FERNANDO BIAZI NASCIMENTO

CONTROLE DE BRAÇOS MECÂNICOS POR MEIO DE TÉCNICAS DE INTELIGÊNCIA ARTIFICIAL

Trabalho de Graduação Interdisciplinar apresentado à escola de Engenharia da Universidade Presbiteriana Mackenzie como requisito parcial para obtenção do título de Bacharel em Engenharia Elétrica.

ORIENTADOR: PROF. DR. MARCO ANTÔNIO ASSIS DE MELO

São Paulo
2010

AGRADECIMENTOS

Ao Prof. Dr. Marco Antônio Assis de Melo, pela fantástica orientação focada na conclusão do trabalho sem deixar de incentivar pesquisas relacionadas e pela qual muito eu aprendi.

Ao Prof. Dr. Gustavo Pérez Alvarez, meu primeiro orientador para este trabalho, por ter sugerido o tema e iniciado o trabalho de orientação sem o qual o trabalho não seria realizado, ou não seria de forma bem sucedida.

À Profª. Drª. Ivanilda Matille pelas excelentes sugestões tanto de formatação como formas de exposição das informações.

Ao Eng. Franco Pontillo por disponibilizar os recursos de servidor web para conteúdo citado neste trabalho.

E, por fim, aos desenvolvedores dos softwares utilizados para a realização deste trabalho e da apresentação: Linux, distribuição Gentoo do Linux, X11, Gnome, Openoffice, Firefox, Lazarus, Blender e o Lives (organizados primeiramente na ordem de utilização neste trabalho e depois na ordem de desenvolvimento dos softwares em si), além dos softwares de suporte dos quais estes dependem.

A imaginação é, de longe, muito mais importante do que o conhecimento (Albert Einstein).

RESUMO

Desde os documentos literários mais antigos disponíveis, existe a presença, em imaginação, de máquinas que podem realizar o trabalho do homem, permanecendo desta forma até as obras de ficção mais recentes. Muitos estudos foram feitos ao longo da história e aos poucos se avança tanto em ficção quanto na realidade, com aplicações em tarefas que não podiam ser realizadas ou que eram muito custosas. Este trabalho tem o objetivo de experimentar a aplicação de redes neurais artificiais no controle de braços mecânicos. São apresentados alguns modelos de redes neurais e de treinamento. Uma implementação de rede neural artificial do tipo perceptron multi-camada com algoritmo de aprendizado retro-propagação é feita em linguagem Object Pascal seguindo as orientações do trabalho, que permite, da mesma forma, implementá-la em outras linguagens de programação. O modelo proposto (orientado a objeto) pode ser facilmente modificado para criar implementações de outros tipos de redes neurais artificiais ou mesmo a utilização em outras aplicações. As simulações mostram que é possível utilizar redes neurais artificiais do tipo perceptron multi-camadas para o controle de braços mecânicos e que um bom conjunto de dados de treinamento podem resultar em um comportamento muito eficiente.

Palavras-chave: Rede neural artificial. Braço mecânico. Robô. Perceptron multi-camada. Retro-propagação.

ABSTRACT

Since the oldest literary documents available, there is the presence of machines that can do the men's work, remaining so until the more recent fiction works. Many studies has been done along the history and, smoothly, there are advances both in fiction and in reality, with applications on tasks that could not be done or that has a high cost. This work have the objective of experiment the application of artificial neural networks operating on the control of mechanical arms. It will show some models of neural networks and training techniques. An implementation of an artificial neural network of the type multilayer perceptron with backpropagation learning algorithm is done using Object Pascal programming language following the directions of this work, that allows, on the same way, to implement it using another programming languages. The proposed model (object oriented) can be easily modified to create implementations of other types of artificial neural networks or even the use in other applications. The simulations shows that it's possible to use artificial neural networks of the type multilayer perceptron in order to control mechanical arms and that a good training set may result in a very efficient behavior.

Keywords: Artificial Neural Network. Mechanical arm. Robot. Multilayer perceptron. Backpropagation.

LISTA DE ILUSTRAÇÕES

Desenho 1: Juntas mecânicas simples utilizadas em robôs.....18
Desenho 2: Estrutura de um neurônio multipolar.....26
Fotografia 1: Robô esférico ABB modelo IRB 2400.....19
Quadro 1: Tipos de controle inteligente que podem substituir alguns tipos de controles convencionais.....22
Quadro 2: Comparação de algumas características do cérebro humano e de um computador. 28
Quadro 3: Tempos de ação e observações para os controles simulados.....84
Tabela 1: Fatores utilizados nas simulações de controle PID para o braço de um segmento.. .78

LISTA DE DIAGRAMAS

Diagrama 1: Diagrama em blocos do problema de controle de um robô.....20
Diagrama 2: Diagrama esquemático de um neurônio artificial.....24
Diagrama 3: Exemplo de rede SLP.....29
Diagrama 4: Exemplo de rede MLP.....30
Diagrama 5: Exemplo de rede Hopfield.....31
Diagrama 6: Etapas de desenvolvimento e aplicação de um sistema de controle.....39
Diagrama 7: Funcionalidade de um ponteiro usado como um vetor de vários valores.....46
Diagrama 8: Classe TNeuron, método Create.....47
Diagrama 9: Classe TNeuron, método Destroy.....48
Diagrama 10: Classe TNeuron, método checkRangeWith.....48
Diagrama 11: Classe TNeuron, método getWeight.....49
Diagrama 12: Classe TNeuron, método setWeight.....49
Diagrama 13: Classe TNeuron, método process.....49
Diagrama 14: Classe TNeuron, método setAbsoluteError.....50
Diagrama 15: Classe TNeuron, métod addDeltas.....50
Diagrama 16: Classe TNeuron, método adjust.....50
Diagrama 17: Classe TLayer, método Create.....52
Diagrama 18: Classe TLayer, método Destroy.....52
Diagrama 19: Classe TLayer, método freeNeurons.....53
Diagrama 20: Classe TLayer, método checkRangeWith.....53
Diagrama 21: Classe TLayer, método proccess.....53
Diagrama 22: Classe TLayer, método readOutputs.....54
Diagrama 23: Classe TLayer, método setErros.....54
Diagrama 24: Classe TLayer, método readPrevErrors.....55
Diagrama 25: Classe TLayer, método addDeltas.....55
Diagrama 26: Classe TLayer, método adjust.....55
Diagrama 27: Classe TLayer, método getNeuronDelta.....56
Diagrama 28: Classe TLayer, método getNeuronOutput.....56
Diagrama 29: Classe TLayer, método getNeuronWeight.....56
Diagrama 30: Classe TLayer, método getNeuronWeight.....56
Diagrama 31: Classe TNN, método Create.....58
Diagrama 32: Classe TNN, método Destroy.....58

Diagrama 33: Classe TNN, método freeLayers.....59
Diagrama 34: Classe TNN, método checkRangeWith.....59
Diagrama 35: Classe TNN, método process.....60
Diagrama 36: Classe TNN, método readOutputs.....60
Diagrama 37: Classe TNN, método getOutput.....61
Diagrama 38: Classe TNN, método computeErrosTo.....61
Diagrama 39: Classe TNN, método addDeltas.....62
Diagrama 40: Classe TNN, método adjust.....62
Diagrama 41: Classe TNN, método train.....63
Diagrama 42: Classe TNN, método getLayer.....63
Diagrama 43: Classe TNN, método getLayerInputCount.....63
Diagrama 44: Classe TNN, método getNeuronCount.....63
Diagrama 45: Classe TNN, método getNeuronOutput.....64
Diagrama 46: Classe TNN, método getNeuronWeight.....64
Diagrama 47: Classe TNN, método setNeuronWeight.....64
Diagrama 48: Diagrama em blocos da aproximação feita para o cálculo integral.....66
Diagrama 49: Diagrama em blocos da aproximação feita para o cálculo diferencial.....66
Diagrama 50: Diagrama em blocos simplificado do sistema de controle simulado.....68
Diagrama 51: Diagrama em blocos do sistema de controle simulado com o bloco de registro dos valores.....68
Diagrama 52: Diagrama em blocos do controlador PID no sistema simulado.....69
Diagrama 53: Diagrama em blocos do controle do braço de um segmento por RNA.....69

LISTA DE TELAS

Tela 1: Programa de simulação, janela principal.....70
Tela 2: Programa de simulação, janela “Luzes”.....71
Tela 3: Programa de simulação, janela “Nova Rede Neural”.....73
Tela 4: Programa de simulação, janela “Processando...”.....73
Tela 5: Resultados da simulação do braço mecânico com controle proporcional.....79
Tela 6: Resultados da simulação do braço mecânico com controle por RNA simulando controle proporcional.....79
Tela 7: Resultados da simulação do braço mecânico com controle proporcional-integral.....80
Tela 8: Resultados da simulação do braço mecânico com controle por RNA simulando controle proporcional-integral.....80
Tela 9: Resultados da simulação do braço mecânico com controle proporcional-diferencial.. 81
Tela 10: Resultados da simulação do braço mecânico com controle por RNA simulando controle proporcional-diferencial.....82
Tela 11: Resultados da simulação do braço mecânico com controle proporcional-integral-diferencial.....83
Tela 12: Resultados da simulação do braço mecânico com controle por RNA simulando controle proporcional-integral-diferencial.....83

