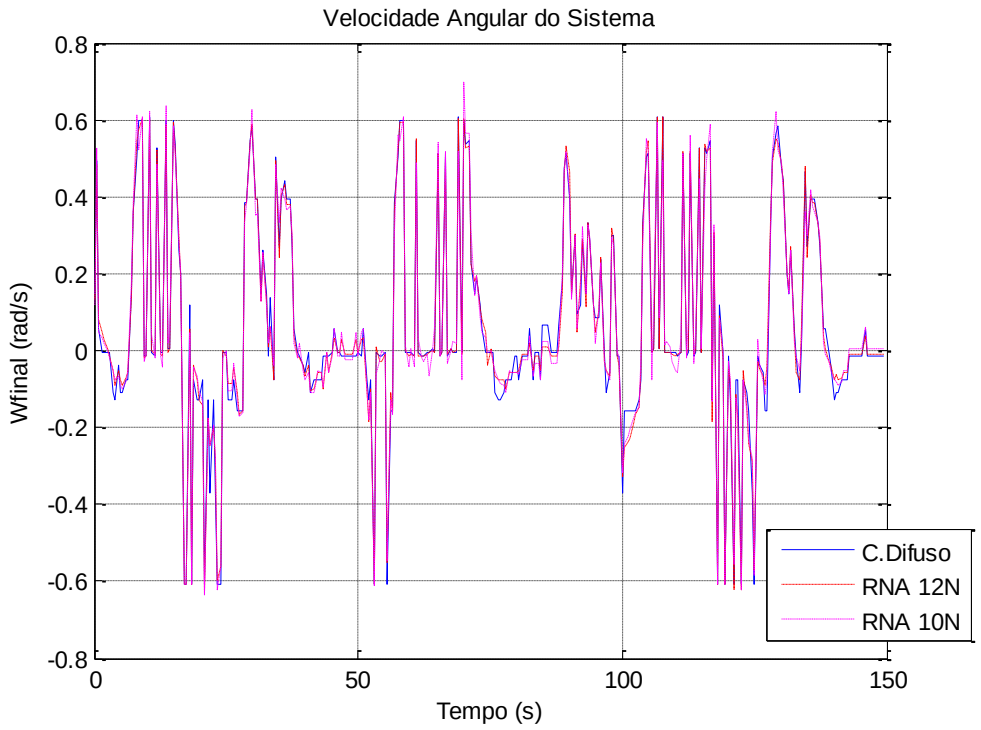


Figura 67 Resultados da simulação da rede neuronal que implementa o comportamento seguir paredes com descontinuidades e desviar-se de obstáculos no MATLAB: velocidade angular

Como se pode verificar pela análise detalhada dos gráficos apresentados anteriormente, os valores de saída da rede neuronal em comparação com os valores do controlador difuso são mais semelhantes do que se verificou na situação anterior com menor número de neurónios na camada escondida, logo o desempenho da rede neuronal, medido através do valor de *Mean Squared Error* (MSE), é superior como foi verificado. O valor do MSE no primeiro caso é de 0,0011 e no segundo caso é de $4{,}3791\times10^{-4}$.

Porém, não se deve concluir que quanto mais neurónios existirem numa camada melhor será a resposta, pois isto não é verdade. A realidade é que um valor muito alto de neurónios significa a especialização da rede neuronal, o que a leva a responder perfeitamente aos valores de entrada treinados, mas se outros valores de entrada lhe forem apresentados a rede neuronal poderá responder de uma forma que não era esperada. Para se resolver este problema deve treinar-se com o menor número possível de neurónios, garantindo-lhe uma capacidade de estimação dos valores de saída generalizada da resposta pretendida, conseguindo assim escapar do problema de especialização.

Como efetuado nos comportamentos anteriores, na Figura 68 são apresentados os valores de saída da rede neuronal implementada em linguagem C, utilizando os pesos obtidos pelo

treino no MATLAB, comparando-os com os valores de saída da rede neuronal implementada no MATLAB e os valores de saída utilizados no treino obtidos através do controlador difuso.

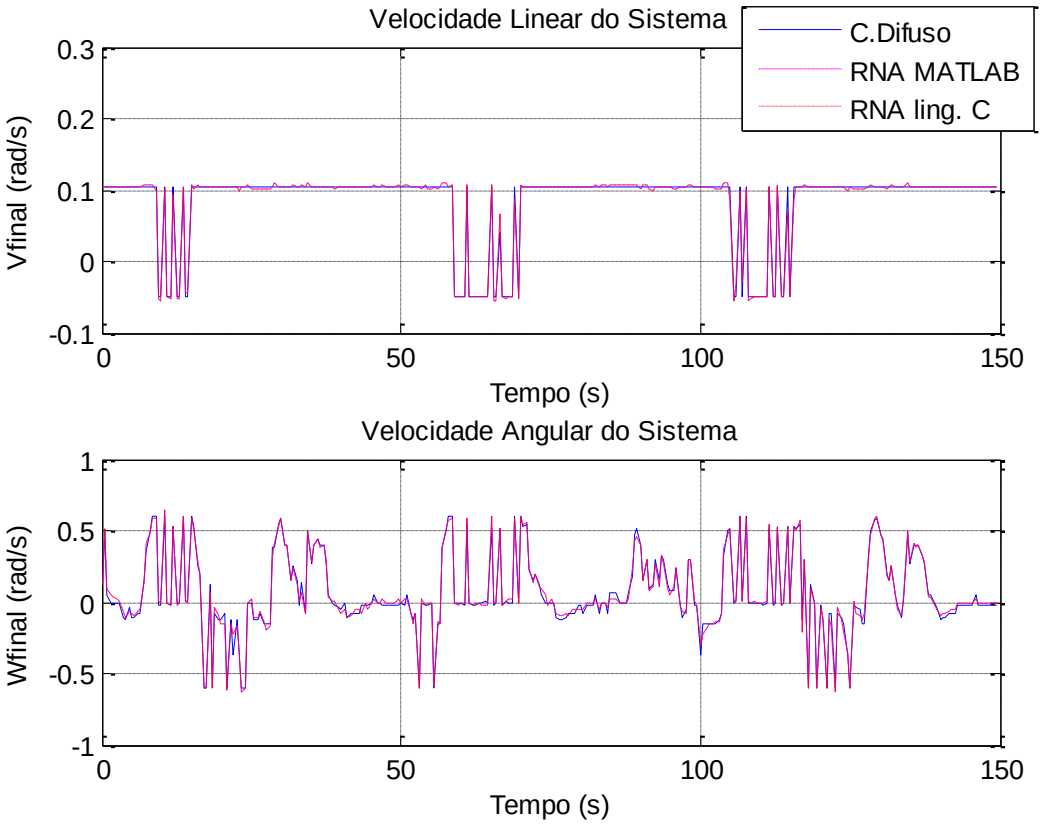


Figura 68 Resposta apresentada pela simulação da rede neuronal para efetuar o comportamento seguir paredes com descontinuidades e desvio de obstáculos, utilizando os dados de entrada do treino no programa em linguagem C

Tal como verificado anteriormente nos comportamentos menos complexos, a implementação da rede neuronal em linguagem C (traço vermelho) apresenta valores de saída semelhantes aos valores de saída da implementação no MATLAB (traço magenta) e não se diferenciam dos valores de saída apresentados pelo controlador difuso (traço azul) num nível que influencie o desempenho deste comportamento comparando as duas vertentes da sua implementação.

Conclui-se que os resultados obtidos pelas simulações das RNA após o treino da rede no MATLAB são satisfatórios. Além disso, comparando os resultados obtidos pelas simulações no MATLAB com os obtidos no programa em linguagem de programação C, pode-se

afirmar que a programação realizada funciona corretamente, podendo-se implementar no AGV uma adaptação desse programa.

5. TESTES E RESULTADOS

Neste capítulo são abordadas as implementações das redes neuronais no AGV e os resultados obtidos nos vários testes experimentais realizados. Primeiro são apresentados os resultados obtidos com cada uma das RNA criadas e simuladas no capítulo anterior nos mesmos percursos utilizados para retirar as amostras do controlo difuso. Também são relatados os resultados obtidos com a RNA num percurso não treinado. Posteriormente, é relatada a implementação de uma RNA a partir do controlo remoto e os seu resultados. Por fim é apresentado um resumo da análise dos resultados obtidos.

5.1. IMPLEMENTAÇÃO DA RNA NUM MICROCONTROLADOR PIC

Após se ter efetuado um projeto satisfatório das redes neuronais através do MATLAB, referido nas secções 4.2, 4.4 e 4.5 do quarto capítulo e se ter verificado a sua funcionalidade através de um programa desenvolvido na linguagem de programação C, referido na secção 4.3, que simula o funcionamento da rede neuronal, passou-se à fase seguinte, a implementação da rede neuronal no microcontrolador do AGV que se pretende controlar. Anteriormente, este microcontrolador alojava o controlador difuso que foi modificado para se retirar amostras para o treino da rede neuronal, como referido na secção 4.1.

Para criar o programa da rede neuronal foi utilizado o programa do controlador difuso como base, eliminando o código desnecessário relativo ao controlador difuso, que foi substituído

pelo código que implementa a rede neuronal. O código que implementa a rede neuronal é semelhante ao implementado no computador, tendo sido este também usado como base, adicionando o código necessário. O fluxograma que representa o funcionamento do controlo através da rede neuronal artificial é apresentado na Figura 69.

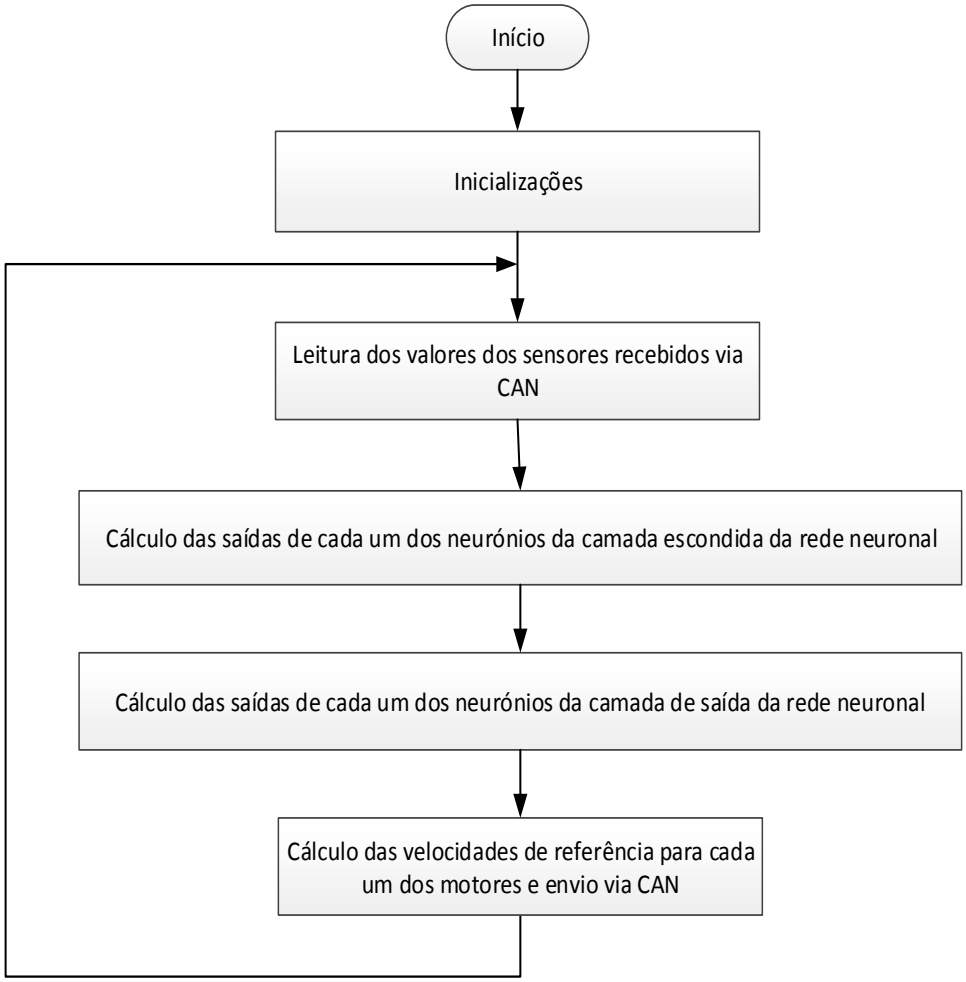


Figura 69 Fluxograma da implementação no µC da rede neuronal artificial

Os pesos e as polarizações utilizadas na implementação em microcontrolador são as mesmas obtidas nas redes neuronais criadas e simuladas nas secções 4.2 até à 4.5. Os pesos e as polarizações obtidos no MATLAB são escritos num ficheiro de texto através do comando `dlmwrite`, que tem como parâmetros o nome do ficheiro e a variável a escrever no ficheiro. Adicionando a opção `-append` garante-se que se o ficheiro já existir não é reescrito. Posteriormente estes pesos são adicionados ao código do programa que implementa a rede

neuronal artificial no microcontrolador. Em seguida são apresentados os testes experimentais efetuados e os resultados obtidos.

5.1.1. IMPLEMENTAÇÃO DE UMA RNA QUE IMPLEMENTA O COMPORTAMENTO DE SEGUIR A PAREDE

A RNA que efetua este comportamento apresenta os pesos e as polarizações apresentadas no extrato de código seguinte, retirado do programa implementado no microcontrolador de controlo do AGV. Esta rede, como foi referido na secção 4.2, é constituída por 2 entradas (`NUM_INPUTS_HL`), 10 neurónios na camada escondida (`NUM_NEURONS_HL` e `NUM_INPUTS_OL`) e 2 neurónios na camada de saída (`NUM_NEURONS_OL`).

```
float weights_HL[NUM_INPUTS_HL*NUM_NEURONS_HL]={
2.8541,1.2563,3.2044,-2.4793,-1.9700,-0.9356,
-1.9799,-2.1162,0.9033,1.4534,-0.1028,-0.3116,
1.4073,1.7258,1.4529,-0.6962,-2.0023,1.1052,
1.7641,-0.5878};

float weights_OL[NUM_INPUTS_OL*NUM_NEURONS_OL]={
0.0000,-0.0000,-0.0000,-0.0000,0.0000,0.0000,
0.0000,0.0007,0.0001,-0.0794,0.1339,0.0812,
0.4510,0.1561,-0.6254,-2.1915,-0.3741,0.1279,
-0.0699,0.7637};

float bias_HL[NUM_NEURONS_HL]={
-15.7863,-1.6361,13.6711,5.7759,-7.0909,1.0287,
-1.6252,9.6354,-11.4682,14.2626};

float bias_OL[NUM_NEURONS_OL]={0.1830,0.1598};
```

O vetor `weights_HL` identifica os valores dos pesos de cada uma das entradas em cada um dos neurónios da camada escondida da rede neuronal. Neste caso, este vetor é constituído por 20 valores (2 entradas vezes 10 neurónios). O vetor `weights_OL` identifica os valores dos pesos de cada uma das entradas em cada um dos neurónios da camada de saída da rede neuronal. Neste caso, este vetor é constituído por 20 valores (10 entradas – as saídas dos neurónios da camada anterior – vezes 2 neurónios). O vetor `bias_HL` identifica os valores das polarizações de cada um dos neurónios da camada escondida da rede neuronal e o vetor `bias_OL` identifica os valores das polarizações de cada um dos neurónios da camada de saída.

Com a implementação desta rede neuronal multicamada proactiva 2-10-2 obtiveram-se os resultados apresentados na Figura 70 que representa, através das medições efetuadas pelos sonares, o AGV a percorrer o trajeto treinado, apresentado na secção 4.2 do Capítulo 4.