SUMÁRIO

1. INTRODUÇÃO....12
1.1. OBJETIVOS....12
1.2. JUSTIFICATIVAS....12
1.3. METODOLOGIA....13
1.4. ESTRUTURA DO TRABALHO....13
2. REVISÃO DA LITERATURA....15
3. INTRODUÇÃO TEÓRICA....17
3.1. BRAÇOS MECÂNICOS....17
3.2. INTELIGÊNCIA ARTIFICIAL....21
3.3. REDES NEURAIS ARTIFICIAIS....23
3.3.1. Neurônios Artificiais....24
3.3.2. Redes Neurais Naturais....25
3.3.3. Tipos de RNA....28
3.3.4. Treinamento....32
3.3.5. Vantagens Esperadas da Utilização de RNA's para Controle de Braços Mecânicos....39
4. DESENVOLVIMENTO DE REDE NEURAL PARA USO EM PROGRAMAS DE COMPUTADOR....42
4.1. AVALIAÇÃO DA LINGUAGEM DE PROGRAMAÇÃO E DA INTERFACE DE DESENVOLVIMENTO....42
4.2. AVALIAÇÃO DO TIPO DE RNA A SER IMPLEMENTADA....43
4.3. DESENVOLVIMENTO DO CÓDIGO FONTE....45
4.3.1. Desenvolvimento da classe TNeuron....46
4.3.2. Implementação da classe TLayer....51
4.3.3. Implementação da classe TNN....56
5. PROGRAMA DE SIMULAÇÃO QUE UTILIZA A RNA DESENVOLVIDA. 65
5.1. CARATERÍSTICAS E FINALIDADE DO PROGRAMA....65
5.2. APRESENTAÇÃO GRÁFICA DO PROGRAMA....70
5.3. UTILIZAÇÃO DO PROGRAMA....74
5.3.1. Procedimentos para Simulação com Controlador PID....75
5.3.2. Procedimentos para Simulação com Controle por RNA....75
6. SIMULAÇÕES....77

6.1. PREPARAÇÃO E DEFINIÇÃO DE PARÂMETROS....77
6.2. RESULTADOS E ANÁLISES....78
7. CONCLUSÕES....85
REFERÊNCIAS....87
APÊNDICES....91
ANEXOS....105

1. INTRODUÇÃO

Máquinas capazes de realizar trabalhos servindo a humanidade estão presente em obras de ficção desde a mais antiga obra literária europeu-ocidental conhecida, e estão se tornando realidade conforme a evolução de tecnologias permite.

Robôs são máquinas que podem agir sobre seu ambiente com base em decisões tomadas de acordo com detecções, pelos seus sensores, das características do ambiente e dos objetivos, com a possibilidade de alteração dos objetivos por meio de uma reprogramação.

Atualmente, robôs com forma humana são voltados principalmente para estudo e demonstrações, os robôs que realmente possuem aplicações práticas têm forma adaptada às funções que podem desempenhar, como robôs de pintura, soldagem ou de manipulação de objetos.

O controle dos robôs normalmente é feito com associação de programas de computadores e controladores lógico-programáveis (CLP's). Neste trabalho estuda-se a concepção de redes neurais artificiais (RNA's) que possam fazer o controle de braços mecânicos.

1.1. OBJETIVOS

Será feita uma introdução a sistemas de braços mecânicos, a sistemas de redes neurais artificiais, ao emprego das técnicas de inteligência artificial no controle de articulações de braços mecânicos, e finalmente, uma simulação em computador com um exemplo de braço mecânico e uma rede neural que possa controlá-lo.

Pretende-se assim, de simplificar a construção, a implantação e operação de braços mecânicos, aumentar o número de aplicações em que podem atuar e aperfeiçoar as já existentes.

1.2. JUSTIFICATIVAS

Com maior facilidade do controle de braços mecânicos, os custos para se instalar um robô que use este tipo de mecanismo ficarão menores, isto deve refletir em um aumento da quantidade de empresas que podem instalar robôs em suas linhas de produção ou mesmo outros tipos de entidades para realizar qualquer tipo de tarefa a que se apliquem robôs.

Com aprimoramento do controle, aumenta-se a precisão das tarefas já realizadas por robôs e o próprio número de aplicações, operações de risco ou em ambientes inóspitos para os seres humanos como, por exemplo, manutenção de centrais nucleares, operações de instalação e manutenção de equipamentos a altas profundidades ou no espaço, manipulação de cargas pesadas a alturas elevadas, operações em ambientes de temperaturas extremas ou que representam risco de infecções ou contaminações.

Desde o documento literário europeu-ocidental mais antigo de que se tem notícia existe o registro da imaginação de máquinas capazes de realizar trabalho que possa ajudar o homem. A evolução se deu tanto em ficção como na realidade. Esta evolução também foi tomada como inspiração para a elaboração deste trabalho.

1.3. METODOLOGIA

Este trabalho inclui uma introdução aos assuntos que utiliza, seguido do desenvolvimento comentado de uma RNA do tipo perceptron multi-camada (multilayer perceptron – MLP) e a apresentação do programa desenvolvido que a utiliza. O programa conta, ainda com uma simulação de controlador PID com o qual são gerados dados de treinamento e para comparação com o controle por RNA.

Foram feitas simulações com características diferentes do controlador PID para gerar dados que foram gravados em conjuntos de treinamento utilizados para treinar, cada um uma RNA, e depois foram comparados os comportamentos do braço mecânico controlado por cada RNA e pelo controlador PID que gerou os seus respectivos dados de treinamento. Os gráficos gerados para comparação mostram apenas um movimento longo e que permite a observação do comportamento ao longo da trajetória.

1.4. ESTRUTURA DO TRABALHO

A introdução ao assunto, objetivos, justificativas e metodologia são feitos no capítulo 1. A revisão da literatura que resume o conteúdo de alguns trabalhos é feita no capítulo 2.

A introdução teórica que inclui algumas definições de robótica para controle de braços mecânicos, dedefinições de RNA's entre outras técnicas de inteligência artificial é feita no capítulo 3.

O desenvolvimento e explicação das ferramentas utilizadas neste trabalho são mostrados nos capítulo 4 e 5, as simulações e os resultados no capítulo 6 e a conclusão no capítulo 7.

2. REVISÃO DA LITERATURA

No trabalho de ASANO, et al. foi proposta uma RNA para calcular os ajustes das constantes Kp, Ki e Kd de controladores PID durante o uso (somente os ajustes, não os valores totais em si). Os ajustes que a RNA faz no controlador são dependentes dos valores de enttrada.O trabalho mostra em simulações que a RNA pode fazer ajustes durante o uso do controlador de acordo com os valores de entrada do mesmo para evitar a oscilação e diz que o esquema proposto de controle está sendo implementado em um sistema real e sendo extendido a sistema de várias variáveis.

BONACORSO possui estudos de controles e atuadores pneumáticos, incluindo sistemas hidráulicos como freios ou direção hidráulica.

CHRISTODOULOU e GEORGIOPOULOS na obra *Applications of Neural Networks in Electromagnetics* fizeram uma excelente introdução a RNA's de diversos tipos com explicações sobre o funcoinamento e variados exemplos práticos e experimentais em algumas áreas (como interpretação de imagens por exemplo), além da aplicação na área de eletromagnetismo, obviamente.

GUPTA tem um estudo avançado dos cálculos envolvidos para realizar o controle de robôs.

FAUSSET em *Fundamentals of Neural Networls* tem um estudo aprofundado de RNA's, apresentando-as divididas por aplicação, e não por ordem cronológica da evolução das mesmas.

FRÖHLICH disponibilizou classes desenvolvidas em java capazes de criar RNA's com retro-propagação ou do tipo Kohonen, com documentação para uso e um exemplo de aplicação para redes Kohonen.

No trabalho de SMAGT e SCHULTEN, sugere-se um algoritmo realimentado altamente adaptativo para controlar um braço mecânico referido como rubbertuator SoftArm e que possui atuadores do tipo músculos artificiais. O sistema de controle consiste em dois programas rodando em dois processadores, o primeiro processador roda o programa que calcula a velocidade e a aceleração da junta, os dados são transmitidos ao segundo, que possui a RNA, que calcula o comando a ser enviado aos atuadores. Houve sucesso seguindo a trajetória $\text{sen}(t)\cos^2(11t)$ com erro de até 1° para movimentos de “velocidade moderada” de uma junta conforme descrito na conclusão. O trabalho diz que mais testes são necessários para velocidades mais altas e que se iniciaram estudos para controle simultâneo de várias juntas.

Informações históricas e registros podem ser encontradas nos trabalhos de CAEIRO, PIRES, TATIBANA e KAETSU, e TESLA MEMORIAL SOCIETY OF NEW YORK, sendo o primeiro e o terceiro tratando de robótica e os outros dois tratando de RNA's.

Os trabalhos de GUPTA, de OGATA, de SPONG, tratam de teorias de controle, alguns somente de partes específicas. Outros estudos de inteligência artificial e RNA's podem ser encontrados nos trabalhos de FARRELL, de FU, de HARVEY, e de TATIBANA e KAETSU.

Informações úteis e possivelmente atualizadas podem ser encontradas na Wikipédia, porém, para esta fonte, o conteúdo deve sempre ser verificado perante a verificação das fontes citadas no arquivo, estas também devendo ser verificadas para assegurar que são adequadas.