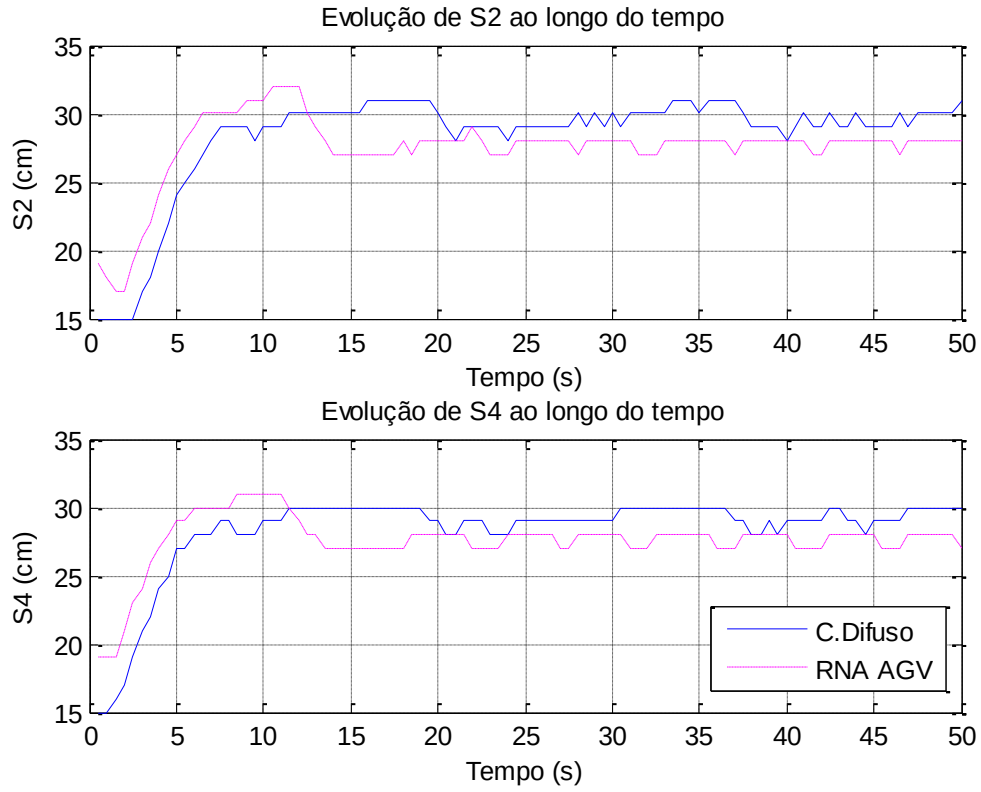


Figura 70 Resultados experimentais obtidos com a RNA que implementa o comportamento seguir a parede: medições dos sensores

Na Figura 71 é representado, através da evolução das entradas da rede, o percurso efetuado pelo AGV a percorrer o trajeto treinado apresentado na secção 4.2 do Capítulo 4.

Como se pode verificar pela análise dos gráficos apresentados, a resposta real do AGV com o controlo por RNA, apresentada a traço de cor magenta, é semelhante ao controlo difuso que estava anteriormente implementado e que foi utilizado como objetivo do treino realizado da rede neuronal, apresentado a traço de cor azul. Existem pequenas diferenças entre os valores apresentados quando comparando o treino e o teste experimental, sendo a diferença mais relevante, a distância do AGV à parede, que neste caso não tendeu para 30 cm, valor de referência utilizado, mas para 27,5 cm.

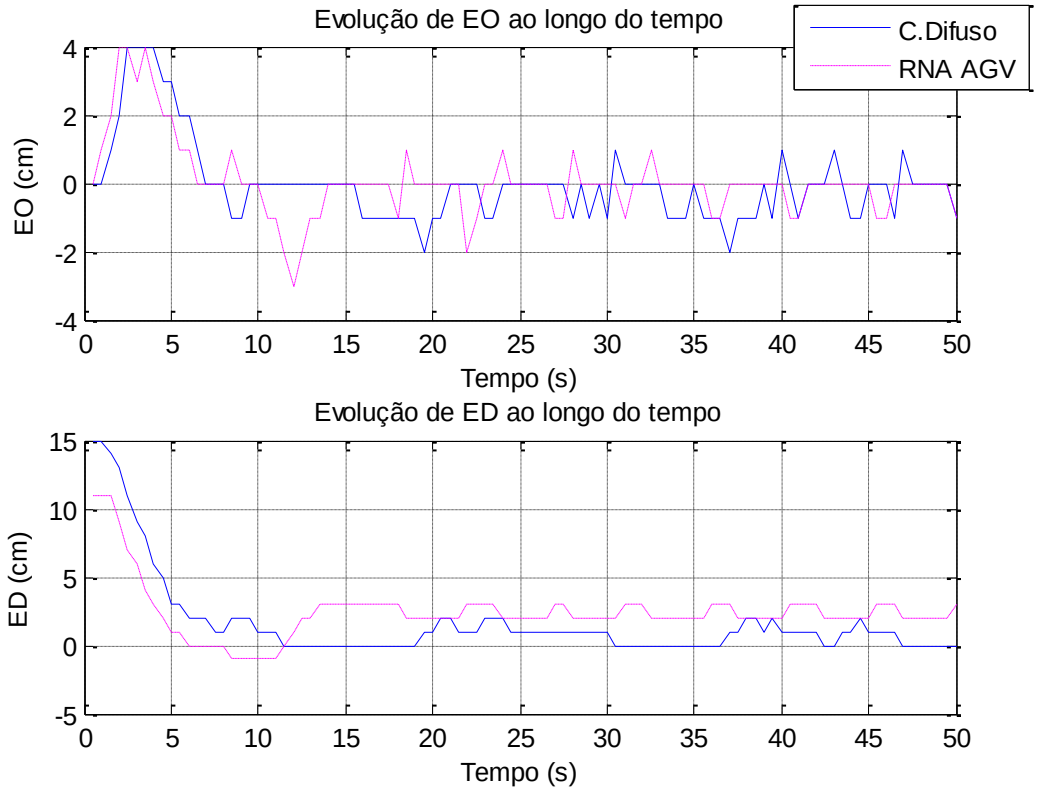


Figura 71 Resultados experimentais obtidos com a RNA que implementa o comportamento seguir a parede: erros calculados

5.1.2. IMPLEMENTAÇÃO DE UMA RNA QUE IMPLEMENTA O SEGUIMENTO DE UMA PAREDE COM DESCONTINUIDADES

A RNA que efetua este comportamento apresenta os pesos e as polarizações apresentadas no extrato de código seguinte, retirado do programa implementado no AGV. Esta rede, como foi referido na secção 4.4, é constituída por 4 entradas (`NUM_INPUTS_HL`), 10 neurónios na camada escondida (`NUM_NEURONS_HL` e `NUM_INPUTS_OL`) e 2 neurónios na camada de saída (`NUM_NEURONS_OL`).

```
float weights_HL[NUM_INPUTS_HL*NUM_NEURONS_HL]={
0.3346,-0.2951,0.1648,0.1102,0.6428,-0.2730,
-0.4544,-0.2408,-0.1632,-0.3608,-0.0062,0.6252,
0.6994,-0.7144,-0.2874,0.0870,3.4358,-4.9570,
-3.1577,0.6046,-0.8728,0.9301,0.7585,-2.0701,
-0.0019,0.0161,-0.7414,-0.5440,-1.0747,0.9160,
```

```
1.3062,-0.0838,-0.4393,0.8297,0.3482,-0.9793,
-0.4592,0.3372,-0.1938,-0.1078};

float weights_OL[NUM_INPUTS_OL*NUM_NEURONS_OL]={
0.0000,-0.0000,0.0000,0.0000,-0.0000,0.0000,
0.0000,-0.0000,-0.0000,0.0000,0.4902,0.2782,
0.3526,-0.8719,0.1646,1.8353,0.2589,-0.3251,
-2.2737,0.4747};

float bias_HL[NUM_NEURONS_HL]={
-7.7557,-15.5130,13.2493,-6.1659,9.4512,3.8163,
-0.2440,10.5114,-9.7417,5.1061};

float bias_OL[NUM_NEURONS_OL]={0.1043,0.1525};
```

De forma análoga à rede neuronal implementada anteriormente, cada um dos vetores apresentados anteriormente refere-se às caraterísticas da rede neuronal artificial obtidas pelo treino, pesos das ligações e polaridades dos neurónios. Nesta rede neuronal, em comparação com a rede neuronal anterior, o número de entradas passou de 2 para 4, logo o vetor que se refere aos pesos das ligações de entrada, `weights_HL`, tem um tamanho superior, 40 valores (4 entradas vezes 10 neurónios).

Com a implementação desta rede neuronal multicamada proactiva 4-10-2 obtiveram-se os resultados apresentados que representam o percurso efetuado pelo AGV a percorrer o trajeto treinado apresentado na secção 4.4; na Figura 72, através das variáveis de entrada *S*1 e *S*3, e na Figura 73, através das variáveis *EO* e *ED*.

Os resultados obtidos pelo teste experimental (traço de cor magenta) quando comparados com os dados utilizados no treino (traço de cor azul) não são iguais. Se a rede se comportasse de forma exatamente igual ao controlador difuso os traços apresentados sobrepor-se-iam, o que não acontece. Mas as variações apresentadas pelos traços são semelhantes, podendo assim concluir-se que a rede neuronal implementada comporta-se de forma semelhante ao controlador difuso.

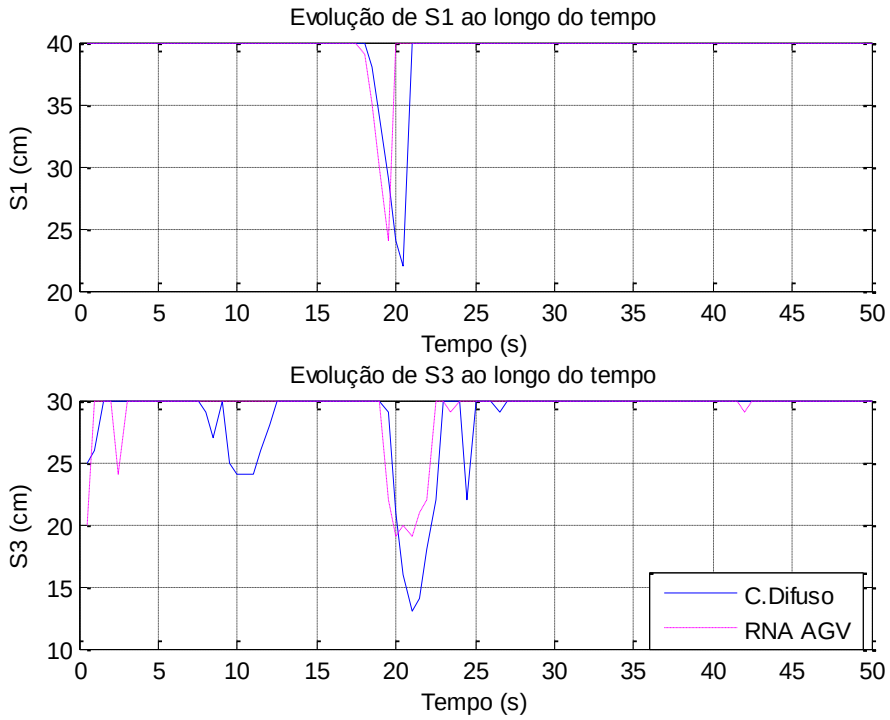


Figura 72 Resultados experimentais obtidos com a RNA que implementa o seguimento de uma parede com descontinuidades: sensores de deteção de obstáculos

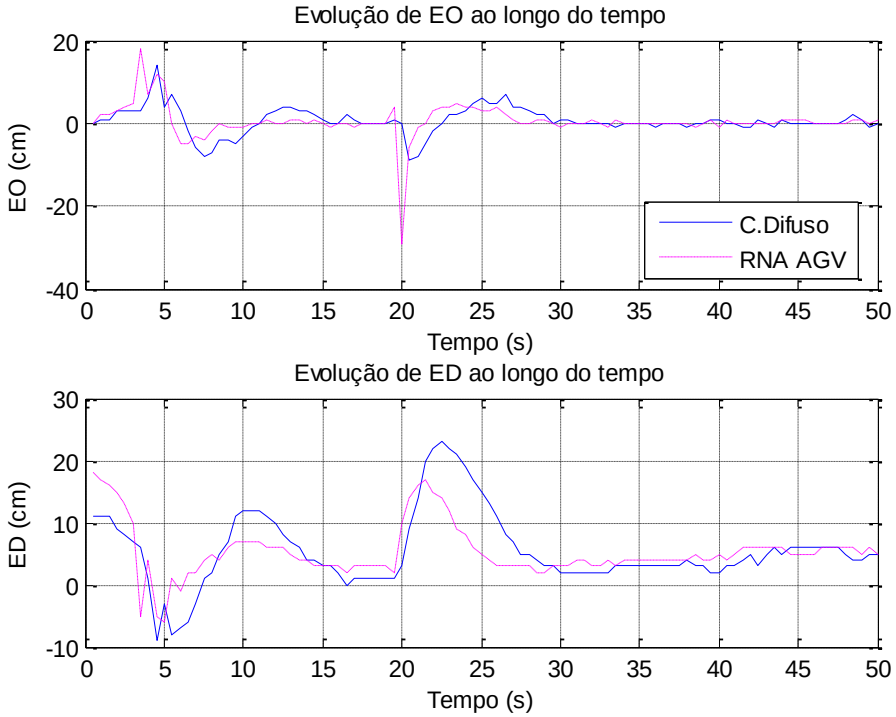


Figura 73 Resultados experimentais obtidos com a RNA que implementa o seguimento de uma parede com descontinuidades: erros calculados

5.1.3. IMPLEMENTAÇÃO DE UMA RNA QUE IMPLEMENTA O SEGUIMENTO DE UMA PAREDE COM DESCONTINUIDADES E O DESVIO DE OBSTÁCULOS

A RNA que efetua este comportamento apresenta os pesos e as polarizações apresentadas no extrato de código seguinte, retirado do programa implementado no AGV. Esta rede, como foi referido na secção 4.5, é constituída por 4 entradas (`NUM_INPUTS_HL`), 12 neurónios na camada escondida (`NUM_NEURONS_HL` e `NUM_INPUTS_OL`) e 2 neurónios na camada de saída (`NUM_NEURONS_OL`).

```
float weights_HL[NUM_INPUTS_HL*NUM_NEURONS_HL]={
-0.7338,1.3681,0.1000,-0.8677,-0.0619,-0.3327,
-0.1198,0.0672,0.6205,0.1674,-0.0946,0.0438,
-0.2026,0.2136,0.0883,-0.4365,-0.0146,-0.2582,
-0.1755,0.0695,-12.0797,2.7665,-7.0710,-14.3571,
5.6687,-0.1504,-0.1541,0.1829,-0.2321,0.4538,
-0.6432,-0.2246,-0.0177,-0.2652,-0.1857,0.0729,
11.1072,-6.6221,18.6269,-4.8226,1.1761,-0.0788,
-0.6787,0.6004,-1.5523,1.9146,-0.3669,0.5419};

float weights_OL[NUM_INPUTS_OL*NUM_NEURONS_OL]={
0.0101,0.0014,-0.0031,-0.0111,-0.0157,-0.0054,
0.1592,0.0043,0.0144,0.0003,-0.0052,-0.0004,
0.1842,1.1464,-0.4583,-0.3240,6.6091,-0.0976,
0.6473,0.1619,-6.2459,0.0357,-0.1222,-0.0987};

float bias_HL[NUM_NEURONS_HL]={
-0.5651,14.6281,-27.2550,6.7352,2.3888,305.0618,
-115.8594,-4.2389,2.4989,52.2467,31.3592,14.6253
};

float bias_OL[NUM_NEURONS_OL]={-0.0482,-1.0261};
```

De forma análoga às redes neuronais implementadas anteriormente, cada um dos vetores apresentados anteriormente refere-se às caraterísticas da rede neuronal artificial obtidas pelo treino, pesos das ligações e polaridades dos neurónios. Nesta rede neuronal, em comparação com a rede neuronal anterior, o número de neurónios da camada escondida passou de 10 para 12, logo o vetor que se refere aos pesos das ligações de entrada, `weights_HL`, tem um tamanho superior, 48 valores (4 entradas vezes 12 neurónios), o vetor que se refere às polarizações dos neurónios da camada escondida, `biass_HL`, ficou com um tamanho de 12 valores e o vetor referente às entradas dos neurónios da camada de saída (saídas dos

neurónios da camada escondida), `weights_OL`, passou a conter 24 valores (12 entradas vezes 2 neurónios).

Com a implementação desta rede neuronal multicamada proactiva 4-12-2 obtiveram-se os resultados apresentados na Figura 74 e Figura 75, que representa o AGV a efetuar o trajeto treinado apresentado na secção 4.5. Nos gráficos da Figura 74 é apresentada a evolução das variáveis de entrada *S*1 e *S*3 da RNA implementada no AGV, enquanto nos gráficos da Figura 75 é apresentada a evolução das variáveis de entrada *EO* e *ED* da mesma RNA.