3. INTRODUÇÃO TEÓRICA

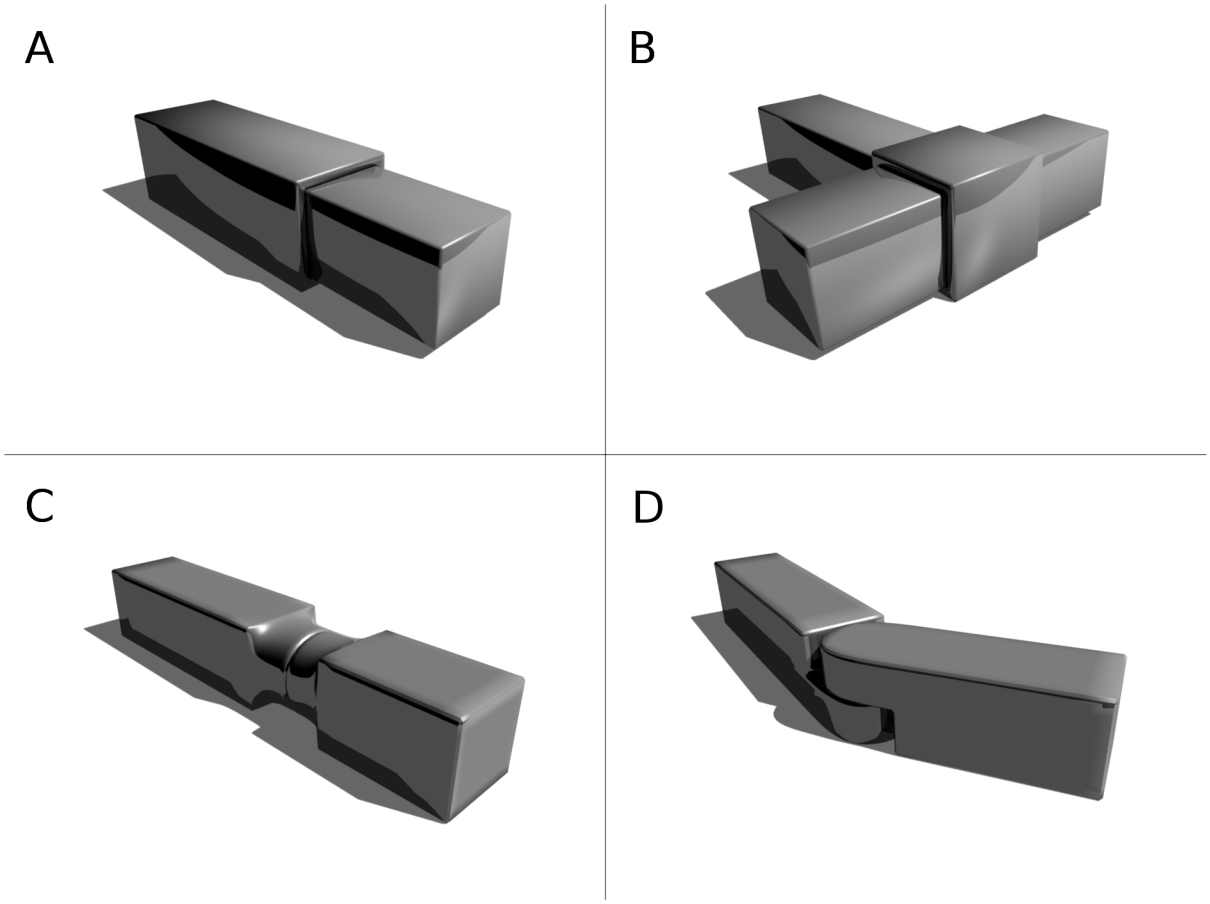
Para o desenvolvimento deste trabalho foram estudados geometria de braços mecânicos e o funcionamento de RNA's, mostrados neste capítulo com uma breve abordagem das áreas às quais pertencem.

3.1. BRAÇOS MECÂNICOS

Uma importante classe de robôs são os braços mecânicos, que possuem diversas aplicações incluindo pintura, montagem de peças, manipulação de objetos, soldagem, corte, acabamentos e outras aplicações em indústrias, exploração espacial, exploração a grandes profundidades e manutenção ou operação de sistemas de difícil acesso. Nos últimos anos também vêm sendo utilizados como ferramenta para remoção e/ou desarmamento de explosivos por forças táticas anti-bomba (GUPTA, 1997).

Os robôs desta classe possuem juntas e vínculos. Os vínculos são as pares rígidas que dão as características de dimensões do robô e as juntas são responsáveis por unir e mover os vínculos um em relação ao outro para a operação.

Existem dois tipos principais de juntas, prismáticas, que deslizam o vínculo posterior em relação ao vínculo anterior, e as rotacionais ou revolutivas, que giram o vínculo posterior em relação ao anterior. Ambas podem estar dispostas de várias formas, são mostradas as disposições longitudinal e transversal para cada uma no Desenho 1.



Desenho 1: Juntas mecânicas simples utilizadas em robôs
A: Prismática longitudinal; B: Prismática transversal;
C: Rotacional longitudinal; D: Rotacional transversal.

Considerando que não existem redundâncias entre os movimentos das juntas, isto é, o movimento de uma junta não pode ser sempre associado ao movimento de outra(s) junta(s), de forma que não podem ser substituídas por uma única com o mesmo efeito, cada junta adiciona ao robô um grau de liberdade, seja ela prismática ou rotacional.

Existem juntas mais complexas que podem ser obtidas com associação dos tipos básicos, mas resultam em dificuldades como restrições mecânicas ou a forma de acionamento por atuadores, lembrando-se que todas as juntas de um braço mecânico devem ser controladas em todos os seus graus de liberdade.

Os robôs são classificados conforme a área (ou, mais adequadamente, o volume) de atuação aproximado a um sólido geométrico. O robô cartesiano atua no volume de um paralelepípedo, e possui somente juntas prismáticas. O robô cilíndrico atua no volume de um cilindro e possui uma junta prismática e uma ou mais juntas rotacionais. O robô esférico atua no volume de uma esfera, e é formado por várias juntas rotacionais e algumas vezes,

prismáticas também. Estas são as três classificações mais comuns de robôs. A Fotografia 1 apresenta, ao centro, um robô esférico da ABB modelo 2400.

Fotografia 1: Robô esférico ABB modelo IRB 2400
Fonte: ABB, c2010.

O movimento de cada articulação influencia a posição da ferramenta ou garra do robô. Para representar as relações, utiliza-se uma série de valores que descrevem as posições atuais, além das alterações. Normalmente os robôs esféricos possuem seis graus de liberdade, o que resulta em uma redundância quanto às configurações dos vínculos para atingir as posições da ferramenta (existe mais de uma configuração de posição dos vínculos que pode colocar a ferramenta em uma mesma posição), o que pode ser útil para alcançar um objetivo desviando-se de um obstáculo ou simplesmente ter a opção de escolher uma configuração mais rápida para um objetivo dependendo da configuração atual.

Quando uma junta se move em um robô, o modo como as outras juntas movem a ferramenta também se altera, pois além da cinemática do próprio braço, mudam fatores externos que atuam sobre ele, como gravidade, possivelmente magnetismo, posições de obstáculos que não são as mesmas em relação às outras juntas, inércia e atrito.

Por causa da complexidade da cinemática e da dinâmica do manipulador, o problema de controle é geralmente dividido em três estágios: planejamento de movimento, geração da trajetória e monitoramento da trajetória (SPONG, 1996, p. 1341, tradução nossa).

Outros fatores que complicam ainda mais os cálculos são flexibilidade dos vínculos, inércia, atrito, força gravitacional e outras forças externas que atuam sobre todo o robô. A técnica utilizada para o projeto de robôs é fazer o modelamento do efeito dinâmico mais dominante para o caso particular, ignorando os outros e fazendo-o insensível ou muito robusto em relação aos efeitos ignorados.

Devido à quantidade de fatores envolvidos e à dificuldade de tratar todos eles, o problema de controle de um braço mecânico é normalmente simplificado para se fazer o

modelamento do efeito dinâmico mais dominante para o caso particular e fazendo-o suficientemente robusto em relação aos outros efeitos de modo que possam ser ignorados.

Seja a origem do sistema de coordenadas principal colocado na base do robô, "x" a posição da ferramenta em relação ao sistema de coordenadas principal com três coordenadas pertencentes ao conjunto dos números reais, um vetor "$q = (q_1, q_2, \ldots, q_n)$" que possui uma medida da posição de cada articulação e desta forma é chamado de configuração, uma matriz R 3 x 3 que descreve a inclinação e rotação da ferramenta em relação ao sistema de coordenadas principal, uma matriz T^n que descreve um toroide no espaço, sendo ele formado por n círculos (numericamente igual àquantidade de juntas), e uma matriz SE(3) cujos elementos são chamados movimentos rígidos e que descreve posições possíveis para a ferramenta, a equação (1) pode ser escrita para descrever o problema de controle do braço mecânico:

$$X_0 = \begin{bmatrix} x(q) \\ R(q) \end{bmatrix} = \mathrm{f}_0(q) : T^n \rightarrow SE(3) \tag{1}$$

Várias configurações das articulações podem resultar na mesma posição da ferramenta, assim, se a configuração das articulações são conhecidas e a da ferramenta não, é possível determinar a posição correta da ferramenta, mas a relação inversa precisa de outras informações para poder ser completada, e ambas podem fazer parte do cálculo do movimento.

Em alguns casos é possível utilizar controles simples como um controle proporcional-diferencial (PD) em uma única articulação para realizar o movimento e atingir o objetivo, desde que correções sejam feitas caso existam agentes externos que interfiram no controle. Também há estudos de controle proporcional-integral-diferencial (PID) para simplificar mais o problema de controle, diminuindo a necessidade de conhecimento de todos os agentes externos que interferem. O problema do controle de um robô, é representado no Diagrama 1.

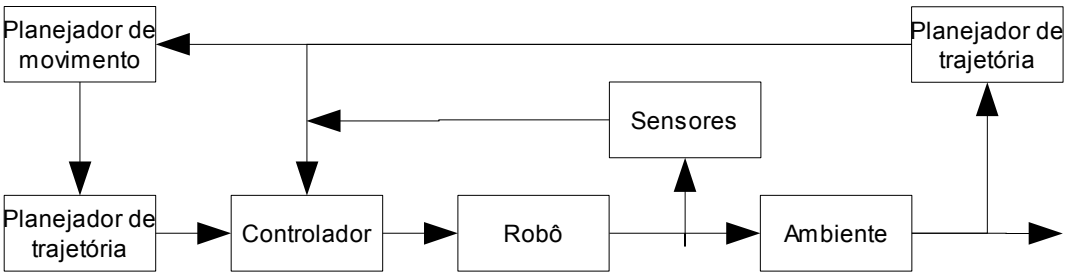


Diagrama 1: Diagrama em blocos do problema de controle de um robô.
Fonte: SPONG, p. 1341

Um estudo aprofundado do controle de braços mecânicos, com análises dos fatores e agentes que devem ser incluídos nos cálculos é apresentado por Spong, 1996 p. 1339-1351 e, no mesmo livro, outros autores que tratam de temas semelhantes em sequência.

3.2. INTELIGÊNCIA ARTIFICIAL

Inteligência artificial é a ciência que pretende imitar, com algoritmos, características de sistemas biológicos inteligentes (TATIBANA; KAETSU, 2000). Ela se desenvolve cada vez mais rapidamente devido ao aumento do poder computacional disponível.

Um dos tipos de controle inteligente são os *sistemas especialistas*, que imitam as conclusões de um especialista em um determinado assunto. Ele consiste de um banco de dados e uma máquina de inferência. A máquina de inferência percorre o banco de dados e recolhe, de acordo com valores e sequencia pré-determinados, as informações necessárias das entradas até que as possibilidades de entradas se esgotem ou que o banco de dados indique uma única conclusão para aquele conjunto de entradas, sem depender de outras.

Sistemas Fuzzy são sistemas baseados em regras que usam lógica fuzzy para representação de conhecimento e inferência. Por meio das regras ajustáveis ele determina quando um determinado valor pertence ou não a um conjunto e com base nisto exibe os resultados, podem ser usados para sensoriamento por exemplo.

Sistemas de Planejamento emulam atividades de planejamento humanas, e auxiliam na tomada de decisões.

Redes Neurais Artificiais emulam redes neurais biológicas para o processamento de informações. Seu funcionamento é baseado em somas paralelas e sequenciais de entradas ponderadas por pesos individualmente ajustados por meio de treinamento.

Algoritmos genéticos são sistemas que consideram uma população de possíveis respostas e unem estas respostas misturando-as como ocorre em reprodução genética, levando em consideração o “crossover” e “mutação”, para obter sucessivas gerações da população e produzir uma solução melhor para o problema por meio de emulação de uma “seleção natural”.

Desses mecanismos citados, todos são implementados em software, porém, é possível implementar sistemas algoritmos genéticos, fuzzy e RNA's diretamente em hardware dedicado com processamento paralelo, o que aumenta muito a capacidade do projeto se

comparado a um sistema equivalente implementado em software rodando em processadores genéricos, economizando tempo (devido à possibilidade de trabalhar muitos dados paralelamente) e energia (um processador desenvolvido especialmente para uma aplicação normalmente não necessita executar tarefas extras não relacionadas à aplicação), além de simplificar alguns aspectos do projeto (por não necessitar de partes que estariam presentes em sistemas genéricos).

Devido às características de cada técnica, cada uma tente a ser mais adequada para algumas funções. O Quadro 1 mostra controles inteligentes que podem ser usados em substituição a alguns controles convencionais.

Controle Convencional	Técnica de Controle Inteligente
Controle não linear	Fuzzy
Controle adaptativo	Fuzzy, RNA
Identificação	Fuzzy, RNA
Controle hierárquico	Fuzzy
Sistema de resposta automática	Sistema especialista
Redes de Petri	Sistema especialista
Controle ideal	Algoritmo genético, RNA

Quadro 1: Tipos de controle inteligente que podem substituir alguns tipos de controles convencionais.
Fonte: PASSINO, 1996.

Controle não linear é feito por um controlador em que a saída não depende linearmente da entrada. “Um controlador adaptativo é um controlador cujos parâmetros são ajustados continuamente para acomodar mudanças em dinâmica de processo e distúrbios” (HÄGGLUND; ÅSTRÖM, 1996, p. 824, tradução nossa). Controle hierárquico e ideal são explicados por LEWIS:

> Tal *esquemas de controles hierárquicos*, consistindo de um loop de realimentação linear interno cujo ganho é computado por uma equação quadrática externa são típicos de esquemas de controle modernos. Controle ideal é fundamentalmente um *algoritmo de modelo não-causal* que exige futuras informações sobre a planta e objetivos de desempenho. (1996, p. 763, grifo do autor, tradução nossa)

Redes de Petri são um tipo de modelagem de sistemas que possuem posições que guardam informações. As posições são interligadas por meio de transições que transformam a informação, não há ligações diretas entre duas posições ou entre duas transições. Uma transição pode receber valores de várias posições, e pode alimentar várias posições, inclusive a(s) mesma(s) de entrada (ZAMBONI, 1997).

3.3. REDES NEURAIS ARTIFICIAIS

RNA é uma estrutura formada por vários elementos (neurônios artificiais), que realizam operações matemáticas simples agrupados de forma que possam, em conjunto, operando paralelamente e sequencialmente, realizar operações complexas, com a possibilidade de ajustes de fatores que modificam seu comportamento.

O objetivo do estudo de redes neurais artificiais é criar sistemas que executam processamento característico de redes neurais naturais na realização de tarefas complexas, criando assim máquinas (não necessariamente de atuação física, mas também em tomadas de decisões e realização de análises) que desempenhem funções muito além das somente relacionadas a cálculos matemáticos simples ou comparações de valores (TATIBANA; KAETSU, 2000).

Os primeiros documentos que tratam de rede neural são atribuídos a McCulloch e Pitts, e com data de 1943 (FAUSETT, 1994). Em 1949 Donald Hebb escreveu a obra “The Organization of Behavior” (FAUSETT, 1994), em que propôs uma lei de aprendizado para os neurônios, foi o primeiro a fazer isto (CHRISTODOULOU; GEORGIOPOULOS, 2001).

Em 1958, Rosenblatt desenvolveu uma classe de RNA's comumente chamada de perceptron (FAUSETT, 1994; CHRISTODOULOU; GEORGIOPOULOS. 2001).

Em 1960, Widrow e Hoff desenvolveram um novo algoritmo para o treinamento de RNA's perceptron com uma camada. Ainda nesta década, Minsky e Papert mostraram limitações das capacidades de perceptrons de uma camada (FAUSETT, 1994).

Na década de 1970 poucas publicações foram feiras sobre redes neurais, embora alguns pesquisadores continuaram com seus trabalhos nesta área, a citar Grossberg, por um modelo de neurônio baseado em equações diferenciais não lineares e Kohonen, pelos mapas de características auto-organizáveis (FAUSETT, 1994).

Em 1982 Hopfield propõe um novo tipo de rede, que usa memória auto-associativa e a partir de 1986, Rumelhart e outros pesquisadores descobriram independentemente um algoritmo capaz de treinar redes neurais com várias camadas (FAUSETT, 1994).

A partir daí a pesquisa sobre redes neurais artificiais ganhou força e surgiram outros tipos de rede e algoritmos de aprendizado, com desempenhos aprimorados para alguns tipos de aplicações.

3.3.1. Neurônios Artificiais

Os neurônios artificiais são unidades de cálculo que guardam consigo uma pequena quantidade de informação e seguem um único modelo matemático com variações que modificam seu comportamento, mas não o princípio: um neurônio (artificial) deve fornecer um resultado (saída) dependente da soma ponderada de suas entradas. O modelo básico de um neurônio é mostrado no Diagrama 2 (TATIBANA; KAETSU, 2000).

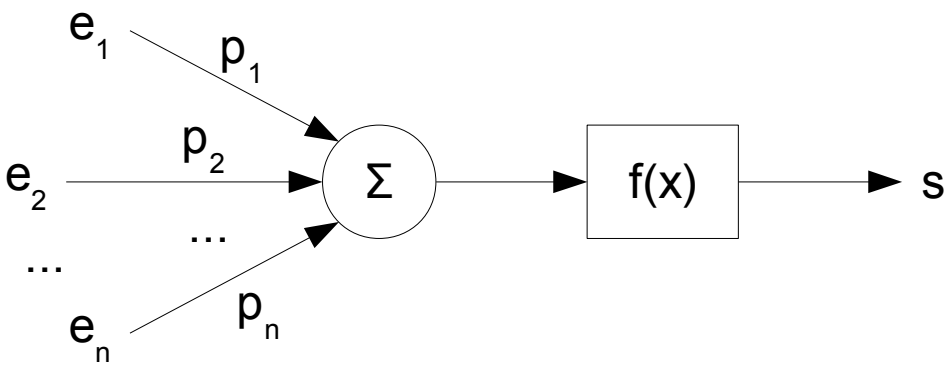


Diagrama 2: Diagrama esquemático de um neurônio artificial

O funcionamento é simples (CHRISTODOULOU; GEORGIOPOULOS, 2001; HARVEY, 1994; TATIBANA; KAETSU, 2000):

Passo 1: Os valores de cada entrada e_n são preenchidos pelas entradas da rede neural ou pelas saídas de outros neurônios que integram a rede;

Passo 2: Cada valor de entrada é ponderado com o valor do peso correspondente de cada sinapse p_n;

Passo 3: Os valores ponderados são somados na *soma*;

Passo 4: O valor obtido passa por uma função de transferência e o resultado desta função de transferência é a saída *s* do neurônio.

As funções de transferência mais comuns são as funções degrau, rampa e sigmóide. Funções lineares têm a inconveniência de tornar camadas de neurônios matematicamente agrupáveis, isto implica na possibilidade de simplificar uma rede de várias camadas em uma única camada. Múltiplas camadas neste caso seriam desnecessárias, e apenas aumentariam o custo de processamento necessário para o funcionamento. Exemplos de funções de transferência são a função degrau na equação (2) , a função rampa, na equação (3) e a função sigmóide na equação (4).

$$f_t(x) = a \cdot u(x) \tag{2}$$

$$f_t(x) = a \cdot x \tag{3}$$

$$f_t(x) = \frac{1}{1+e^{-x}} \tag{4}$$

São também comuns variações destas funções para atribuir características desejadas para a RNA sendo projetada. A função sigmóide, por exemplo, ser transformada em uma função ímpar[1] simplesmente subtraindo-se 0,5 da função original. A equação (5) representa a transformação que o neurônio executa.

$$o = f_t\left(\sum_{j=1}^{n} i_j \cdot w_j\right) \tag{5}$$

Onde n é o número de entradas, i_j são os valores das entradas, w_j são os pesos de cada entrada, f_t é a função de transferência e o é a saída.