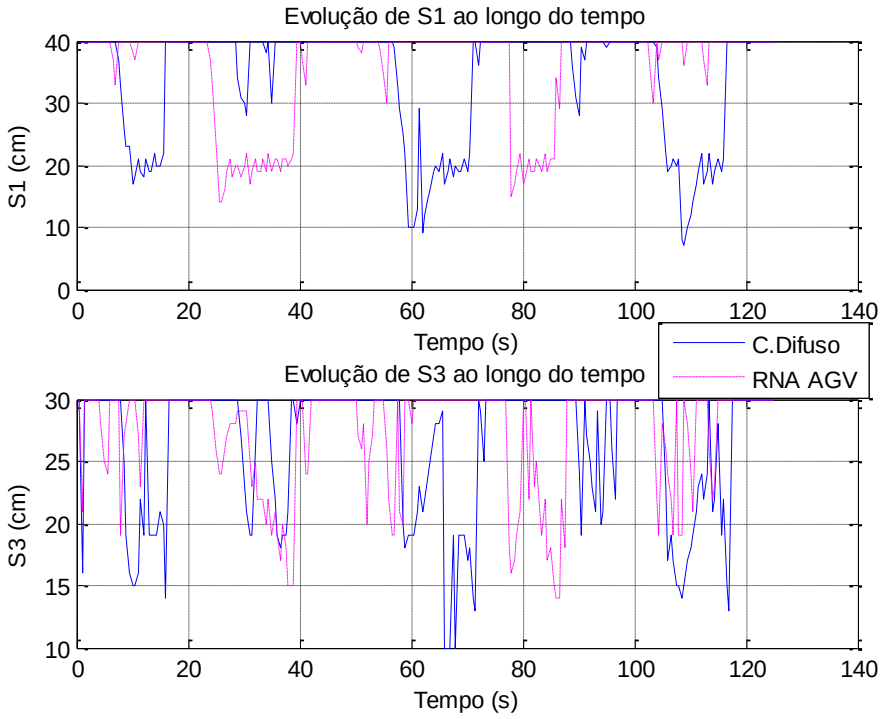


Figura 74 Resultados experimentais obtidos com a RNA que implementa o seguimento de uma parede com descontinuidades e o desvio de obstáculos: sensores de deteção de obstáculos

Ao analisar os gráficos apresentados anteriormente verificou-se uma grande discrepância na sua localização temporal, entre os valores do treino (traço azul) e os valores obtidos experimentalmente (traço magenta) sem causa que se conseguisse apontar imediatamente. Mas os comportamentos que se pretende que o AGV apresente são observados nos gráficos das Figura 74 e Figura 75, pois as variações nas medições dos sensores e dos erros são semelhantes tanto no controlador difuso como no controlo através da rede neuronal, apenas

deslocados no espaço temporal. Uma possível justificação para isto ter acontecido, é em todos os resultados experimentais apresentados o número de amostras de ambos os controlos, lógica difusa e rede neuronal, serem iguais, o que não se verifica neste caso, podendo ter havido alguma irregularidade na obtenção destes dados experimentais o que causou a falta de sincronismo visível nos gráficos.

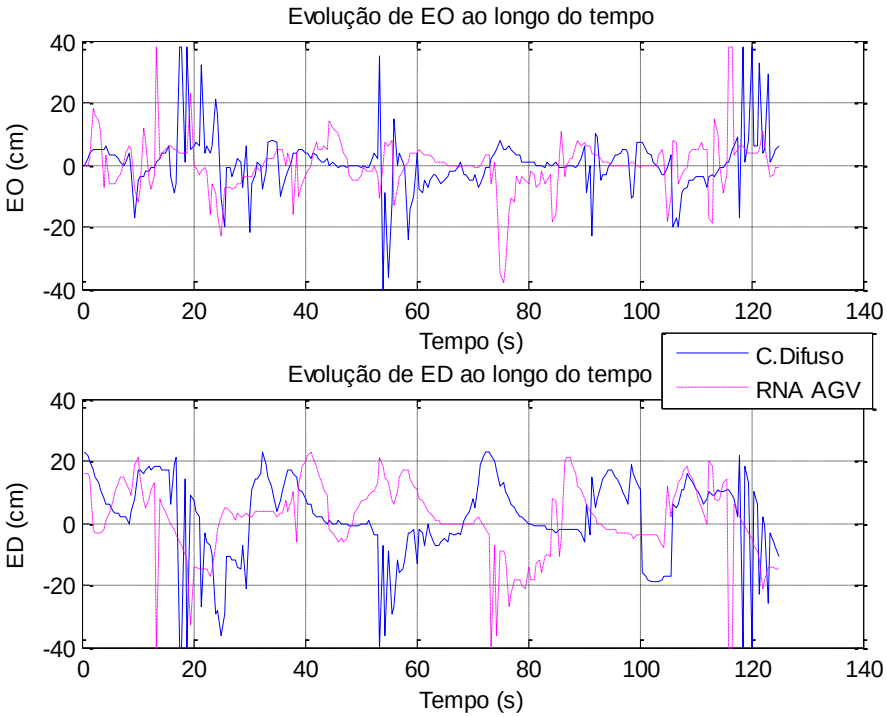


Figura 75 Resultados experimentais obtidos com a RNA que implementa o seguimento de uma parede com descontinuidades e o desvio de obstáculos: erros calculados

Para verificar se esta discrepância entre os dados utilizados no treino e os dados experimentais era pontual foi novamente amostrado o sistema difuso e o sistema neuronal a efetuar o mesmo trajeto que anteriormente. Os resultados deste teste são apresentados nas Figura 76 e Figura 77 através da evolução das variáveis de entrada da RNA. A evolução das variáveis de entrada *S*1 e *S*3, é apresentada nos gráficos da Figura 76, enquanto na Figura 77 são apresentados os gráficos da evolução das variáveis de entrada *EO* e *ED* da RNA, ao longo do tempo amostrado no teste.

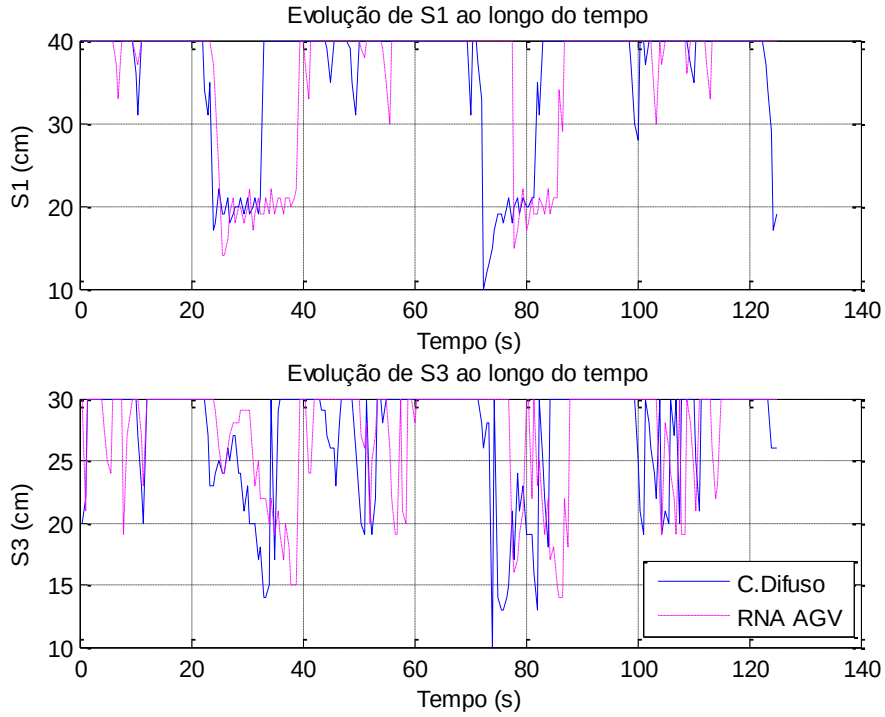


Figura 76 Resultados experimentais obtidos com a RNA e com o controlador difuso implementados no AGV a efetuar o trajeto de treino: sensores de deteção de obstáculos

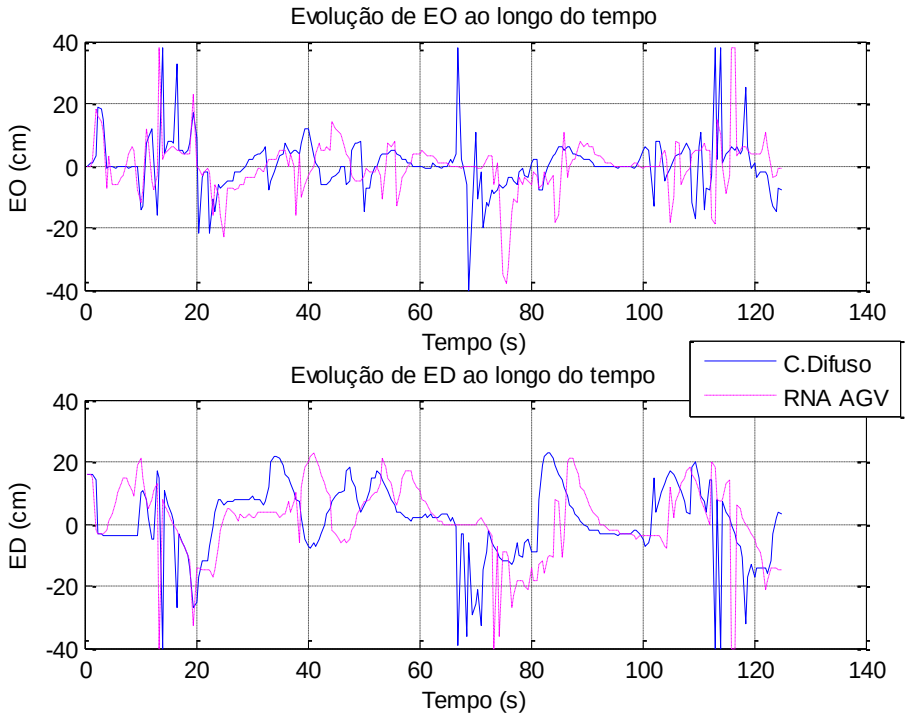


Figura 77 Resultados experimentais obtidos com a RNA e com o controlador difuso implementados no AGV a efetuar o trajeto de treino: erros calculados

Ao analisar estes resultados pode-se verificar que a discrepância temporal que existia nos resultados anteriores não se verificou e que a resposta do sistema quando controlado por lógica difusa ou pela rede neuronal é semelhante. Neste caso o número de amostras é igual em ambos os controlos.

5.2. PERCURSO ALEATÓRIO

Para validar a utilização deste sistema de controlo em percursos não treinados, foi testado o funcionamento do AGV a percorrer o seguinte trajeto apresentado na Figura 78.

a)

b)

c)

Figura 78 Trajeto percorrido pelo AGV neste teste: a) secção inicial do percurso, b) secção intermédia do percurso e c) secção final do percurso

O trajeto apresentado na Figura 78 é um percurso complexo, sendo que o AGV ao percorrê-lo executa todos os comportamentos anteriormente treinados, na secção 4.5, o qual se

verificou que funcionava satisfatoriamente no percurso de treino, como descrito na secção 5.1.3. Assim o teste efetuado consistia em colocar o AGV, controlado com a RNA implementada na secção 5.1.3, a percorrer este percurso.

Os resultados obtidos no teste são apresentados na Figura 79 e Figura 80, que representam o percurso efetuado pelo AGV. Na Figura 79 é apresentada a evolução das variáveis de entrada *S*1 e *S*3, que demonstram a deteção de obstáculos no percurso, enquanto na Figura 80 é apresentada a evolução das variáveis de entrada *EO* e *ED*, que mostram os erros calculados em relação à parede que o AGV segue no seu percurso.

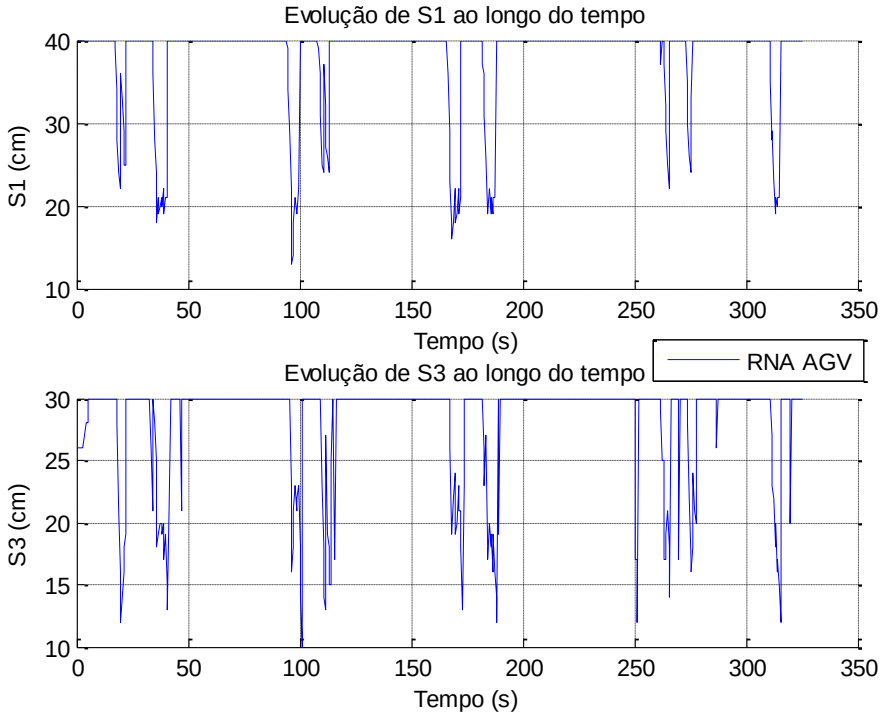


Figura 79 Resultados experimentais obtidos com a RNA implementada no AGV a efetuar o trajeto aleatório: sensores de deteção de obstáculos

Como se pode verificar pela análise dos gráficos apresentados na Figura 79 e Figura 80, o trajeto realizado pelo AGV ao efetuar o percurso aleatório foi satisfatório, pois não se aproximou excessivamente dos obstáculos que se encontravam no seu caminho, deixando sempre uma distância mínima de 10 cm para com o obstáculo ao se afastar pela sua esquerda, como se pode ver pela análise do gráfico do sensor *S*3 apresentado na Figura 79. Nas secções em que o AGV tinha uma parede para seguir, a distância a que este se colocava da parede era a distância de referência de 30 cm, visível no gráfico do Erro de Distância (*ED*) da Figura 80 pelos valores nulo de erro que são apresentados.

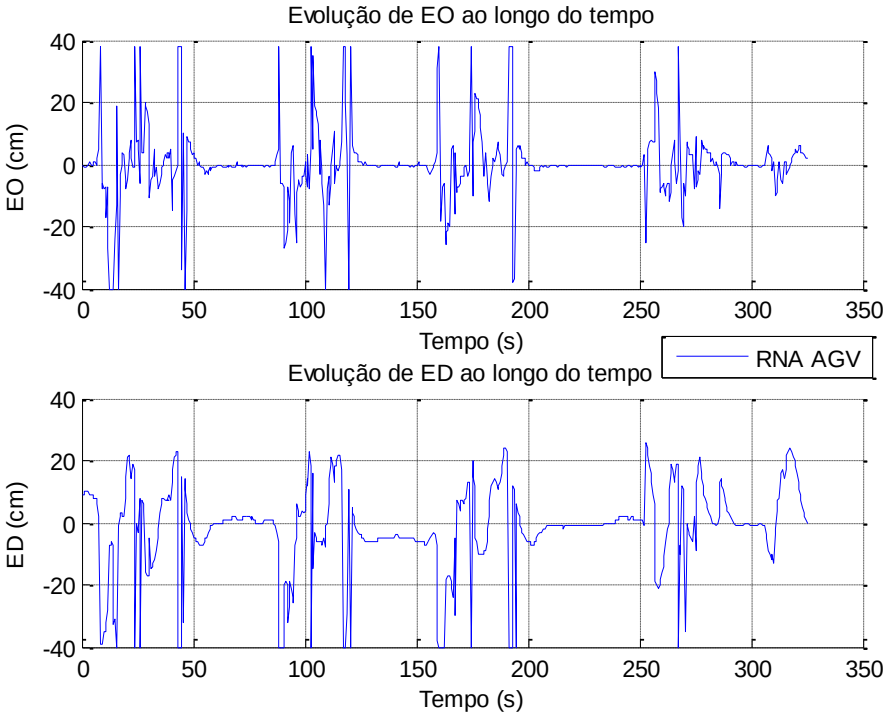


Figura 80 Resultados experimentais obtidos com a RNA implementada no AGV a efetuar o trajeto aleatório: erros calculados

Além deste percurso, também foi colocado o AGV a percorrer outros percursos aleatórios, em que se verificou que o funcionamento do AGV, para estes casos, foi igualmente satisfatório.

5.3. CRIAÇÃO, TREINO E IMPLEMENTAÇÃO DE UMA REDE NEURONAL A PARTIR DE DADOS RETIRADOS DO CONTROLO REMOTO

Através do controlo remoto disponibilizado no AGV e implementado na aplicação Android do *smartphone*, apresentada na subsecção 3.8.4 do Capítulo 3, foi implementado um sistema que permite efetuar o treino do robô com os dados do AGV (as medições de distância efetuadas pelos sonares e velocidades angulares de referência para os motores) quando este percorre um trajeto controlado através do controlo remoto. O objetivo final desta implementação era que o sistema controlado autonomamente através de uma RNA se comportasse de forma semelhante ao sistema quando controlado remotamente, percorrendo assim nos dois casos um trajeto semelhante.