A saída de um neurônio é sempre única, e ela pode alimentar a entrada de um ou mais neurônios ou a saída da rede, dependendo da arquitetura utilizada e da posição que ocupa o neurônio em questão.

3.3.2. Redes Neurais Naturais

Redes neurais naturais são os sistemas nervosos presente na natureza, formadas por células chamadas neurônios. Os neurônios têm a capacidade de receber sinais de sensores e outros neurônios, criarem um sinal resultante com base nos sinais recebidos e enviar adiante, para outros neurônios ou o tecido que receberá o sinal (Enc. Larousse Cultural, 1998).

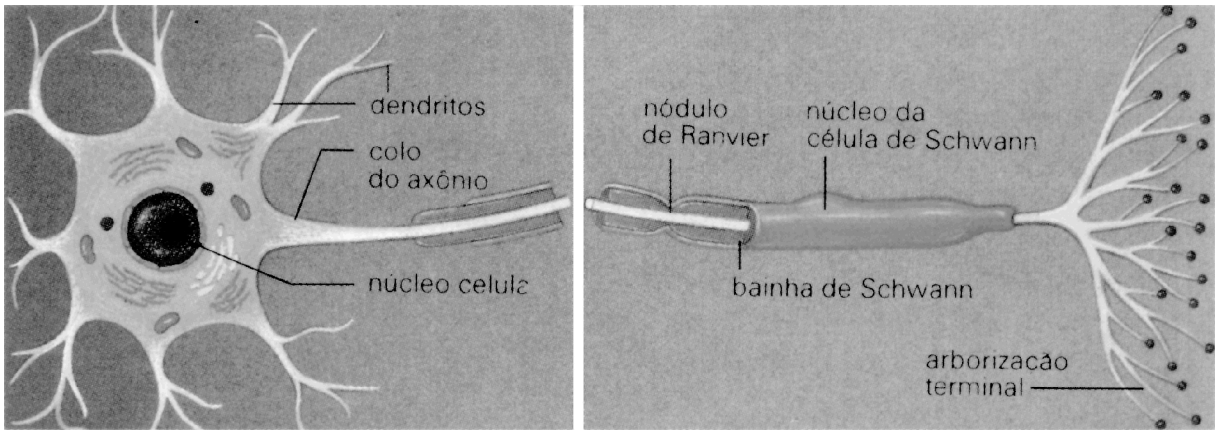
Cada neurônio é formado de um *corpo celular*, *dendritos* e um *axônio*. Os *dendritos* são prolongamentos finos da célula que têm a função de receber os sinais, o *corpo celular* contém os elementos comuns de uma célula além de outros específicos e é responsável por processar os sinais recebidos e gerar o sinal de saída, por esta razão, é também chamado de *soma*, e o *axônio* é o prolongamento principal, que recebe o sinal resultante e o conduz para outros neurônios ou às células do tecido que deve receber o sinal. A

1 Propriedade geométrica que define funções no plano que são simétricas em relação a um eixo que passa pela origem do sistema de coordenadas com inclinação de -45° em relação ao sistema de coordenadas cartesianas, isto é, se f(x) é ímpar, então f(x) = -f(-x).

conexão entre o dendrito de um neurônio e o(s) axônio(s) de outro(s) se dá por meio das *sinapses*, que são basicamente uma solução química de diversos componentes presentes nestas conexões. A sinapse em si desempenha papel importante, pois ela possui um fator adaptativo de atenuação dos sinais que passa por ela.

Os neurônios são classificados de acordo com a disposição dos prolongamentos, com a forma do corpo celular e com a origem ou destino dos sinais que ele recebe ou emite. O Desenho 2 mostra um dos tipos de neurônios conhecidos.

Os sinais são transmitidos na forma de impulsos elétricos. Normalmente os sinais são recebidos pelos dendritos e o corpo celular, processados e transmitidos ao axônio, que passa o sinal adiante na cadeia de transmissão, porém existem casos em que a condução se dá em sentido oposto, principalmente nos dendritos (circuitos locais). Excitabilidade, condução e mediação química são propriedades específicas de cada neurônio.



Desenho 2: Estrutura de um neurônio multipolar.
Fonte: NEURÔNIO, Grande Enc. Larousse Cultural, 1998, vol. 17, p. 4194.

3.3.2.1. RNA's Comparadas com Redes Neurais Naturais

Uma das comparações mais famosas que existe entre homem e máquina é o teste de Turing, segundo o qual se uma máquina pode responder a perguntas de forma que não se possa distinguir da resposta de um ser humano, então esta máquina apresenta inteligência.

Porém, desenvolveram-se programas que podem responder a uma quantidade muito grande de perguntas de forma a não parecer um sistema automático e que não possuem algoritmos que lhe atribuam inteligência (KOCH, 2008). Mesmo perguntas não previamente programadas recebem respostas genéricas de forma que não se percebe facilmente que se trata de um sistema automático com saídas pré-definidas. Não se tratam, no entanto, de sistemas

inteligentes. Nota-se, portanto, que para que o teste seja eficiente, as perguntas devem ser muito bem elaboradas.

Apesar dos grandes avanços, a inteligência artificial (não somente a parte de RNA) tem resultados muito distantes dos possíveis por meio dos resultados obtidos em sistemas naturais. Os sistemas artificiais se limitam a realizar operações relativamente simples, embora nem sempre de fácil execução e algumas vezes cansativas ou mesmo impossíveis de se realizar.

A tarefa de reconhecer um rosto conhecido em uma cena qualquer, para uma pessoa, leva não mais que alguns décimos de segundo, enquanto para um computador pode levar dias, dependendo da complexidade de detalhes necessários para identificar o objeto. Porém reconhecer caracteres em placas de todos os veículos que passam em uma grande avenida expressa para analisar possíveis infratores como no caso da cidade de São Paulo, que tem restrições de circulação em horários determinados na semana de acordo com a placa do veículo, e registrar todas as que se enquadram na infração é uma tarefa que uma pessoa comum provavelmente não poderia executar, mas um sistema de inteligência artificial é muito eficaz nesta função.

3.3.2.2. O Cérebro Humano

As capacidades de processamento de uma rede neural natural como o cérebro humano estão muito além do conhecimento atual. Embora se possa explicar que uma rede neural aprende a receber os sinais dos sensores, processar e enviar comandos, não é possível explicar como surgem outras variáveis ao processamento, como a consciência, no caso dos humanos e, provavelmente, em outros animais. No momento, ninguém realmente sabe exatamente o que consciência é (KOCH, 2008).

Esta falta de conhecimento está muito relacionada à dificuldade em fazer uma observação direta, minuciosa e precisa durante o funcionamento e à grande complexidade presente.

Além de cada cérebro ser único, eles constantemente se alteram em resposta a novas experiências, e mesmo estimulando um cérebro com entradas exatamente iguais, o conjunto de sinais de saída (considera-se que há muitos sinais de saída) nunca se repetirá (HORGAN, 2008).

Outro fator que eleva a complexidade é a quantidade de neurônios e conexões. Estima se que o cérebro de um homem adulto tenha cerca de 100 bilhões de neurônios, sendo

que cada um pode se conectar a 100 mil outros. O Quadro 2 contém comparação de algumas características de um cérebro humano e de um computador.

	Cérebro humano	**Computador**
Elemento processador	Neurônio	Transistor
Quantidade de elementos processadores	10^{11} elementos (estimado)	Até $3*10^{8}$ elementos
Base de tempo de operação	milisegundos	nanosegundos
Modo de operação	Paralelo	Sequencial, normalmente 1 a 4 operações simultâneas
Ligações entre elementos processados	Até 10^{5}	< 10

Quadro 2: Comparação de algumas características do cérebro humano e de um computador.
Fonte: TATIBANA; KAETSU, 2000.

3.3.3. Tipos de RNA

Representado como um elemento, o neurônio artificial é uma estrutura bem simples capaz de calcular uma saída com operações em seus valores de entrada, mas evidentemente, a capacidade de processamento é bem limitada. Para aumentar a capacidade de treinamento, estes neurônios são organizados em redes que permitem aumentar a complexidade e, consequentemente, a capacidade de processamento.

De acordo com características dos neurônios utilizados e a organização deles, as RNA's recebem diferentes classificações, algumas das quais são utilizadas para resolver tipos de problemas específicos .

Como não há uma regra se a rede neural é física ou apenas um software, não há necessariamente um espaço físico ocupado, a única coisa definida é a transmissão de sinal das entradas da rede para os neurônios, e das saídas dos neurônios para as saídas da rede, sendo que dependendo do tipo de rede os sinais podem passar por vários neurônios que os transformam. Deste modo, a organização que se leva em consideração é a do caminho percorrido pelos sinais durante o processamento, ou, escrita de outra forma, como é feito o processamento do sinal.

3.3.3.1. Perceptron

Também chamado de perceptron de camada única (single layer perceptron – SLP), este tipo de rede foi criado por Frank Rosenblatt em 1958 (FAUSETT, 1994) e é uma rede bem simples de apenas duas camadas, uma de entrada (que simplesmente recebe o valor do ambiente externo) e uma de saída (que possui os neurônios que realizam as operações), sendo que cada valor da camada de entrada é conectada a todos os neurônios da camada de saída, a função de transferência é a função degrau, e cada neurônio calcula uma das saídas de sua camada, assim, o número de saídas é igual ao número de neurônios.

Os dados seguem sempre para frente, entram pela primeira camada (apenas coleta de valores, sem processamento), passam pelos neurônios e saem, não há outro caminho possível, como ilustrado no Diagrama 3, que possui uma rede Perceptron de quatro entradas e duas saídas. Este tipo de rede trata problemas de lógica binária de operações simples como AND e OR, mas não pode realizar operações XOR. Pode ser usada para propósitos de classificação.

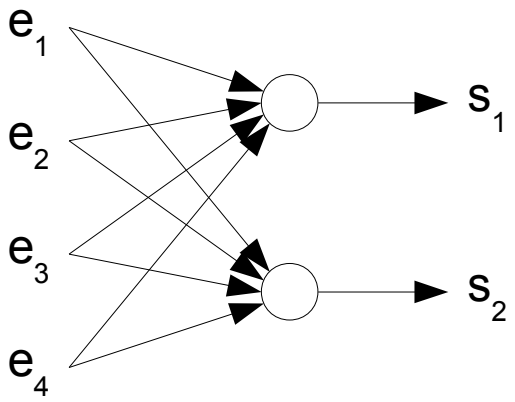


Diagrama 3: Exemplo de rede SLP

3.3.3.2. Adaline/Madaline

Desenvolvida por Widrow e Hoff em 1960, a rede Adaline (acrônimo de Adaptative Linear Neuron – Neurônio linear adaptativo) é composta de um único neurônio, tipicamente usa entradas bipolares (1 e -1) para suas entradas e saídas desejadas (apesar de não estar restrito a estes valores), possui uma entrada bias, de valor 1 com um peso também ajustável (FAUSETT, 1994).

A principal contribuição desta rede foi o algoritmo de treinamento, chamado de regra delta, que consiste em encontrar o conjunto de pesos que resultam no menor erro quadrático médio do sistema para todo o conjunto de dados de treinamento (CHRISTODOULOU; GEORGIOPOULOS, 2001). Depois de treinadas, se a saída da rede for bipolar, uma função de *threshold* é aplicada.

Várias redes adaline podem ser organizadas paralelamente, de forma que produzam várias saídas, o tratamento de processamento e treinamento neste caso é particular para cada uma delas.

Quando redes adaline são combinadas de forma que as saídas são ligadas às entradas de outras redes adaline, o conjunto passa a ser chamado de madaline (acrônimo de **m**any **adaline,** muitos adaline) (FAUSETT, 1994).

3.3.3.3. Perceptron Multi-Camada

Apesar de existir desde a década de 1960, este tipo de rede não possuía um algoritmo capaz de executar o treinamento (CHRISTODOULOU; GEORGIOPOULOS, 2001), a arquitetura é semelhante à do perceptron, mas contém uma ou mais camadas "escondidas" ou "ocultas" entre as camadas de entrada e a de saída, e os neurônios podem usar funções lineares ou sigmóides no lugar da função degrau.

Esta nova rede permitiu a execução de funções mais complexas incluindo processamento semelhante a combinações de várias funções lógicas e é comumente chamada de perceptron multi-camada (multi-layer perceptron – MLP). O Diagrama 4 é uma representação de uma MLP com duas entradas, quatro neurônios na camada oculta (única) e dois neurônios na camada de saída.

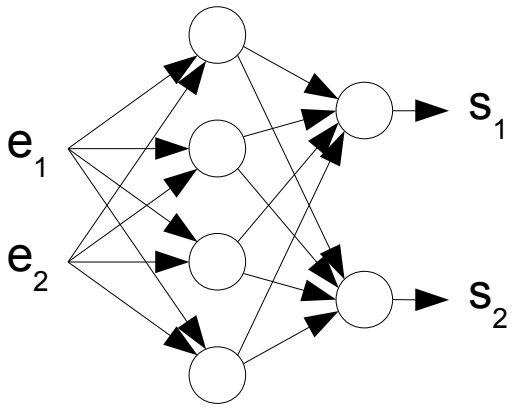


Diagrama 4: Exemplo de rede MLP

No tipo padrão de conexão de uma MLP, todas as saídas de uma camada são conectadas à entrada da outra camada, nenhuma conexão pode ser feita entre neurônios da mesma camada ou da saída de neurônios de uma camada qualquer e a entrada de neurônios de camadas anteriores. Pode haver um neurônio chamado de bias para cada camada, que não possui entradas, e cuja saída é constante, normalmente um valor unitário.

3.3.3.4. Redes Neurais Recorrentes

Redes neurais recorrentes são redes que tem conexões entre neurônios que formam ciclos pelo qual a informação pode ficar circulando, criando assim uma memória dos estados passados.

A rede Hopfield, um exemplo de rede recorrente, foi criada em 1982 por J.J. Hopfield (FAUSETT, 1994), esta rede tem a saída de cada neurônio conectada às entradas de todos os outros neurônios da rede. Não há distinção entre os neurônios, exceto que os neurônios de entrada possuem uma entrada a mais em relação aos outros neurônios, entrada esta que vem de fora da rede, e os neurônios de saída têm a saída ligada a um leitor externo à rede conforme o Diagrama 5. O principal uso deste tipo de redes é o reconhecimento de imagens.

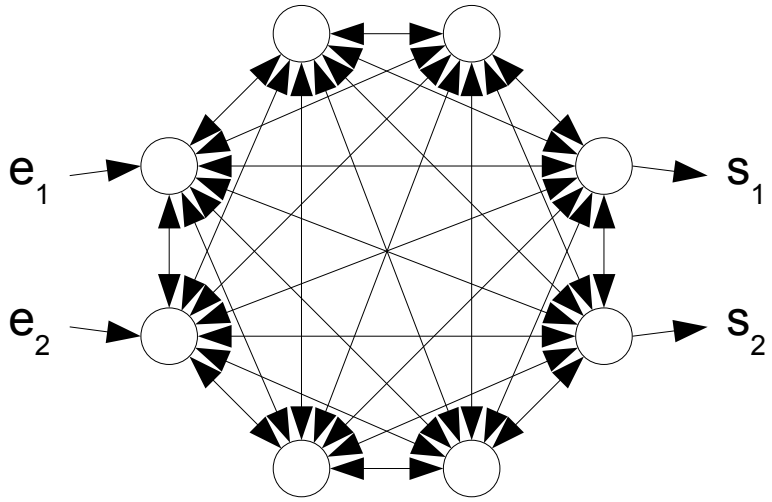


Diagrama 5: Exemplo de rede Hopfield

3.3.3.5. Outros tipos de Redes Neurais

Existem vários outros tipos de redes neurais, como as redes ARTMAP e as redes Kohonen (FAUSETT, 1994; CHRISTODOULOU; GEORGIOPOULOS, 2001), cada

uma possui características particulares de estrutura e processamento que conferem a elas algumas vantagens para se trabalhar com alguns tipos de informações.

3.3.4. Treinamento

Para a criação e utilização prática de uma RNA qualquer, é necessário que exista um algoritmo por meio do qual ela possa ser treinada para adquirir o comportamento desejado em função das entradas.

Normalmente, quando uma RNA é criada pela primeira vez, os pesos de seus neurônios, que podem ser considerados os fatores responsáveis pelo comportamento e/ou o conhecimento, são definidos com valores aleatórios, de forma que não representam, a princípio, um comportamento definido. Estes valores devem ser alterados a fim de garantir à RNA a capacidade de processar as informações e exibir os resultados corretamente.

Em qualquer deles, o conjunto de entradas deve ser suficientemente numeroso para que a RNA adquira a abstração necessária para processar toda a faixa de entradas possível, porém uma quantidade excessiva pode fazer o treinamento ficar muito demorado.

Outro fator importante é a quantidade de vezes que a rede é treinada, se os mesmos conjuntos de treinamento são apresentados demasiadamente à RNA, ela pode ficar "viciada", e perder a abstração desejada, acabando por se comportar de forma inadequada a outras entradas.

3.3.4.1. Métodos de aprendizagem

Método de aprendizagem refere-se à maneira que o treinamento é feito e não ao algoritmo utilizado. A decisão por qual método utilizar depende das características do sistema, incluindo a RNA como a disponibilidade de dados adequados, a possibilidade de gerar estes dados ou o custo de se fazer isto, o tipo de RNA utilizada ou outras particularidades. A seguir estão descritos alguns dos métodos de aprendizagem utilizados (CHRISTODOULOU; GEORGIOPOULOS, 2001).

3.3.4.1.1. Aprendizagem Supervisionada

A principal característica deste método de aprendizagem é a presença de um elemento externo, um "professor". A RNA, a princípio, não possui nenhum conhecimento do ambiente em qual está operando, mas sim o professor.

Para cada conjunto de entradas apresentado à rede, o conjunto de respostas desejadas é definido pelo professor e a rede é treinada para que as saídas se aproximem deste conjunto. O conjunto de respostas pode estar armazenado em uma lista ou ser gerado conforme a necessidade.

3.3.4.1.2. Aprendizagem Não-supervisionada

Na aprendizagem não supervisionada, não existe a presença de um "professor", deste modo, não há a apresentação, para a RNA, de um conjunto de respostas desejado para cada conjunto de entrada. Para o treinamento, se leva em consideração a media da qualidade da representação das saídas apresentadas pela RNA, e o treinamento é feito para que esta medida seja a melhor possível.

3.3.4.1.3. Aprendizagem Híbrida

Existem casos em que a associação dos dois tipos de aprendizagem acima pode ser utilizada em conjunto para melhorar a eficiência do treinamento. Se o conjunto de dados de treinamento é muito grande, a aprendizagem supervisionada em uma rede MLP torna-se muito demorado, como dizem CHRISTODOULOU e GEORGIOPOULOS: "as the training data set increases in size, the training time of the multilayer perceptron increases exponentially" (2001, p. 22).

Para estes casos, se pode utilizar uma estrutura de RNA que aceite uma aprendizagem não supervisionada para resolver parte do problema e posteriormente uma estrutura que aceite uma aprendizagem supervisionada (como a MLP) para concluir o processamento.

Como exemplo se pode citar o treinamento para muitas versões de dígitos numéricos manuscritos, para reduzir o tempo de treinamento, é possível fazer, com uma RNA que aceite uma aprendizagem não-supervisionada, um agrupamento dos conjuntos de entrada e, posteriormente, classificar os dados agrupados com uma MLP.