Como nas outras implementações efetuadas, a rede neuronal é uma rede multicamada proactiva com quatro entradas (valores medidos pelos sonares) e duas saídas (valores das

velocidades angulares de referência para cada um dos motores) como é apresentado na Figura 81.

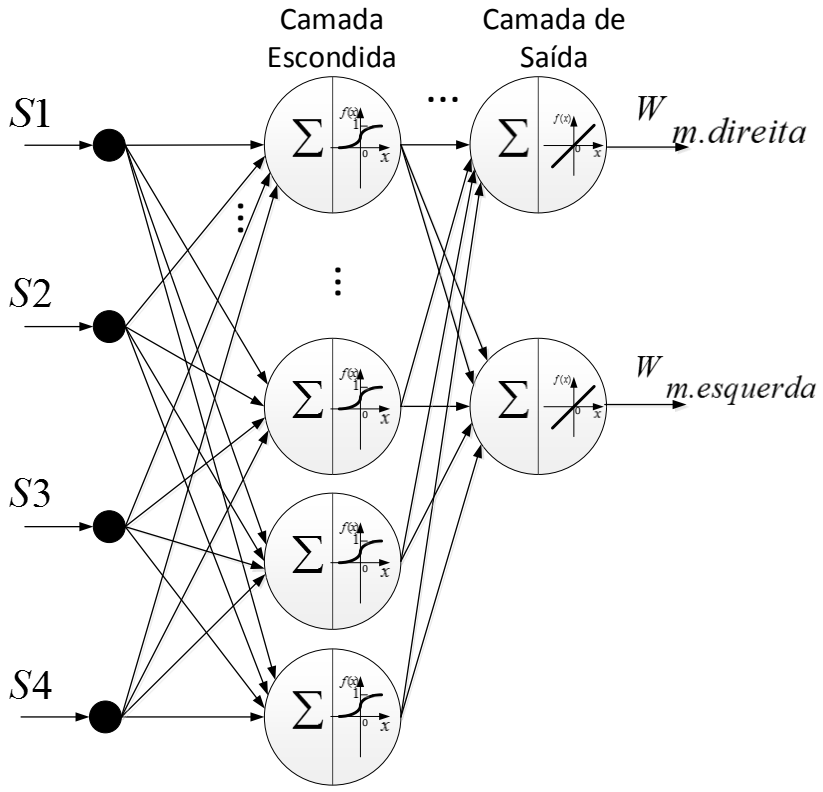


Figura 81 Rede neuronal criada para se comportar como o controlo remoto amostrado

Assim, para obter os dados de treino para esta rede neuronal utilizou-se o AGV no modo controlado remotamente, efetuando percursos com a mesma complexidade do utilizado na secção 4.5. Para retirar as amostras do AGV quando controlado remotamente, efetuaram-se alterações ao programa que implementa a RNA e ao controlo remoto no AGV, adicionando-se código que efetua as amostras das medições efetuadas pelos sensores e dos valores de referência de velocidades angulares enviadas para o controlador PI. Esta rotina é igual à rotina apresentada no fluxograma mostrado na Figura 38 (Capítulo 4), à exceção que ao invés da execução do controlador difuso é efetuada a descodificação do comando recebido via Bluetooth.

A obtenção do resultado pretendido desta experiência, ou seja, o robô a funcionar de forma satisfatória no modo autónomo com uma rede neuronal treinada a partir do controlo remoto, apenas era possível atingir através do método de tentativa e erro, isto é, controlava-se remotamente o AGV a percorrer um trajeto, retirando amostras do funcionamento (medições dos sonares e velocidades angulares aplicadas aos motores). Posteriormente, estas amostras

eram utilizadas no treino de uma rede neuronal do tipo apresentado na Figura 81, obtendo-se os pesos e as polarizações que caraterizam a rede neuronal treinada. Por fim verificava-se se a rede neuronal treinada cumpria os requisitos necessários efetuando um teste experimental no AGV. Se o resultado do teste experimental não fosse satisfatório, tornava-se a efetuar os processos anteriormente referidos. O resultado do teste era satisfatório se o AGV percorresse o percurso treinado de uma forma que se observasse semelhante à treinada, mantendo uma distância (por exemplo 20 cm) à parede que segue e se desviasse de obstáculos que se encontrem no seu percurso. Posteriormente testava-se o AGV controlado com essa rede neuronal no percurso aleatório utilizado anteriormente na secção 5.2.

Inicialmente retiraram-se amostras do percurso a ser percorrido pelo AGV, apresentado na Figura 60 até à Figura 62, controlado remotamente para utilizar no treino da rede neuronal. Estes dados criavam uma rede neuronal que apresentava um funcionamento satisfatório quando o percurso que o AGV tinha de percorrer era o mesmo que o treinado; no percurso aleatório isto já não se verificava. Por isso foi necessário testar outras soluções que permitissem desenvolver uma rede neuronal que funcionasse de forma satisfatória no percurso aleatório.

5.3.1. REDE NEURONAL TREINADA COM DADOS AMOSTRADOS DO AGV CONTROLADO REMOTAMENTE A PERCORRER OS 3 PERCURSOS DIFERENTES

De forma a resolver o problema anterior, do AGV não percorrer satisfatoriamente o percurso aleatório quando controlado com uma RNA treinada com dados amostrados do robô a efetuar o percurso apresentado na Figura 60 até à Figura 62, foi realizado um teste experimental composto por três percursos diferentes, em que um destes três percursos era o percurso apresentado nessas figuras. Os dados amostrados do AGV a percorrer os três percursos são apresentados na Figura 82, Figura 83 e Figura 84 e estes foram utilizados para o treino da RNA. Tanto os dados de entrada como os dados de saída que foram amostrados e que são apresentados na Figura 82, Figura 83 e Figura 84 correspondem aos três percursos diferentes que foram utilizados no treino desta nova RNA. Como se verifica nos gráficos, não se distinguem os três percursos porque quando se efetuou a amostragem não se teve isso em atenção, tendo-se efetuado esta de forma continua. Além disso não são apresentadas fotografias dos outros dois percursos, além do já apresentado na secção 4.5, porque na ocasião em que os percursos foram amostrados, estes também foram criados. Os percursos eram as duas paredes de um corredor onde se colocaram obstáculos, uma caixa era colocada

aleatoriamente no caminho do AGV para este se ter de desviar. Na Figura 82 é apresentada a evolução das variáveis de entrada *S*1 e *S*3 ao longo do tempo amostrado.

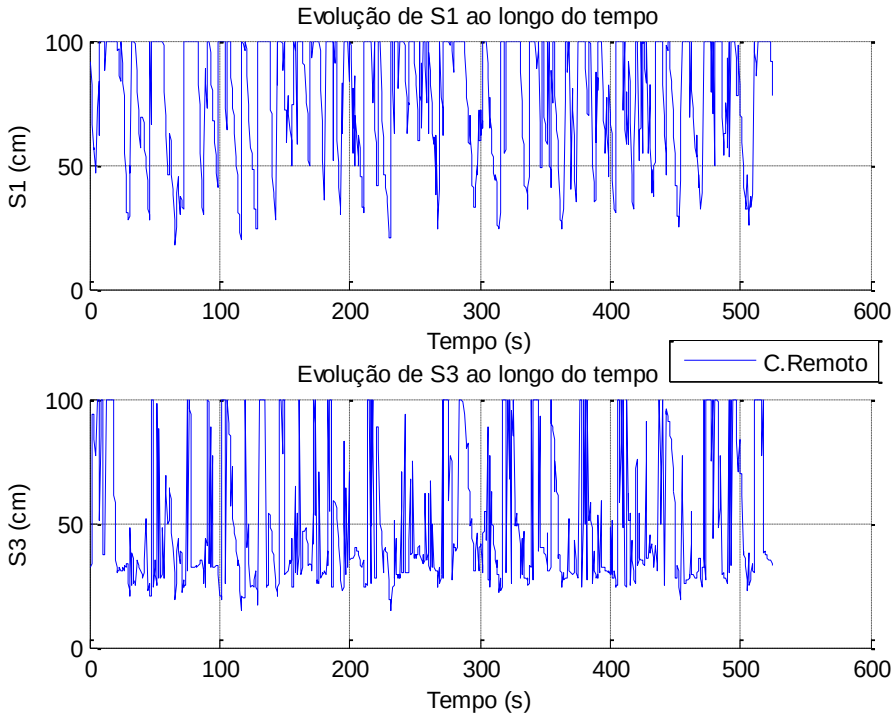


Figura 82 Dados de entrada para o treino da RNA amostrados do controlo remoto enquanto o AGV percorre os 3 trajetos diferentes: *S*1 e *S*3

Como se pode verificar pela análise da Figura 82 e Figura 83, os valores máximos das medições dos sensores foram limitados a 100 cm de forma a que se tenha um conjunto de treino limitado para que os pares de entradas-saídas utilizados no treino o representem de forma satisfatória. Se este conjunto não for representado de forma satisfatória, os valores de saída poderão não ser estimados satisfatoriamente quando na entrada se tem um conjunto de valores não treinados. Um exemplo deste problema era se o conjunto de entrada máximo medido pelos sensores [*S*1 *S*2 *S*3 *S*4]=[100 100 100 100] cm, não for amostrado, ficando com um valor definido na saída, como por exemplo [*Wm.direito Wm.esquerdo*]=[5 5], ao se treinar a rede, o valor de saída resultante poderia ultrapassar a velocidade angular máxima do motor que é de 7 rad/s. A evolução das variáveis de entrada *S*2 e *S*4 ao longo do tempo amostrado é apresentada na Figura 83. Na Figura 84 é mostrada a evolução do valor de referência da velocidade para cada um dos motores do AGV e que foram utilizadas como *targets* no treino da RNA.

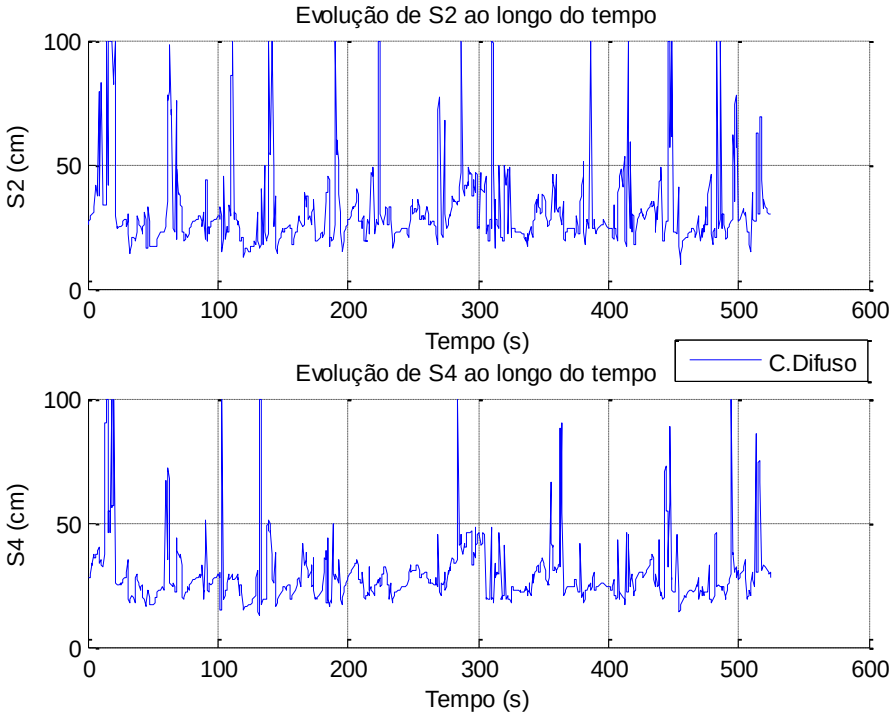


Figura 83 Dados de entrada para o treino da RNA amostrados do controlo remoto enquanto o AGV percorre os 3 trajetos diferentes: *S2* e *S4*

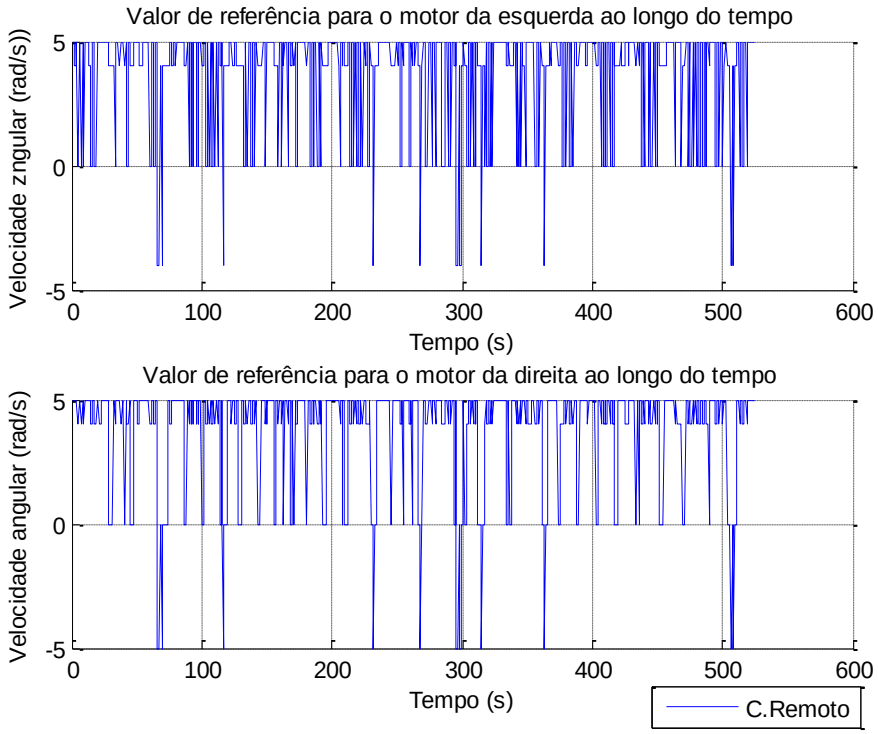


Figura 84 Dados de saída para o treino da RNA amostrados do controlo remoto enquanto o AGV percorre os 3 percursos diferentes

Como é visível nos gráficos da Figura 84, os valores de saída, que são os objetivos (*targets*) da rede neuronal, estão divididos em valores definidos, como apresentados na Tabela 10 (Capítulo 3), para cada um dos motores do AGV.

De uma forma semelhante aos treinos anteriores utilizou-se o seguinte código no MATLAB para efetuar a criação, treino e simulação da rede neuronal.

```
remote_data=load('remote_samples_phone_3_trajetos
diferentes.txt');
T=0.5;

%% Defining values to the Network
entradas=[remote_data(:,2) remote_data(:,3)
remote_data(:,4) remote_data(:,5)];
saidas_desejadas=[remote_data(:,6)
remote_data(:,7)];

inputs=entradas';
targets=saidas_desejadas';
%% Create a Neural Network
rede = newff(minmax(inputs),[12 2],
{'logsig' 'purelin'});
rede.initFcn
rede.layers{1}.initFcn
rede.layers{2}.initFcn

%% Initialize weights
rede=init(rede)
pesosIL = rede.IW{1}
pesosLW = rede.LW{2}
polaridade1 = rede.b{1}
polaridade2 = rede.b{2}

%% Train network
rede.trainParam.showCommandLine = 1;
rede.trainParam.show = 10;
rede.trainParam.lr = 0.01;
rede.trainParam.mc = 0.9;
rede.trainParam.epochs = 2000;
rede.trainParam.goal = 0;
[rede treino] = train(rede,inputs,targets);
PerformanceFinal=treino.perf(end)
pesosIL = rede.IW{1}
pesosLW = rede.LW{2}
polaridade1 = rede.b{1}
polaridade2 = rede.b{2}
pause
nntraintool('close')

%% Simulate the Network
saidas=sim(rede,inputs);
subplot(211);
```

```
final_point=length(remote_data(:,1));
time_var=1:1:final_point;
plot(time_var*T,targets(1,1:final_point),'k-');
grid on;hold on;
plot(time_var*T,saidas(1,1:final_point),'k--');
xlabel('Tempo(s)');ylabel('rad/s');
title('Velocidade Angular Motor da Direita ');
subplot(212);
plot(time_var*T,targets(2,1:final_point),'k-');
grid on;hold on;
plot(time_var*T,saidas(2,1:final_point),'k--');
xlabel('Tempo(s)');ylabel('rad/s');
title('Velocidade Angular Motor da Esquerda');
pause
```

Utilizando o código apresentado, que implementa uma rede proactiva 4-12-2 no MATLAB, obtiveram-se os seguintes resultados apresentados na Figura 85 e Figura 86. A traço azul são apresentados os valores das velocidades utilizadas como *targets* no treino e a traço magenta os valores das velocidades obtidas pela simulação da rede neuronal no MATLAB. Estas figuras mostram as comparações entre o treino e a simulação para a velocidade angular do motor da roda direita e esquerda do AGV, respetivamente.

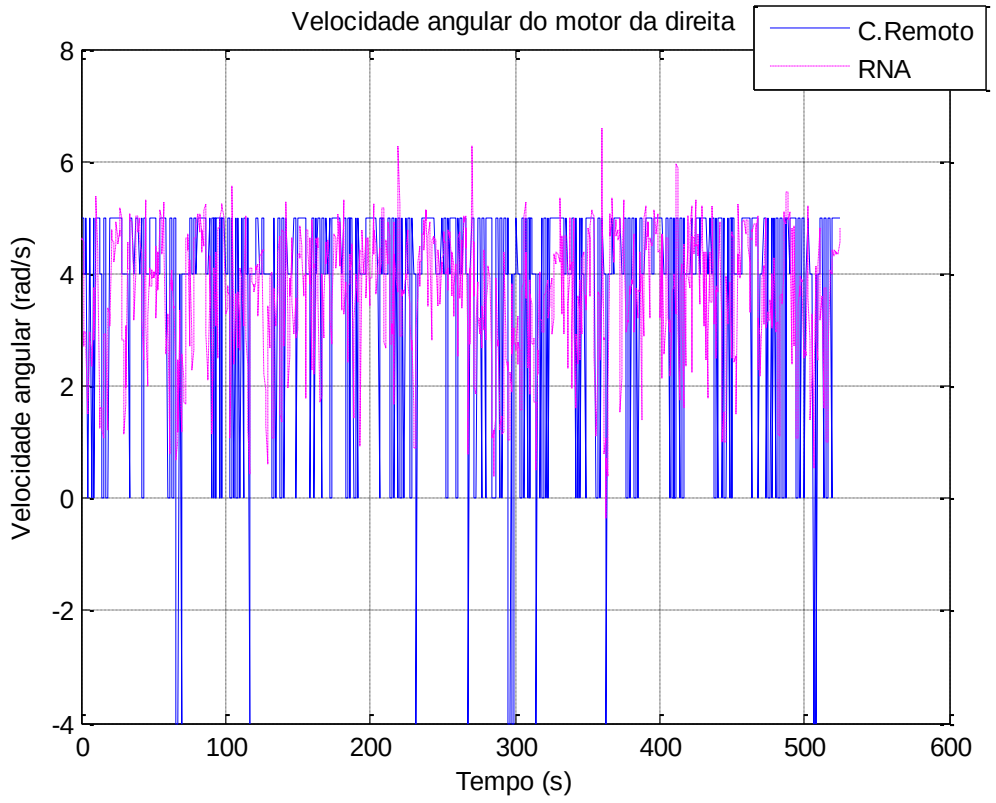


Figura 85 Resultados da simulação da RNA simulada no MATLAB quando a entrada são os valores de entrada utilizados no treino: velocidade angular do motor da direita

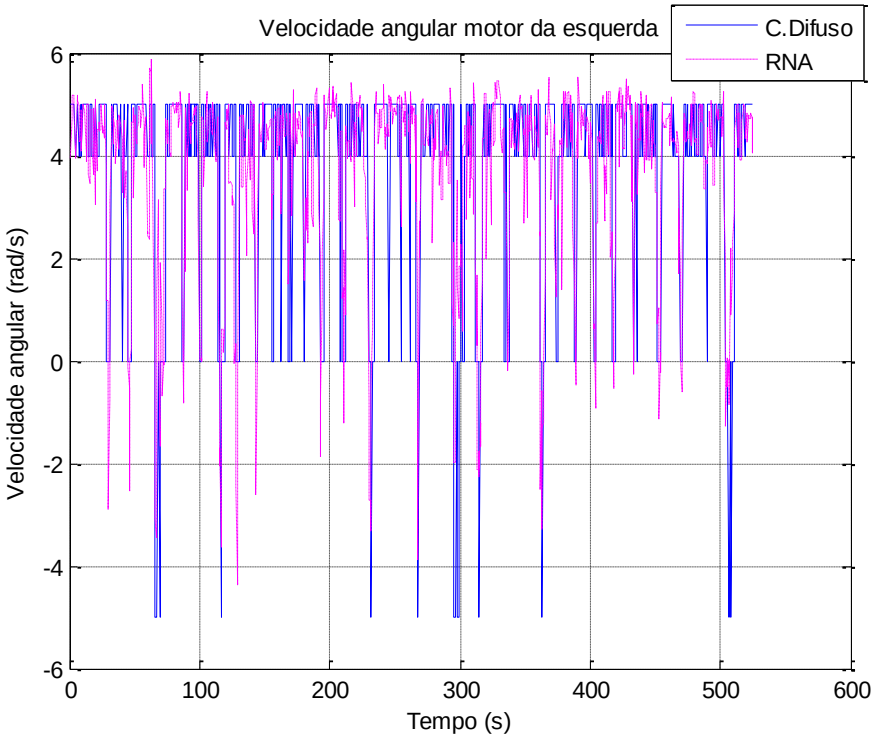


Figura 86 Resultados da simulação da RNA simulada no MATLAB quando a entrada são os valores de entrada utilizados no treino: velocidade angular do motor da esquerda

O valor do desempenho desta rede neuronal treinada é de *MSE* = 2,7345, o que significa que os valores de saída da rede neuronal são por regra diferentes dos valores dos *targets* utilizados no treino. Isto é visível na Figura 87, onde são apresentados os primeiros 200 valores obtidos da simulação da rede neuronal (traço magenta) e os respetivos primeiros 200 valores utilizados no treino como *targets* (traço azul). Nesta figura verifica-se que os valores que a rede neuronal apresenta na sua saída não são os valores definidos utilizados no treino. Esta discrepância de valores era esperada e pretendida, pois assim o AGV irá apresentar um funcionamento mais suave nas variações das velocidades do que teria se os valores de saída da rede neuronal fossem exatamente os utilizados como *targets.*

A implementação desta rede neuronal no AGV apresenta um funcionamento ótimo no percurso treinado, mas no percurso aleatório apresenta uma situação em que reage de uma forma não pretendida. Na zona apresentada na Figura 88, que pertence ao percurso aleatório, o AGV controlado autonomamente pela rede neuronal treinada através de dados amostrados do controlo remoto, na esquina (destacada pelo círculo vermelho) não se desviava da parede e posteriormente aproximava-se desta até ir de encontro a ela.

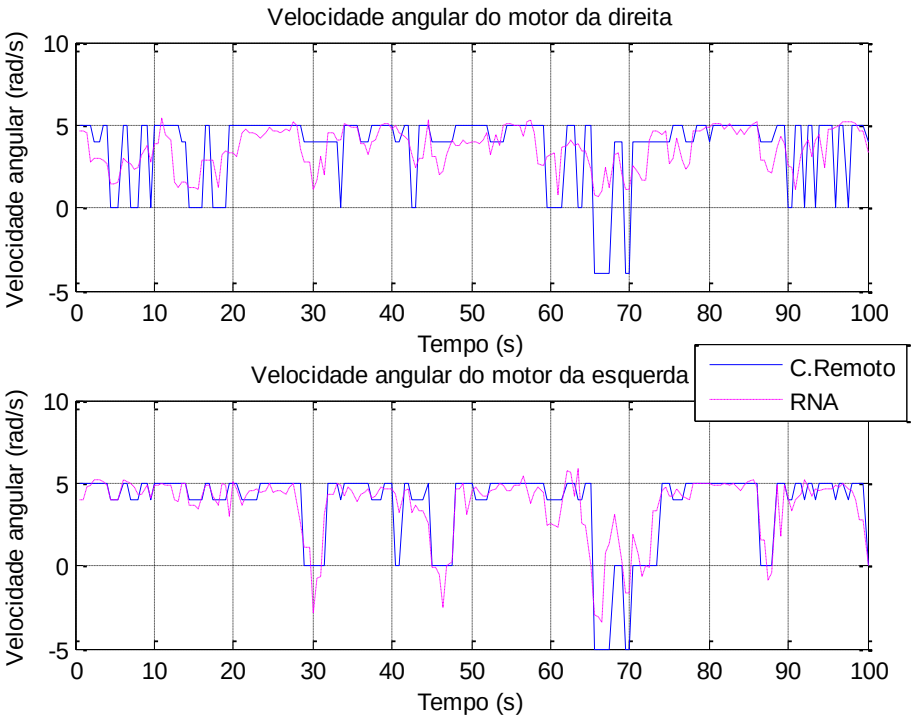


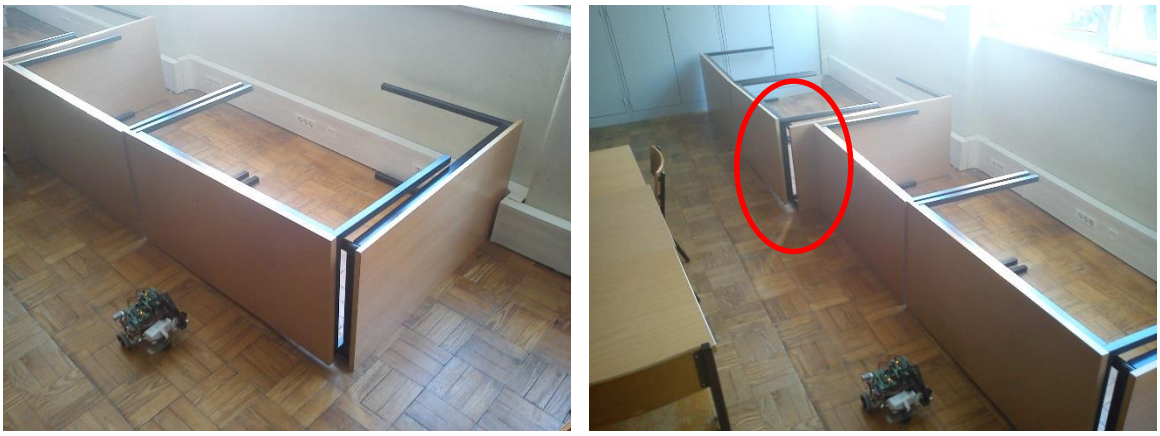
Figura 87 Resultados da simulação da RNA simulada no MATLAB quando a entrada são os 200 primeiros valores de entrada utilizados no treino

De forma a solucionar este problema resolveu-se criar um percurso que tivesse uma zona com um problema semelhante a este para tentar resolvê-lo através da adição dos dados desse percurso ao dados utilizados no treino desta RNA.

Figura 88 Zona problemática do percurso aleatório

5.3.2. REDE NEURONAL TREINADA COM DADOS AMOSTRADOS DO AGV CONTROLADO REMOTAMENTE A PERCORRER OS 3 PERCURSOS DIFERENTES E O PERCURSO CRIADO PARA RESOLVER O PROBLEMA ENCONTRADO ANTERIORMENTE

O percurso criado para resolver o problema anterior é apresentado na Figura 89, destacando-se com um círculo vermelho a secção que se pretende que simule a zona problemática. Os dados amostrados do AGV controlado remotamente a percorrer este percurso foram adicionados a um ficheiro que contém também as amostras utilizadas na secção 5.3.1.



a) **b)**

Figura 89 Percurso criado para treinar a rede neuronal para resolver o problema encontrado: a) primeira secção e b) segunda secção

Na Figura 90, Figura 91 e Figura 92 são apresentados todos os dados amostrados que serão utilizados no treino desta rede neuronal. Comparando estes dados de entrada com os dados de entrada utilizados na secção anterior, nota-se que os dados deste percurso foram adicionados no início e no fim. O percurso amostrado gerou 150 amostras, ou seja, os primeiros e os últimos 75 segundos (a frequência de amostragem é de 2 Hz) apresentados nos gráficos dos valores de entrada e de saída correspondem ao novo percurso.

Na Figura 90 são mostrados os dados que se referem à evolução de $S1$ e $S3$ durante o tempo amostrado, enquanto os dados referentes à evolução de $S2$ e $S4$ durante o tempo amostrado são apresentados na Figura 91. Na Figura 92 são apresentadas as velocidades dos motores do AGV, que são os *targets* do treino da RNA, correspondentes às entradas amostradas.

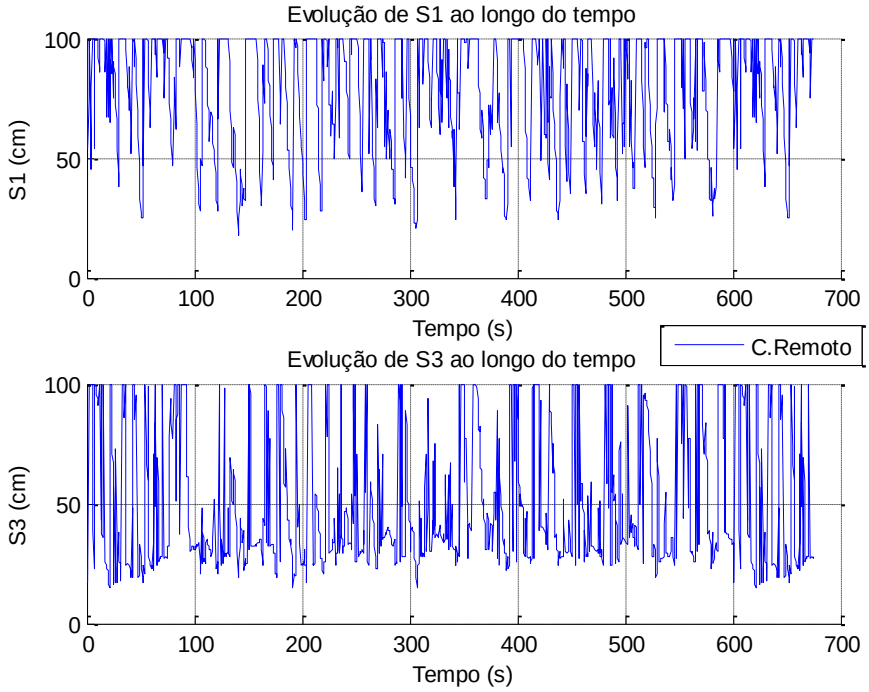


Figura 90 Dados de entrada para o treino da RNA amostrados do controlo remoto enquanto o AGV percorre os 3 percursos diferentes e o percurso criado para resolver o problema encontrado anteriormente: *S*1 e *S*3

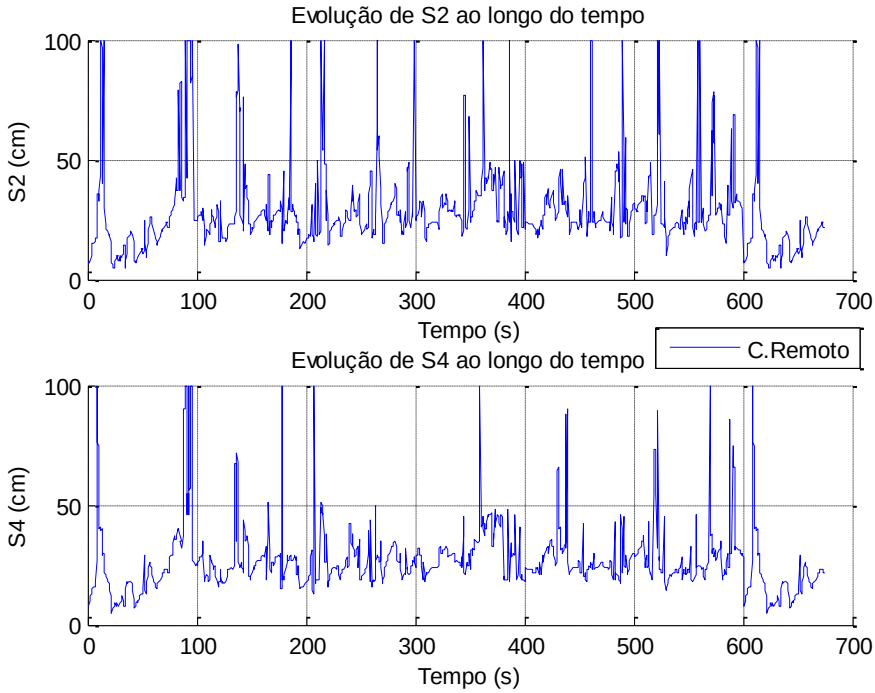


Figura 91 Dados de entrada para o treino da RNA amostrados do controlo remoto enquanto o AGV percorre os 3 percursos diferentes e o percurso criado para resolver o problema encontrado anteriormente: *S*2 e *S*4