3.3.4.1.4. Aprendizagem de Reforço

Este tipo de aprendizagem consiste em buscar um mapeamento de estados do ambiente com as ações a serem tomadas. Ao contrário dos outros métodos, os conjuntos de treinamento de entrada/saída não são apresentados, e as saídas não são corrigidas. O foco reside em desempenho, que envolve comparar exploração de novas ações e comparação com o conhecimento atual.

Este tipo de aprendizagem considera que, partindo de um estado S(t) e depois aplicando uma ação A(S(t)), chega-se a um estado S(t+1) e procura maximizar uma somatória de recompensas dos estados. Esta soma é feita para vários estados futuros, o melhor conjunto de ações resulta no maior valor da somatória.

3.3.4.2. Tarefas de aprendizagem

Tarefa de aprendizagem é a ação executada pelo treinamento para que a rede se aproxime do comportamento desejado. A seguir são citadas cinco das mais comuns técnicas utilizadas (CHRISTODOULOU; GEORGIOPOULOS, 2001). Embora algumas delas possam ser consideradas similares a outras, existem estudos separados de cada uma e de sua importância no estudo de RNA's, desta forma, são apresentadas separadamente.

3.3.4.2.1. Aproximação

Supondo que, em um dado intervalo, as saídas possam ser calculadas em função das entradas com uma função não linear, possivelmente desconhecida, descrita pela equação (6) a seguir, esta tarefa consiste em fazer com que uma RNA passe a simular esta função com base na apresentação de um conjunto finito de entradas e saídas desejadas contidas no intervalo. Esta tarefa é apropriada para aprendizagem supervisionada.

$$y = \mathrm{f}(x) \tag{6}$$

Onde, dentro de um intervalo, y é a saída desejada para cada entrada x correspondente.

3.3.4.2.2. Associação

Pode ser autoassociação ou heteroassociação. Na autoassociação, a RNA é treinada com um conjunto de padrões apresentado, posteriormente, se apresenta uma versão distorcida ou incompleta do padrão supostamente gravado, e o padrão correto deve ser recuperado. A heteroassociação funciona de forma similar, porém, os padrões desejados de saída diferem dos padrões de entrada, ou seja, quando o padrão A qualquer é apresentado à RNA, deseja-se que a saída seja o padrão B.

3.3.4.2.3. Classificação de padrões

Neste tipo de tarefa de aprendizado, existe um número fixo de classificações que um conjunto de entradas pode receber. Para o treinamento, apresenta-se repetitivamente à RNA o conjunto de padrões de entrada com sua respectiva classificação. Quando, depois de treinada a rede recebe como entrada um padrão qualquer, ela deve classificá-lo de entre os conjuntos disponíveis. Esta tarefa é adequada para treinamento supervisionado. Com esta tarefa de aprendizagem, as RNA's podem assumir limites não lineares entre as diferentes classificações e deste modo pode resolver problemas complexos de classificação de padrões.

3.3.4.2.4. Predição

Consiste em treinar uma RNA com uma série de valores passados já conhecidos para que ela continue a gerar valores que devem ser continuação da série apresentada.

3.3.4.2.5. Agrupamento

Consiste em classificar os conjuntos de entradas em grupos com características similares. Os grupos, a princípio, não são conhecidos, eles são formados de acordo com as similaridades existentes entre os conjuntos de entrada. Esta tarefa de aprendizado é aplicada a RNA's com treinamento não-supervisionado.

3.3.4.3. Algoritmos de treinamento

Junto com o surgimento das RNA's, apareceu a necessidade de desenvolver algoritmos por meio dos quais elas pudessem ser treinadas para que passassem a realizar os processamentos desejados.

Os algoritmos que serão apresentados são adequados para treinar redes do tipo SLP e MLP, existem outros algoritmos, tanto para o treinamento destes tipos de rede quando para treinamento de outros tipos, mas não serão discutidos.

3.3.4.3.1. Algoritmo de treinamento do Perceptron

Todos os neurônios de uma SLP ficam paralelamente na mesma camada, e não há nenhuma ligação entre eles, desta forma, a resposta de cada um não depende de nenhum outro e, consequentemente, o treinamento também não. A função de transferência é a função degrau resultando na equação (7) para os neurônios.

$$o = \mathrm{u}\left(\sum_{j=1}^{n} i_j \cdot w_j\right) \tag{7}$$

O treinamento é aplicado para cada neurônio separadamente, para cada grupo de entradas do conjunto de treinamento, por meio do seguinte procedimento (CHRISTODOULOU; GEORGIOPOULOS, 2001, p.42-43):

Passo 1: Inicializar a rede neural com valores aleatórios pequenos para todos os pesos;

Passo 2: ler o primeiro grupo de entradas do conjunto de treinamento;

Passo 3: calcular a saída de cada neurônio, como a rede é a perceptron, com a equação (7);

Passo 4: para cada neurônio, se a saída o for igual à saída desejada y (do conjunto de treinamento), ir para o passo 5, do contrário, somar um fator de correção a todos os pesos do neurônio definido pela equação (8), em que η é uma taxa de treinamento que deve ter um valor adequado de acordo com características do sistema e da RNA;

$$\Delta w_j = \eta \cdot (y - o) \cdot i_j \tag{8}$$

Passo 5: se todos os grupos do conjunto de treinamento foram lidos, pular para o passo 6, do contrário, ler o próximo grupo de entradas do conjunto de treinamento e pular para o passo 3;

Passo 6: Se nenhum peso foi alterado, o treinamento está concluído, do contrário, pular para o passo 2.

A taxa de treinamento depende da rede e do sistema a que ela é aplicada, um valor muito baixo pode fazer o treinamento ficar muito demorado e um valor muito alto pode fazer com que os pesos fiquem oscilando de forma que o treinamento não seja capaz de fazer a RNA convergir para o comportamento desejado.

3.3.4.3.2. Regra Delta

Este treinamento foi desenvolvido com a rede Adaline que, em relação a uma rede Perceptron, os neurônios utilizam uma função rampa ao invés de uma função degrau. Os passos para treinamento são iguais aos do treinamento do Perceptron, exceto que no passo 3, os neurônios executam a equação (9) e que, no passo 6, o treinamento acaba quando uma condição que determina o sucesso do treinamento é satisfeita (caso contrário retornar ao passo 2 como no treinamento do perceptron) (CHRISTODOULOU; GEORGIOPOULOS, 2001, p. 57).

$$o = \sum_{j=1}^{n} i_j \cdot w_j \tag{9}$$

A condição para conclusão do treinamento não é pré-definida, mas pode ser, por exemplo, um erro quadrático médio de todos os dados de treinamento ser menor que um valor determinado.

3.3.4.3.3. Retro-propagação

Este algoritmo foi publicado em 1986 por Rumelhart, Hinton e Williams (FAUSETT, 1994). Apesar das redes MLP terem surgido na década de 1960, este foi o primeiro algoritmo criado capaz de treiná-las (CHRISTODOULOU; GEORGIOPOULOS, 2001).

Para este algoritmo de treinamento ser aplicável, o fluxo de informação da rede deve ser sempre para frente, nunca para trás ou entre elementos da mesma camada. Além disso, a função de transferência deve ser diferenciável.

Uma função de transferência linear também é indesejável, pelo menos nas camadas ocultas, pois a RNA poderia ser reduzida para uma única camada e se tornaria igual a uma rede Adaline. O treinamento também ficaria igual à regra delta, como pode ser deduzido das equações que são apresentadas mais abaixo, apenas lembrando-se que a derivada de uma função como a equação (10) é simplesmente a constante ^{a}

$$\mathrm{f}(x) = a \cdot x \tag{10}$$

O procedimento para este treinamento é (CHRISTODOULOU; GEORGIOPOULOS, 2001, p73-75):

Passo 1: Inicializar a rede neural com valores aleatórios pequenos para todos os pesos;

Passo 2: ler o primeiro grupo de entradas do conjunto de treinamento;

Passo 3: para cada camada, calcular a saída de cada neurônio por meio da equação (11), onde g*(x)* é a função de transferência utilizada, i_j são as entradas da respectiva camada e w_j são os pesos do neurônio para cada entrada;

$$o = \mathrm{g}\left(\sum_{j=1}^{n} i_j \cdot w_j \right) \tag{11}$$

Passo 4: para cada saída da rede, se a saída *o* for igual à saída desejada *y* (do conjunto de treinamento), ir para o passo 7, do contrário, ir para o passo 5;

Passo 5: calcular um termo de erro (δ) para cada um dos neurônios da camada de saída definido pela equação (12) e posteriormente os termos de erro para os neurônios das camadas anteriores usando a equação (13) onde *k*, *m*, w^{+1} e δ^{+1} são o índice, a quantidade de neurônios, os pesos e os termos de erro, respectivamente, todos da camada posterior. Como os termos de erro são calculados do final para o início (daí o nome retro-propagação), os erros se propagam dos neurônios de saída até os neurônios de entrada se somando aos neurônios vizinhos da mesma camada, assim, os erros de uma camada são os erros ponderados da camada posterior, o que faz sentido, já que a saída desta camada influenciará os valores de todos os neurônios da camada seguinte.

$$\delta = g'\left(\sum_{j=1}^{n} i_j \cdot w_j\right) \cdot (y - o) \tag{12}$$

$$\delta = g'\left(\sum_{j=1}^{n} i_j \cdot w_j\right) \cdot \left(\sum_{k=1}^{m} w_k^{+1} \cdot \delta_k^{+1}\right) \tag{13}$$

Passo 6: aplicar as variações de correções de erros a todos os pesos de todos os neurônios da rede conforme a equação (14), onde j é o índice da entrada, η é a taxa de treinamento e i_j são as respectivas entradas de cada peso.

$$\Delta w_j = \eta \cdot \delta \cdot i_j \tag{14}$$

Passo 7: se todos os grupos do conjunto de treinamento foram lidos, pular para o passo 8, do contrário, ler o próximo grupo de entradas do conjunto de treinamento e pular para o passo 3;

Passo 8: Se a condição que determina o sucesso for satisfeita, o treinamento está concluído, do contrário, pular para o passo 2.