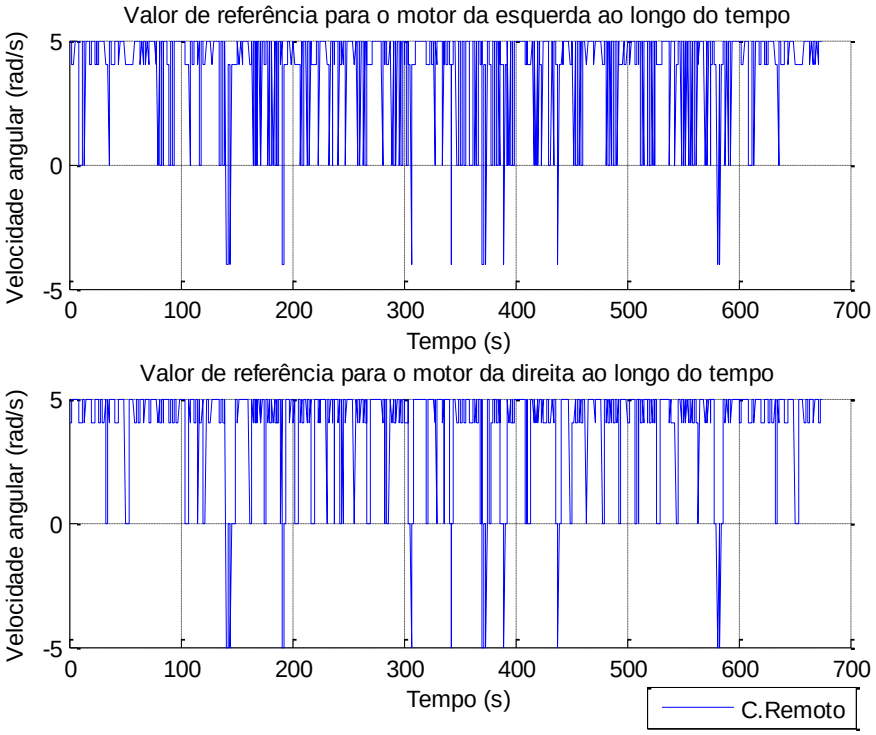


Figura 92 Dados de saída para o treino da RNA amostrados do controlo remoto enquanto o AGV percorre os 3 percursos diferentes e o percurso criado para resolver o problema encontrado anteriormente

Foi efetuado o treino da rede neuronal no MATLAB, com os dados anteriormente apresentados, e foram obtidos os pesos e polarizações que caraterizam a rede neuronal implementada no AGV que são apresentados no extrato de código seguinte.

```
float weights_HL[NUM_INPUTS_HL*NUM_NEURONS_HL]={
-2.9121,0.84788,0.056948,3.4179,0.72979,
0.050392,-0.21149,-1.6562,-0.080192,-0.013085,
-0.071762,0.050985,-0.056079,0.0081359,-0.37068,
0.34931,-0.019776,-0.13182,-0.094899,0.19092,
3.9292,-0.16593,-0.093486,-3.9195,0.22183,
0.72771,-0.0098971,-1.1569,-0.043846,-0.24969,
-2.4264,2.7101,-0.1306,-0.0018774,0.0037989,
0.040957,0.067348,-0.17465,-0.17416,0.96308,
2.567,-0.7591,-2.8622,0.78869,0.0080982,
-0.8986,0.88922,0.29351
};

float weights_OL[NUM_INPUTS_OL*NUM_NEURONS_OL]={
0.14408,1.2045,2.791,3.2888,-1.417,0.58608,
1.2846,-1.4898,-3.446,-1.8137,0.22306,1.9166,
1.2258,-0.064503,1.7337,0.49899,-2.1157,-2.591,
0.15077,-2.6567,-9.3208,1.5752,0.59577,1.7599};

float bias_HL[NUM_NEURONS_HL]={
```

```
-17.7,12.318,11.195,5.6865,6.3986,11.296,
-8.7336,7.076,2.7902,3.5706,1.5786,-8.6886};

float bias_OL[NUM_NEURONS_OL]={1.3423,3.7195};
```

O teste experimental do funcionamento do AGV controlado por esta RNA foi efetuado no percurso aleatório apresentado na secção 5.2. O teste foi realizado em duas secções deste percurso. A primeira secção é apresentada na Figura 78a) e a segunda secção é apresentada na Figura 78b) e Figura 78c). Na Figura 93 são mostrados os resultados da primeira secção através das variáveis de entrada *S*1 e *S*3, enquanto na Figura 94 são apresentados os resultados da primeira secção através das variáveis de entrada *S*2 e *S*4.

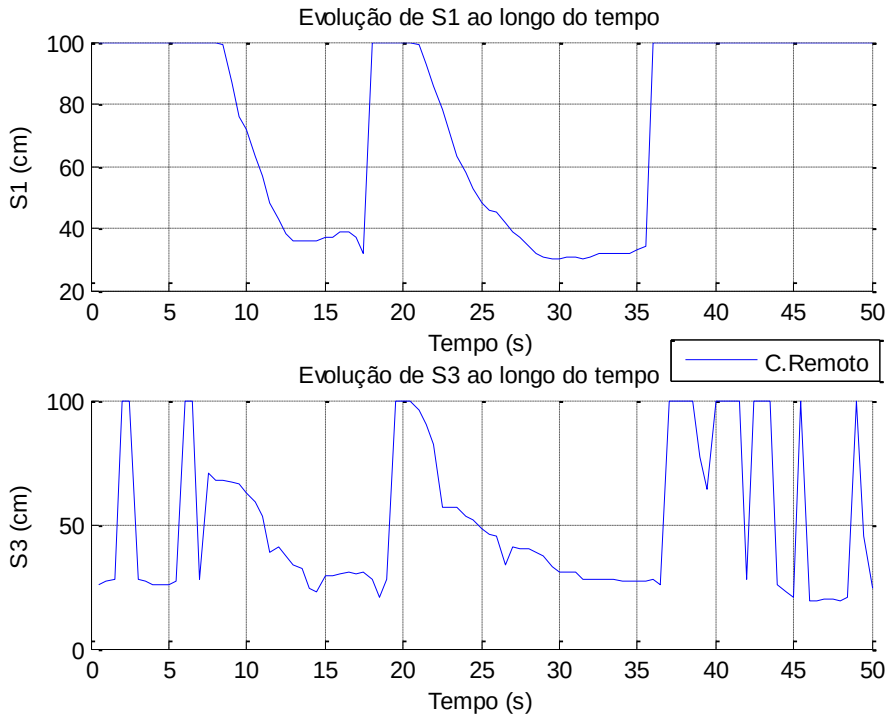


Figura 93 Resultados experimentais do AGV a percorrer a primeira secção do percurso aleatório: entradas *S*1 e *S*3

Como se pode verificar pela análise dos gráficos da Figura 93, o AGV nunca se aproximou demasiado de um obstáculo, mantendo uma distância mínima de pelo menos 20 cm.

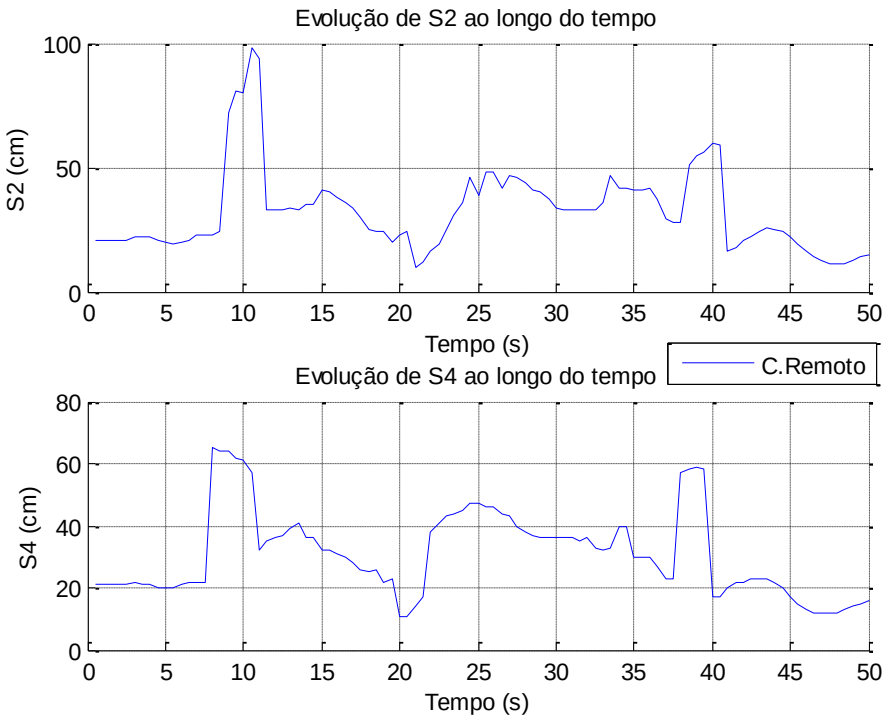


Figura 94 Resultados experimentais do AGV a percorrer a primeira secção do percurso aleatório: entradas *S*2 e *S*4

Analisando os gráficos da Figura 94 verifica-se que no início e no fim desta secção amostrada, o AGV mantinha uma distância de 20 cm à parede que seguia. Na zona intermédia, em que este se desviava de obstáculos no seu percurso, a distância mantida à parede diminuía no máximo para 10 cm.

Os resultados obtidos pelo AGV a percorrer a segunda secção são apresentados na Figura 95 através das variáveis de entrada *S*1 e *S*3 e na Figura 96 através das variáveis de entrada *S*2 e *S*4.

Os resultados da segunda secção são semelhantes aos resultados da primeira, no que se refere à distância a obstáculos - a distância a um obstáculo foi de pelo menos 20 cm (Figura 95). Já no que se refere à distância mantida à parede que o AGV segue - este manteve uma distância de 7 cm, que é menor do que no caso anterior. Também a média da distância mantida desceu de pelo menos 30 cm para 20 cm. Como se verifica pela análise dos gráficos da Figura 96, ocorrem variações da distância mantida à parede que o AGV segue, apesar de esta ser uma parede contínua. Isto foi causado pelos dados utilizados no treino também apresentarem estas variações, isto é, os dados amostrados, durante o controlo remoto do AGV, também variavam a distância mantida à parede, mesmo sendo esta contínua.

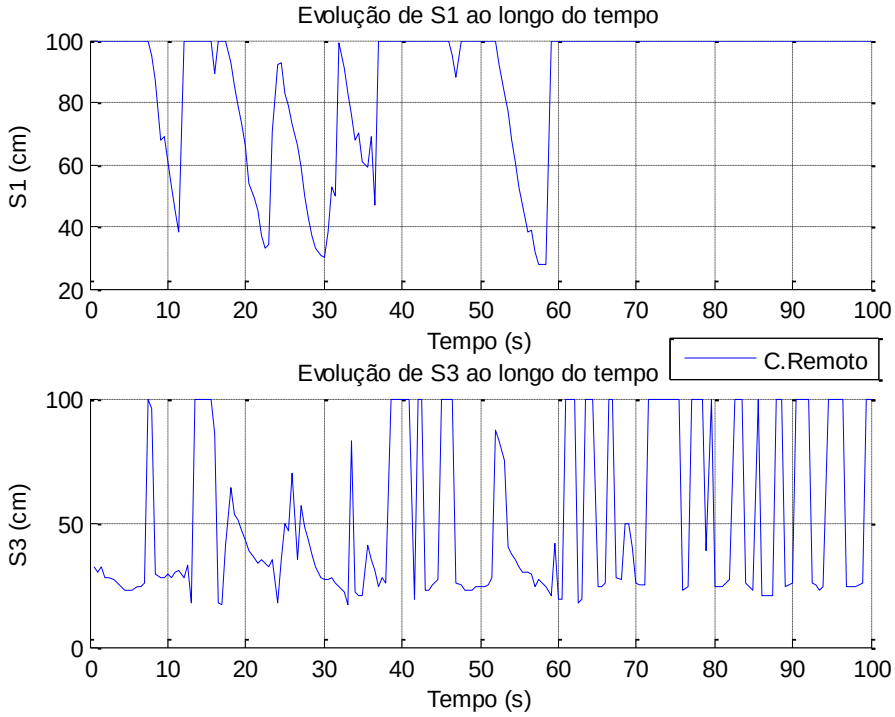


Figura 95 Resultados experimentais do AGV a percorrer a segunda secção do percurso aleatório: entradas *S*1 e *S*3

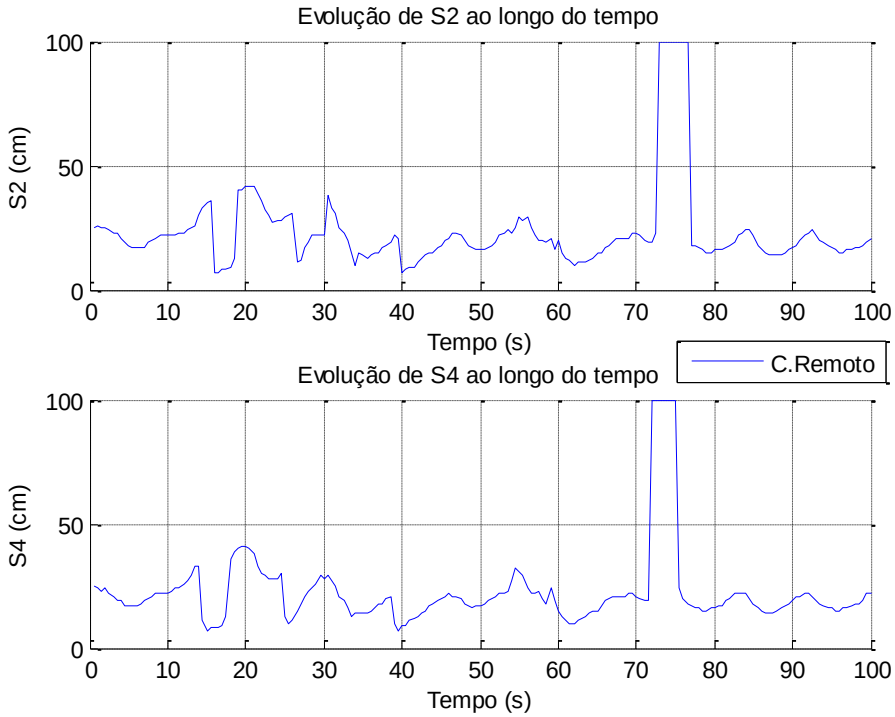


Figura 96 Resultados experimentais do AGV a percorrer a segunda secção do percurso aleatório: entradas *S*2 e *S*4

Apesar de alguns aspetos referidos na análise dos resultados apresentados anteriormente, conclui-se que se obteve um funcionamento satisfatório do AGV controlado pela rede neuronal no percurso aleatório.

5.4. DISCUSSÃO E CONCLUSÕES DOS RESULTADOS OBTIDOS

Nesta seção são referidas algumas das conclusões que se podem retirar das implementações efetuadas.

Analisando os resultados dos testes experimentais relatados, pode-se concluir que a implementação da rede neuronal a partir do controlo por lógica difusa que anteriormente estava implementado no AGV foi um sucesso.

Através do MATLAB foram criadas e treinadas redes neuronais artificiais que posteriormente se implementaram no AGV disponibilizado para realizar testes experimentais, que em geral foram muito satisfatórios.

Como fator de comparação do desempenho entre a RNA e o controlador difuso foi selecionado a soma dos erros quadráticos das variáveis, *ED* e *EO*, através das seguintes equações:

$$S_{EO} = \sum_{i=1}^{N} {EO_i}^2 \tag{30}$$

$$S_{ED} = \sum_{i=1}^{N} {ED_i}^2 \tag{31}$$

em que *N* é o número de amostras do percurso efetuado.

Os resultados obtidos são apresentados na Tabela 12, para cada um dos comportamentos implementados. Pela análise dos resultados obtidos, conclui-se que a RNA apresenta valores menores de erro em todos os comportamentos, exceto no comportamento de seguir uma parede com descontinuidades, no qual o valor de S_{EO} é menor no controlador difuso do que na RNA. Apesar disso pode-se considerar que o desempenho da RNA, em geral, é melhor que o desempenho do controlador difuso nos percursos efetuados.

Tabela 12 Comparação do desempenho entre a RNA e o controlador difuso

Comportamento	Controlo			
	Difuso		RNA	
	S_{EO}	S_{ED}	S_{EO}	S_{ED}
Seguir uma parede	0,0139	0,1256	0,0123	0,1021
Seguir uma parede com descontinuidades	0,1057	0,6554	0,1811	0,4650
Seguir uma parede com descontinuidades e desvio de obstáculos	1,9345	3,6527	1,8497	3,5684

Analisando os resultados dos testes experimentais relatados, pode-se concluir que a implementação da rede neuronal a partir do controlo remoto teve um desempenho satisfatório.

Comparando os resultados obtidos no percurso aleatório entre as duas abordagens implementadas, pode-se concluir que o funcionamento do AGV controlado pela RNA criada a partir do controlador difuso em comparação com a RNA criada a partir do controlo remoto apresenta um funcionamento melhor. Mas se existirem percursos onde ambas as abordagens não funcionem como esperado, a adaptação da RNA a partir do controlo remoto é muito mais simples do que a adaptação da RNA a partir do controlo difuso.

6. CONCLUSÕES

Ao longo deste texto foram sendo apresentadas conclusões que permitiram sustentar as opções de desenvolvimento efetuadas ao longo do projeto. Assim, neste último capítulo é efetuada uma síntese das principais conclusões, consequências e relevância do trabalho realizado e perspetivados futuros desenvolvimentos.

No MATLAB foram implementadas várias rotinas que permitem analisar os dados para criar, treinar e simular uma rede neuronal para ser implementada no AGV.

Efetuaram-se testes experimentais no AGV e validaram-se os resultados obtidos no MATLAB e no programa em linguagem C. Em termos de implementação foram abordadas as questões referentes à implementação de um controlo através de uma RNA que se pretendia que funcionasse como um controlador difuso previamente implementado e um controlo através de uma RNA que se comportasse como o trajeto efetuado através de um controlo remoto implementado no AGV.

O controlador difuso previa três comportamentos distintos, sendo eles: “Seguir Parede”, “Evitar Obstáculo” e “Emergência”. A existência de três comportamentos levou a uma implementação por fases, com a sua complexidade de implementação a aumentar, permitindo assim verificar as diferenças que o controlo pela RNA apresentava em comparação com o controlador difuso em cada uma das fases. Assim verificou-se que o

controlo neuronal era bastante semelhante ao controlador difuso nas duas primeiras fases: "Criação e treino de uma RNA que implementa o comportamento de seguir a parede" e "Criação e treino de uma RNA que implementa o comportamento de seguir uma parede com descontinuidades". No entanto, na terceira fase, "Criação e treino de uma RNA que implementa o seguimento de uma parede com descontinuidades e o desvio de obstáculos" já se verificou uma diferença mais acentuada entre o controlo neuronal e o controlador difuso. Apesar disto considera-se que o controlo neuronal funciona satisfatoriamente em todas as três fases pois o AGV funciona como pretendido, ou seja mantém uma distância fixa à parede que está a seguir, desvia-se dos obstáculos que se apresentem no seu percurso e no caso de não conseguir desviar-se, recua e posteriormente ultrapassa esse obstáculo.

A passagem da implementação em MATLAB para uma implementação em microcontrolador foi intercalada por uma implementação na linguagem de programação C no computador, permitindo assim uma plataforma de teste estática já bastante semelhante à plataforma final onde se pretendia implementar a rede neuronal criada. Assim conseguiu-se analisar o desempenho e a resposta da rede neuronal programada na linguagem C quando na entrada se tivesse os valores de treino utilizados no MATLAB. Como foi verificado, a resposta da rede neuronal programada na linguagem C era praticamente igual à resposta simulada no MATLAB.

Com os testes experimentais efetuados, desenvolveu-se código para o MATLAB, em linguagem C e para microcontrolador que permitiram validar a utilização das redes neuronais para controlo de um AGV tomando a decisão dos comportamentos que deve tomar com o objetivo de percorrer um percurso conhecido (treinado) ou desconhecido (não treinado).

A partir do conhecimento adquirido até agora com este estudo é possível utilizar este sistema e o código criado em MATLAB, linguagem C e microcontrolador para implementar e testar novas RNA no AGV disponibilizado.

A implementação da rede neuronal a partir do controlo remoto implementado permitiu verificar que através de um método de ensino simples se consegue obter um funcionamento complexo apesar de não ser perfeito. A maior dificuldade deste trabalho foi conseguir obter uma amostragem correta do trajeto pretendido na implementação da rede neuronal a partir do controlo remoto, pois como o controlo remoto implementado era simples, o seu desempenho não era o melhor.

O tempo de execução da rede neuronal implementada no AGV é, em média, 14 milissegundos na fase mais complexa, enquanto no controlador difuso é, em média, 12 milissegundos. A causa para o tempo de execução de um ciclo da implementação da rede neuronal ser superior à implementação do controlador difuso é não se ter otimizado o tipo de variáveis utilizadas na rede neuronal.

Como futuro melhoramento, podia-se substituir o controlo remoto simples implementado, por um com mais níveis diferenciados, conseguindo obter mais conjuntos inconfundíveis de pares entrada-saída, permitindo assim efetuar um treino da RNA com melhor desempenho. Outro melhoramento, que com a RNA implementada nesta abordagem seria menos trabalhosa do que com a abordagem anterior, seria o acréscimo do número de sonares utilizados, principalmente a colocação de sensores do lado esquerdo da plataforma, o que permitiria obter uma melhor "visão do meio ambiente" pelo AGV.

Outro desenvolvimento futuro no trabalho desenvolvido, seria modificar a implementação no AGV, modificando o tipo das variáveis usado, de vírgula flutuante para inteiros, permitindo assim diminuir o tempo de execução do cálculo dos valores de saída da rede neuronal.

Referências Documentais

[1] OSORIO, Dário Jorge dos Reis - Controlo difuso de um AGV, Porto: Instituto Superior de Engenharia do Porto, 2010 [último acesso em 30-08-2013]
http://works.dee.isep.ipp.pt/getpdf.php?A=1050356&B=MEEC-AS

[2] DUARTE-RAMOS, Hermínio – Controlo Neuronal, Lisboa: Faculdade de Ciências e Tecnologia / Universidade Nova de Lisboa

[3] PINTO, Gustavo Filipe Lopes Correia Pinto - Identificar e testar técnicas baseadas em Redes Neuronais e Lógica Difusa para o Controlo de Semáforos em Tráfego Urbano, Porto: Faculdade de Engenharia da Universidade do Porto , 2007

[4] HAGAN, Martin T.; DEMUTH, Howard B.; BEALE, Mark H. - Neural Network Design Matlab Powerpoints [último acesso em 30-08-2013]
http://hagan.okstate.edu/nnd.html

[5] RIBEIRO, Dionísio; MACHADO, Kássio; PERERREIRS, Levi – Rede Neural Perceptron embarcada em robótica AGV/ROVER, 2008 [último acesso em 27-12-2012]
http://www3.iesam-pa.edu.br/ojs/index.php/computacao/article/viewFile/210/201

[6] Manual do sonar SRF05 [último acesso em 30-08-2013]
http://www.robot-electronics.co.uk/htm/srf05tech.htm

[7] HPS-110 Full Manual[último acesso em 30-08-2013]
http://www.handywave-usa.com/downloads/HPS_120_manual_v2.0_english.pdf

[8] IDE Eclipse [último acesso em 10/05/2013]
http://www.eclipse.org/downloads/

[9] Android SDK [último acesso em 10/05/2013]
http://developer.android.com/sdk/index.html

[10] MEIER, Reto - Professional Android 4 Application Development. WROX, Indiana: Wiley Publishing, Inc.,2012

[11] Android Developers - Android API packages index [último acesso em 02/05/2013]
http://developer.android.com/reference/packages.html

[12] SILVA, Jorge; TERRA, Rui - Controlo de um motor CC, Porto: Instituto Superior de Engenharia do Porto, 2013

[13] BEALE, Mark Hudson; HAGAN, Martin T.; DEMUTH, Howard B. - Neural Network Toolbox: User's Guide, MATLAB R2013b, The MathWorks, Inc., 2013

[14] Descrição do algoritmo Nguyen-Widrow no MATLAB [último acesso em 09/09/2013]
http://www.mathworks.com/help/nnet/ref/initnw.html

[15] Descrição do algoritmo Levenberg-Marquardt no MATLAB [último acesso em 09/09/2013] http://www.mathworks.com/help/nnet/ref/trainlm.html

[16] KAWAGUCHI, Kiyoshi – A Multi*thread*ed Software Model for Backpropagation Neural Network Applications, The University of Texas at El Paso, 2000 [último acesso em 30-08-2013]

http://wwwold.ece.utep.edu/research/webfuzzy/docs/kk-thesis/kk-thesis-html/thesis.html

Anexo A. RNA implementada em Linguagem C

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define NUM_NEURONS_HL 12
#define NUM_INPUTS_HL 4

#define NUM_NEURONS_OL 2
#define NUM_INPUTS_OL 12

#define MAX_NUM_ITERATIONS 1000
#define MAX_ERROR_ACEPTABLE 0

#define ACT_SIGMOID 0
#define ACT_BINARY 1
#define ACT_LINEAR 2

#define RANDOM_MAXIMUM 32767

#define MIN_VAL -1
#define MAX_VAL 1

#define VAL_WEIGHTS 1
#define VAL_BIAS 2
#define VAL_INPUTS 3

#define NUM_SAMPLES 300

/*****************************************************************/
/*                                                               */
/*   Função que calcula o valor de saída através da função de    */
/*                          ativação sigmoide                    */
/*                                                               */
/*****************************************************************/

float sigmoid(float x)
{
     float exp_value=0;
     float return_value=0;

     /*** Exponential calculation ***/
     exp_value = exp((double) -x);

     /*** Final sigmoid value ***/
     return_value = 1 / (1 + exp_value);

     return return_value;
}
```

```
/*****************************************************************/
/*                                                               */
/*    Função que calcula o valor de saída através da função de   */
/*                     função de ativação linear                 */
/*                                                               */
/*****************************************************************/

float linear(float x)
{
      return x;
}

/*****************************************************************/
/*                                                               */
/*    Função que calcula o valor de saída através da função de   */
/*                     função de ativação binária                */
/*                                                               */
/*****************************************************************/

float binary(float x, float bias)
{
      float return_value=0;

      /*** Binary Function calculation ***/
      if(x>=bias)
      return_value=1.0;
      else
      return_value=0.0;

      return return_value;
}

/*****************************************************************/
/*                                                               */
/*    Função que calcula a soma de produtos entre os pesos e     */
/*                           as entradas                         */
/*                                                               */
/*****************************************************************/

float sum_of_produts(float *first, float *second, int size)
{
    int i=0;
    float sum = 0.0;

    /*** Perform the sum-of-product calculation here ***/
    sum = 0.0;

    for(i=0;i<size;i++)
        sum = (first[i] * second[i]) + sum;

    return sum;
}
```

```
/*****************************************************************/
/*                                                               */
/*       Função que calcula o valor de saída de um neurónio      */
/*                            artificial                         */
/*                                                               */
/*****************************************************************/

float neuron(float *inputs, float *weights, int size, float bias,
int act_function)
{
      float sum=0;
      float outval=0;

      /* Perform the sum-of-product calculation of inputs and
weights */
      sum = sum_of_produts(inputs, weights, size);

      //printf("Sum = %f \n", sum);
      /* Apply bias Then do the activation function, which is a
binary function */
      if(act_function==ACT_SIGMOID)
            outval = sigmoid(sum+bias);
      if(act_function==ACT_BINARY)
            outval = binary(sum,bias);
      if(act_function==ACT_LINEAR)
            outval = linear(sum+bias);

      return outval;
}

/*****************************************************************/
/*                                                               */
/*    Função que calcula os valores de saída de cada um dos      */
/*                  neurónios de uma camada da rede              */
/*                                                               */
/*****************************************************************/

void layer_neurons(float *inputs, float *outputs,int num_inputs,
int num_outputs,float *weights,float *bias,int act_function)
{
    int i=0;     /* Loop counter */
    /* Process for every neuron in this layer */
    for(i=0;i<num_outputs;i++)
      {
            outputs[i] =
neuron(inputs,&weights[i*num_inputs],num_inputs,bias[i],act_functi
on);
    }
    return;
}
```

```
/*****************************************************************/
/*                                                               */
/*    Função que inicializa o vetor dos pesos de cada uma das    */
/*                  camadas da rede neuronal                     */
/*                                                               */
/*****************************************************************/

void values_init(float *values, int size, float  minval, float
maxval, int op)
{
    int i=0;
    int random_generated=0;

    for(i=0;i<size;i++)
      {
            random_generated = rand();
        values[i] = (maxval - minval) * (float)random_generated /
RANDOM_MAXIMUM + minval;
            if(op==VAL_INPUTS)
                 printf("Entrada inicial %d = %f\n",i,values[i]);
            if(op==VAL_WEIGHTS)
                 printf("Peso inicial %d = %f\n",i,values[i]);
            if(op==VAL_BIAS)
                 printf("Polaridade  inicial %d\n=
%f",i,values[i]);

      }

    return;
}

/*****************************************************************/
/*                                                               */
/*    Função que calcula o valor do beneficio de um neurónio     */
/*                      de camada escondida                      */
/*                                                               */
/*****************************************************************/

float benefit_calc_HL(float output, float *benefit_front, float
*weight_front,int neurons_front)
{
     int k;
     float sum = 0.0;
     float benefit=0.0;

    for(k=0;k<neurons_front;k++)
      {
        sum = sum +
benefit_front[k]*weight_front[k*NUM_NEURONS_OL];
    }

    benefit = output * (1.0 - output) * sum;
      return benefit;
}
```

```
/*****************************************************************/
/*                                                               */
/*    Função que calcula o valor do beneficio de um neurónio     */
/*                        de camada saída                        */
/*                                                               */
/*****************************************************************/

float benefit_calc_OL(float output, float target)
{
    float benefit=0;
    benefit = -2.0 * (target - output);
    return  benefit;
}

/*****************************************************************/
/*                                                               */
/*    Função que ajusta os valores dos pesos de um neurónio      */
/*                                                               */
/*****************************************************************/


void weight_adj(float *weight, float *last_variation, float
*input, int neurons_back, float benefit, float ratio, float alpha)
{
    int k;
      float variation;

    for(k=0;k<neurons_back;k++)
      {
            variation=0;
            variation = ratio /* (1 - alpha) */* benefit *input[k];
        weight[k] = weight[k] /*- ratio * benefit *input[k];*/ -
variation - alpha*last_variation[k];
        printf("Peso ajustado = %f variacao do peso = %f variacao
anterior do peso = %f \n", weight[k],variation,last_variation[k]);
            last_variation[k]=variation;
    }

    return;
}

/*****************************************************************/
/*                                                               */
/*    Função que ajusta o valor de polarização de um neurónio    */
/*                                                               */
/*****************************************************************/

void bias_adj(float *bias, float *last_variation, float benefit,
float ratio, float alpha)
{
     float variation=0;

     variation=ratio /* ( 1 - alpha)*/ * benefit;
    bias[0] = bias[0] /*- ratio * benefit;*/ - variation -
alpha*last_variation[0];
```

```
    printf("Polaridade ajustada = %f variacao da polarizacao = %f
variacao anterior da polaridade = %f\n",
bias[0],variation,last_variation[0]);
     last_variation[0]=variation;
     return;
}

/*****************************************************************/
/*                                                               */
/*                          Função Principal                     */
/*                                                               */
/*****************************************************************/

int main (int argc, char *argv[])
{
     double pi=3.14159265358979323846;
     int iteracao=0;

     float weights_HL[NUM_INPUTS_HL*NUM_NEURONS_HL]={0};
     float weights_OL[NUM_INPUTS_OL*NUM_NEURONS_OL]={0};
     float bias_HL[NUM_NEURONS_HL]={0};
     float bias_OL[NUM_NEURONS_OL]={0};
     float inputs[NUM_SAMPLES][NUM_INPUTS_HL]={0};
     int mode=0;

     float weights_last_var_HL[NUM_INPUTS_HL*NUM_NEURONS_HL]={0};
     float weights_last_var_OL[NUM_INPUTS_OL*NUM_NEURONS_OL]={0};
     float bias_last_var_HL[NUM_NEURONS_HL]={0};
     float bias_last_var_OL[NUM_NEURONS_OL]={0};
     float outputs_HL[NUM_NEURONS_HL]={0};
     float outputs_OL[NUM_NEURONS_OL]={0};
     float targets[NUM_SAMPLES][NUM_NEURONS_OL]={0};
     float learning_ratio=0.01;
     float alpha=0.9;
     float benefit_OL[NUM_NEURONS_OL]={0};
     float benefit_HL[NUM_NEURONS_HL]={0};
     float mean_square_error=100;
     int inicio=1;

     FILE *file;
     float n_line[NUM_SAMPLES]={0};
     float s1[NUM_SAMPLES]={0};
     float s2[NUM_SAMPLES]={0};
     float s3[NUM_SAMPLES]={0};
     float s4[NUM_SAMPLES]={0};
     float EO[NUM_SAMPLES]={0};
     float ED[NUM_SAMPLES]={0};
     float referencia[NUM_SAMPLES]={0};
     float V_final[NUM_SAMPLES]={0};
     float W_final[NUM_SAMPLES]={0};
     int line = 0;

     values_init(weights_HL,(NUM_INPUTS_HL*NUM_NEURONS_HL),MIN_VAL
,MAX_VAL,VAL_WEIGHTS);
     values_init(weights_OL,(NUM_INPUTS_OL*NUM_NEURONS_OL),MIN_VAL
,MAX_VAL,VAL_WEIGHTS);
```

```
     values_init(bias_HL,NUM_NEURONS_HL,MIN_VAL,MAX_VAL,VAL_BIAS);
     values_init(bias_OL,NUM_NEURONS_OL,MIN_VAL,MAX_VAL,VAL_BIAS);

     file = fopen("data_emergencia.txt", "r");
    if(file == NULL){
       perror("Ocorreu o seguinte erro: ");
       printf("O valor de é errno: %d\n", errno);
    }
     printf("Ficheiro Carregado\r\n");
     while(fscanf(file,"%f %f %f %f %f %f %f %f %f
%f\r\n",&n_line[line],&s1[line],&s2[line],

&s3[line],&s4[line],&EO[line],&ED[line],&referencia[line],&V_final
[line],&W_final[line])>9)
    {
          printf("%f %f %f %f %f %f %f %f %f
%f\r\n",n_line[line],s1[line],s2[line],s3[line],s4[line],EO[line],
ED[line],
          referencia[line],V_final[line],W_final[line]);
          line++;
     }
     fclose(file);
     for(line=0;line<NUM_SAMPLES;line++)
     {
          inputs[line][0]=s1[line];
          inputs[line][1]=s3[line];
          inputs[line][2]=EO[line];
          inputs[line][3]=ED[line];
          targets[line][0]=V_final[line];
          targets[line][1]=W_final[line];
          printf("\r\n\r\n%d Entradas: %f %f Saidas desejadas: %f
%f\r\n\r\n",line,inputs[line][0],inputs[line][1],targets[line][0],
targets[line][1]);
     }
     line=0;

     mode=atoi(argv[1]);

     if(mode==0)
     {
               if(inicio==1)
               {
                    file = fopen("pesos_emergencia.txt", "r");
                    if(file == NULL){
                         perror("Ocorreu o seguinte erro: ");
                         printf("O valor de é errno: %d\n",
errno);
                    }
                    printf("Ficheiro Carregado\r\n");
                    printf("\r\nPesos das entradas dos neuronios
da camada escondida:\r\n");

     for(line=0;line<(NUM_INPUTS_HL*NUM_NEURONS_HL);line++)
                    {
                         fscanf(file,"%f ",&weights_HL[line]);
                         printf("%f ",weights_HL[line]);
```

```
                        }
                        printf("\r\nPesos das entradas dos neuronios
da camada de saida:\r\n");

      for(line=0;line<(NUM_INPUTS_OL*NUM_NEURONS_OL);line++)
                        {
                              fscanf(file,"%f ",&weights_OL[line]);
                              printf("%f ",weights_OL[line]);
                        }
                        printf("\r\nPolaridade dos neuronios da
camada escondida:\r\n");
                        for(line=0;line<NUM_NEURONS_HL;line++)
                        {
                              fscanf(file,"%f ",&bias_HL[line]);
                              printf("%f ",bias_HL[line]);
                        }
                        printf("\r\nPolaridade dos neuronios da
camada de saida:\r\n");
                        for(line=0;line<NUM_NEURONS_OL;line++)
                        {
                              fscanf(file,"%f ",&bias_OL[line]);
                              printf("%f ",bias_OL[line]);
                        }
                        printf("\r\n\r\n\r\n");
                        fclose(file);
                  }

                  file = fopen("dados_sim_emergencia.txt", "w+");
                  if(file == NULL){
                        perror("Ocorreu o seguinte erro: ");
                        printf("O valor de é errno: %d\n", errno);
                  }
                  printf("Ficheiro Carregado\r\n");
                  line=0;
                  while(line<NUM_SAMPLES)
                  {
                        printf("\r\n\r\nEntradas: %f %f Saidas
desejadas: %f
%f\r\n\r\n",inputs[line][0],inputs[line][1],targets[line][0],targe
ts[line][1]);


      layer_neurons(inputs[line],outputs_HL,NUM_INPUTS_HL,NUM_NEURO
NS_HL,weights_HL,bias_HL,ACT_SIGMOID);

      layer_neurons(outputs_HL,outputs_OL,NUM_INPUTS_OL,NUM_NEURONS
_OL,weights_OL,bias_OL,ACT_LINEAR);
                        printf("O valor de saida da rede e: %f
%f\n",outputs_OL[0],outputs_OL[1]);
                        printf("Saidas da camada escondida: %f %f %f
%f %f %f %f %f %f
%f\n",outputs_HL[0],outputs_HL[1],outputs_HL[2],outputs_HL[3],

      outputs_HL[4],outputs_HL[5],outputs_HL[6],outputs_HL[7],outpu
ts_HL[8],outputs_HL[9]);
```

```
                        printf("Erro camada de saida: %f
%f\n",(targets[line][0]-outputs_OL[0]),(targets[line][1]-
outputs_OL[1]));
                        if(inicio==1)
                              {
                                    mean_square_error=0;
                                    inicio=0;
                              }
                        mean_square_error=(((targets[line][0]-
outputs_OL[0])*(targets[line][0]-outputs_OL[0])));
                        mean_square_error+=(((targets[line][1]-
outputs_OL[1])*(targets[line][1]-outputs_OL[1])));

      mean_square_error=(mean_square_error/NUM_NEURONS_OL);
                        printf("MSE calculado: %f
\n\n",mean_square_error);
                        fprintf(file,"%d %f %f %f
%f\r\n",line,targets[line][0],outputs_OL[0],targets[line][1],outpu
ts_OL[1]);
                        line++;

                        system("PAUSE");
                  }
                  printf("Fim da simulacao da rede
neuronal\n\n\n\n",iteracao);
                  fclose(file);
                  system("PAUSE");
      }

      if(mode==1)
      {

      while(mean_square_error>MAX_ERROR_ACEPTABLE&&iteracao<MAX_NUM
_ITERATIONS)
            {
                  printf("\r\n\r\nEntradas: %f %f Saidas desejadas:
%f
%f\r\n\r\n",inputs[line][0],inputs[line][1],targets[line][0],targe
ts[line][1]);


      layer_neurons(inputs[line],outputs_HL,NUM_INPUTS_HL,NUM_NEURO
NS_HL,weights_HL,bias_HL,ACT_SIGMOID);

      layer_neurons(outputs_HL,outputs_OL,NUM_INPUTS_OL,NUM_NEURONS
_OL,weights_OL,bias_OL,ACT_LINEAR);
                  printf("O valor de saida da rede e: %f
%f\n",outputs_OL[0],outputs_OL[1]);
                  printf("Saidas da camada escondida: %f %f %f %f %f
%f %f %f %f
%f\n",outputs_HL[0],outputs_HL[1],outputs_HL[2],outputs_HL[3],

      outputs_HL[4],outputs_HL[5],outputs_HL[6],outputs_HL[7],outpu
ts_HL[8],outputs_HL[9]);
```

```
                printf("Erro camada de saida: %f
%f\n",(targets[line][0]-outputs_OL[0]),(targets[line][1]-
outputs_OL[1]));
                if(inicio==1)
                     {
                          mean_square_error=0;
                          inicio=0;
                     }
                mean_square_error=(((targets[line][0]-
outputs_OL[0])*(targets[line][0]-outputs_OL[0])));
                mean_square_error+=(((targets[line][1]-
outputs_OL[1])*(targets[line][1]-outputs_OL[1])));

     mean_square_error=(mean_square_error/NUM_NEURONS_OL);
                printf("MSE calculado: %f
\n\n",mean_square_error);

                benefit_OL[0]=benefit_calc_OL(outputs_OL[0],
targets[line][0]);
                printf("Beneficio neuronio 1 da camada de saida:
%f \n",benefit_OL[0]);
                benefit_OL[1]=benefit_calc_OL(outputs_OL[1],
targets[line][1]);
                printf("Beneficio neuronio 2 da camada de saida:
%f \n",benefit_OL[1]);

                benefit_HL[0]=benefit_calc_HL(outputs_HL[0],
benefit_OL, &weights_OL[0], NUM_NEURONS_OL);
                printf("Beneficio neuronio 1 da camada escondida:
%f \n",benefit_HL[0]);
                benefit_HL[1]=benefit_calc_HL(outputs_HL[1],
benefit_OL, &weights_OL[1], NUM_NEURONS_OL);
                printf("Beneficio neuronio 2 da camada escondida:
%f \n",benefit_HL[1]);
                benefit_HL[2]=benefit_calc_HL(outputs_HL[2],
benefit_OL, &weights_OL[2], NUM_NEURONS_OL);
                printf("Beneficio neuronio 2 da camada escondida:
%f \n",benefit_HL[2]);
                benefit_HL[3]=benefit_calc_HL(outputs_HL[3],
benefit_OL, &weights_OL[3], NUM_NEURONS_OL);
                printf("Beneficio neuronio 2 da camada escondida:
%f \n",benefit_HL[3]);
                benefit_HL[4]=benefit_calc_HL(outputs_HL[4],
benefit_OL, &weights_OL[4], NUM_NEURONS_OL);
                printf("Beneficio neuronio 2 da camada escondida:
%f \n",benefit_HL[4]);
                benefit_HL[5]=benefit_calc_HL(outputs_HL[5],
benefit_OL, &weights_OL[5], NUM_NEURONS_OL);
                printf("Beneficio neuronio 2 da camada escondida:
%f \n",benefit_HL[5]);
                benefit_HL[6]=benefit_calc_HL(outputs_HL[6],
benefit_OL, &weights_OL[6], NUM_NEURONS_OL);
                printf("Beneficio neuronio 2 da camada escondida:
%f \n",benefit_HL[6]);
                benefit_HL[7]=benefit_calc_HL(outputs_HL[7],
benefit_OL, &weights_OL[7], NUM_NEURONS_OL);
```

```
                printf("Beneficio neuronio 2 da camada escondida:
%f \n",benefit_HL[7]);
                benefit_HL[8]=benefit_calc_HL(outputs_HL[7],
benefit_OL, &weights_OL[8], NUM_NEURONS_OL);
                printf("Beneficio neuronio 2 da camada escondida:
%f \n",benefit_HL[8]);
                benefit_HL[9]=benefit_calc_HL(outputs_HL[7],
benefit_OL, &weights_OL[9], NUM_NEURONS_OL);
                printf("Beneficio neuronio 2 da camada escondida:
%f \n",benefit_HL[9]);


     weight_adj(&weights_HL[0*NUM_INPUTS_HL],&weights_last_var_HL[
0*NUM_INPUTS_HL],inputs[line], NUM_INPUTS_HL, benefit_HL[0],
learning_ratio, alpha);

     weight_adj(&weights_HL[1*NUM_INPUTS_HL],&weights_last_var_HL[
1*NUM_INPUTS_HL], inputs[line], NUM_INPUTS_HL, benefit_HL[1],
learning_ratio, alpha);

     weight_adj(&weights_HL[2*NUM_INPUTS_HL],&weights_last_var_HL[
2*NUM_INPUTS_HL], inputs[line], NUM_INPUTS_HL, benefit_HL[2],
learning_ratio, alpha);

     weight_adj(&weights_HL[3*NUM_INPUTS_HL],&weights_last_var_HL[
3*NUM_INPUTS_HL], inputs[line], NUM_INPUTS_HL, benefit_HL[3],
learning_ratio, alpha);

     weight_adj(&weights_HL[4*NUM_INPUTS_HL],&weights_last_var_HL[
4*NUM_INPUTS_HL], inputs[line], NUM_INPUTS_HL, benefit_HL[4],
learning_ratio, alpha);

     weight_adj(&weights_HL[5*NUM_INPUTS_HL],&weights_last_var_HL[
5*NUM_INPUTS_HL], inputs[line], NUM_INPUTS_HL, benefit_HL[5],
learning_ratio, alpha);

     weight_adj(&weights_HL[6*NUM_INPUTS_HL],&weights_last_var_HL[
6*NUM_INPUTS_HL], inputs[line], NUM_INPUTS_HL, benefit_HL[6],
learning_ratio, alpha);

     weight_adj(&weights_HL[7*NUM_INPUTS_HL],&weights_last_var_HL[
7*NUM_INPUTS_HL], inputs[line], NUM_INPUTS_HL, benefit_HL[7],
learning_ratio, alpha);

     weight_adj(&weights_HL[8*NUM_INPUTS_HL],&weights_last_var_HL[
8*NUM_INPUTS_HL], inputs[line], NUM_INPUTS_HL, benefit_HL[8],
learning_ratio, alpha);

     weight_adj(&weights_HL[9*NUM_INPUTS_HL],&weights_last_var_HL[
9*NUM_INPUTS_HL], inputs[line], NUM_INPUTS_HL, benefit_HL[9],
learning_ratio, alpha);


     weight_adj(&weights_OL[0*NUM_INPUTS_OL],&weights_last_var_OL[
0*NUM_INPUTS_OL],outputs_HL,NUM_NEURONS_HL,benefit_OL[0],
learning_ratio, alpha);
```

```
     weight_adj(&weights_OL[1*NUM_INPUTS_OL],&weights_last_var_OL[
1*NUM_INPUTS_OL],outputs_HL,NUM_NEURONS_HL,benefit_OL[1],
learning_ratio, alpha);


     bias_adj(&bias_HL[0],&bias_last_var_HL[0],benefit_HL[0],learn
ing_ratio, alpha);

     bias_adj(&bias_HL[1],&bias_last_var_HL[1],benefit_HL[1],learn
ing_ratio, alpha);

     bias_adj(&bias_HL[2],&bias_last_var_HL[2],benefit_HL[2],learn
ing_ratio, alpha);

     bias_adj(&bias_HL[3],&bias_last_var_HL[3],benefit_HL[3],learn
ing_ratio, alpha);

     bias_adj(&bias_HL[4],&bias_last_var_HL[4],benefit_HL[4],learn
ing_ratio, alpha);

     bias_adj(&bias_HL[5],&bias_last_var_HL[5],benefit_HL[5],learn
ing_ratio, alpha);

     bias_adj(&bias_HL[6],&bias_last_var_HL[6],benefit_HL[6],learn
ing_ratio, alpha);

     bias_adj(&bias_HL[7],&bias_last_var_HL[7],benefit_HL[7],learn
ing_ratio, alpha);

     bias_adj(&bias_HL[8],&bias_last_var_HL[8],benefit_HL[8],learn
ing_ratio, alpha);

     bias_adj(&bias_HL[9],&bias_last_var_HL[9],benefit_HL[9],learn
ing_ratio, alpha);


     bias_adj(&bias_OL[0],&bias_last_var_OL[0],benefit_OL[0],learn
ing_ratio, alpha);

     bias_adj(&bias_OL[1],&bias_last_var_OL[1],benefit_OL[1],learn
ing_ratio, alpha);

                iteracao++;
                line++;
                if(line>NUM_SAMPLES-1)
                     line=0;
                printf("Fim da iteracao %d do treino na linha
%d\n\n\n\n",iteracao,line);
                //system("PAUSE \n");
           }
           printf("\r\nPesos das entradas da camada
escondida:\r\n");
           for(line=0;line<(NUM_INPUTS_HL*NUM_NEURONS_HL);line++)
                printf(" %f",weights_HL[line]);
```

```
            printf("\r\nPesos das entradas da camada de
saida:\r\n");
            for(line=0;line<(NUM_INPUTS_OL*NUM_NEURONS_OL);line++)
                  printf(" %f",weights_OL[line]);
            printf("\r\nPolaridade dos neuronios da camada
escondida:\r\n");
            for(line=0;line<NUM_NEURONS_HL;line++)
                  printf(" %f",bias_HL[line]);
            printf("\r\nPolaridade dos neuronios da camada de
saida:\r\n");
            for(line=0;line<NUM_NEURONS_OL;line++)
                  printf(" %f",bias_OL[line]);
            printf("\r\n");

            file = fopen("pesos_alterados.txt", "w+");
            if(file == NULL){
                  perror("Ocorreu o seguinte erro: ");
                  printf("O valor de é errno: %d\n", errno);
            }
            printf("Ficheiro Carregado\r\n");
            for(line=0;line<(NUM_INPUTS_HL*NUM_NEURONS_HL);line++)
                  fprintf(file,"%f ",weights_HL[line]);
            fprintf(file,"r\n");
            for(line=0;line<(NUM_INPUTS_OL*NUM_NEURONS_OL);line++)
                  fprintf(file,"%f ",weights_OL[line]);
            fprintf(file,"\r\n");
            for(line=0;line<NUM_NEURONS_HL;line++)
                  fprintf(file,"%f ",bias_HL[line]);
            for(line=0;line<NUM_NEURONS_OL;line++)
                  fprintf(file,"%f ",bias_OL[line]);
            fprintf(file,"\r\n");
            fclose(file);
            printf("Pesos guardados no ficheiro carregado\r\n");
            line=0;

            if(iteracao>=MAX_NUM_ITERATIONS)
            {
                  printf("Fim do treino sem sucesso, o erro nao
convergiu \n");
            }
            else printf("Fim do treino com sucesso em %d iteracoes
",iteracao);

      }
      return 0;
}
```

Anexo B. RNA implementada no microcontrolador PIC

O código elaborado para implementar a RNA no microcontrolador PIC está no CD anexado ao relatório da tese.

Anexo C. Aplicação de monitorização e controlo do AGV

O código elaborado para implementar a aplicação de monitorização e controlo do AGV está no CD anexado ao relatório da tese.