Assim como no treinamento do perceptron, a taxa de treinamento deve ser adequada, pois um valor muito pequeno pode tornar o treinamento muito demorado e um valor muito grande pode fazer com que os pesos fiquem oscilando em torno do valor de mínimo erro sem, de fato, convergir.

3.3.5. Vantagens Esperadas da Utilização de RNA's para Controle de Braços Mecânicos

Sendo as fases de desenvolvimento e aplicação de um sistema de controle conforme o Diagrama 6, a utilização de RNA's deve beneficiar primeiramente as fases de projeto e implementação do sistema, provavelmente a fase de manutenção e possivelmente (mas não necessariamente) a fase de funcionamento.

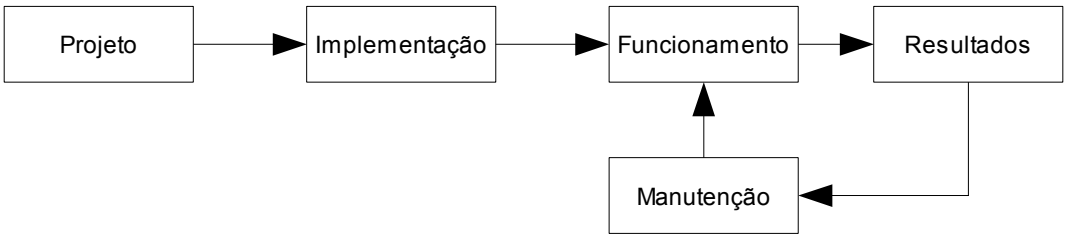


Diagrama 6: Etapas de desenvolvimento e aplicação de um sistema de controle.

O benefício direto é que o sistema necessita menos desenvolvimento de cálculos e manipulações matemáticas para seu funcionamento, o que reduz o risco de problemas relacionados a falhas de projeto ou mesmo erros no próprio projeto ou na implementação, provenientes da complexidade de métodos convencionais, além de possivelmente tornar viáveis operações que antes não eram.

As RNA's, devido à sua natureza, possuem os seguintes benefícios (CHRISTODOULOU; GEORGIOPOULOS, 2001):

Não linearidade – se os elementos que formam a RNA forem não-lineares, a mesma será, consequentemente, não-linear, o que a torna mais abrangente na solução de problemas, tanto lineares quando não-lineares.

Mapeamento de entrada-saída – O funcionamento de uma RNA é baseado em criar um comportamento do processamento dos dados que possa reproduzir as transformações das entradas nas saídas, de acordo com os conjuntos de treinamento, que devem ser suficientemente genéricos para que quando entradas que não fazem parte do conjunto de treinamento são apresentadas, a rede possa calcular saídas adequadas.

Adaptatividade – Enquanto uma RNA estiver soba a ação de um agente de treinamento, ela pode ser modificada para aceitar entradas com características diferentes das que já foram treinadas, assim aumentando a abrangência dos tipos de tratamentos.

Tolerância a falhas – O "conhecimento" em uma RNA é distribuído pelos seus elementos, desta forma, considerando uma RNA propensa a falhas (implementada em hardware por exemplo), se uma parte dos elementos relativamente pequena em relação ao tamanho da RNA falhar, a degradação dos resultados deverá ser também pequena, embora outros fatores, como a importância dos elementos que sofreram a falha, possam contribuir para que esta degradação seja grave.

Implementabilidade VLSI[2] - A natureza de processamento paralelo de uma RNA permite que seja implementada em hardware especializado, o que aumentaria a eficiência em relação à capacidade de processamento e ao consumo de energia.

Uniformidade de análise e projeto – Não importa em que sistema uma RNA será implementada, as funções dela serão sempre as mesmas: receber treinamento, coletar informações, tratar e fornecer as saídas, eventualmente o treinamento pode se repetir conforme necessário. Os elementos são sempre neurônios, e até os algoritmos de treinamento são os mesmos para qualquer aplicação. Determinadas aplicações podem ser tratadas de forma

2 Very Large Scale Integration – Técnica de fabricação de circuitos integrados que permite a integração de milhares de transistores em um único chip.

mais eficiente por um dos algoritmos disponíveis, mas não é necessário criar um algoritmo para tratar uma aplicação específica.

Analogia neurobiológica – A inspiração para o desenvolvimento de RNA's é criar um sistema que possa processar informações assim como o cérebro, que é a maior prova de que redes neurais são possíveis, eficientes e tolerantes a falhas.

4. DESENVOLVIMENTO DE REDE NEURAL PARA USO EM PROGRAMAS DE COMPUTADOR

Antes de iniciar o desenvolvimento de qualquer programa ou biblioteca em computadores, devem ser tomadas algumas decisões, entre elas a linguagem a utilizar, as ferramentas e o que exatamente será desenvolvido. Se houverem dúvidas quanto aos termos utilizados, o glossário, que se encontra ao final deste trabalho, pode ser consultado.

4.1. AVALIAÇÃO DA LINGUAGEM DE PROGRAMAÇÃO E DA INTERFACE DE DESENVOLVIMENTO

O código desenvolvido deve permitir que a estrutura da rede seja definida somente quando for utilizada e que a rede possa ser destruída e em seguida, recriada, com as mesmas características ou características diferentes. Além disso, deve contar com possibilidades de alterações no código para adição ou modificação de funcionalidades sem que isto interfira na compatibilidade com códigos já existentes e reutilização de código para outros desenvolvimentos.

Para atender a estas necessidades, a linguagem escolhida é orientada a objeto, o que permite, além das exigências acima, a definição de estrutura de objetos para posteriores implementações distintas de procedimentos compatíveis, a construção de componentes baseados nos já existentes por meio de herança e a utilização de componentes já escritos para desenvolver novos componentes que os utilizam internamente.

O uso de matrizes para o desenvolvimento da RNA provavelmente resultaria em um uso mais eficiente de memória e de poder computacional, mas os parâmetros dos elementos ficariam distribuídos em matrizes e a estrutura ficaria difícil de entender e a flexibilidade de alterações seria reduzida.

Das linguagens orientadas a objeto, as mais conhecidas e utilizadas são o C++, o Java e o Object Pascal. O Java utiliza grande quantidade de memória, e não tem código compilado verdadeiro, tendo seu funcionamento intermediado por uma máquina virtual, a Java virtual machine (JVM, máquina virtual Java). O C++ e o Object Pascal são compiladas para código de máquina e ambas são atualmente portáveis para vários sistemas.

Para a utilização dos componentes a serem criados, deseja-se um integrated development environment (IDE, ambiente de desenvolvimento integrado) focado em rapid application development (RAD, desenvolvimento rápido de aplicação), com fácil montagem

de componentes visuais e que possua compatibilidade, no mínimo com o sistema operacional Windows e Linux e melhor se compatível também com outros sistemas.

O C++ é muito mais utilizado e tradicional, porém, não foi encontrado para ele um IDE suficientemente estável e comum a vários sistemas, o que pode causar necessidade de adequação de código quando se movendo entre sistemas operacionais.

O Object Pascal possui o Lazarus[3], que funciona nos sistemas operacionais mais comuns e pode usar o mesmo código fonte para gerar programas com a mesma funcionalidade em qualquer um deles, além de apresentar uma interface de fácil criação de aplicações gráficas.

A escolha do Object pascal no IDE Lazarus se mostrou a mais adequada para o desenvolvimento. Além disso, se houver futuro interesse, o código poderá facilmente ser modificado para funcionar no Delphi[4].

Como a criação do código fonte de todos os componentes da RNA é explicado, com um pouco conhecimento de programação é possível, também, traduzir o código para outras linguagens como o C++.

4.2. AVALIAÇÃO DO TIPO DE RNA A SER IMPLEMENTADA

Deseja-se que a RNA controle o movimento de um braço mecânico, incluindo sentido, direção e velocidade. As entradas serão alimentadas com valores normalizados de sensores e poderão ser positivas ou negativas. As saídas poderão ser positivas ou negativas e, após desnormalizadas, alimentarão os atuadores. Para isto, o processamento é feito com valores contínuos (certamente terão precisão limitada, mas a limitação será desprezível).

A rede neural escolhida para atender as necessidades descritas foi a MLP com neurônios com função sigmóide e treinamento do tipo retro-propagação com pequenas adaptações.

Como tanto as entradas quanto as saídas podem ter valores positivos ou negativos, a função sigmóide sofreu adaptação. Originalmente a faixa de valores resultantes de uma função sigmóide, representada na equação (15) é de 0 a 1 para entradas de $-\infty$ a $+\infty$, respectivamente. Subtraindo-se 0,5, a função passa a ter simetria ímpar, o que a torna mais

3 IDE focado em RAD com licensas de uso GPL e LGPL e baseado no compilador fpc. Outras informações bem como cópias do produto podem ser obtidas em <http://www.lazarus.freepascal.org/>.

4 IDE focado em RAD originalmente desenvolvido e distribuido pela Borland e atualmente pela Embarcadero, mais informações podem ser obtidas em <http://www.embarcadero.com/products/delphi>.

adequada para o uso neste caso. A faixa resultante da função modificada é -0,5 a 0,5. Para deixar os cálculos mais fáceis, depois de subtraído 0,5, a função foi multiplicada por 2, para que a faixa de saída seja -1 a 1, resultando na equação (16).

$$f(x)=\frac{1}{1+e^{-x}} \tag{15}$$

$$f(x)=\frac{2}{1+e^{-x}}-1 \tag{16}$$

Para utilização do algoritmo retro-propagação, é necessário conhecer a derivada da função de transferência, equação (17).

$$f'(x)=\frac{\partial f(x)}{\partial x}=\frac{\partial \frac{2}{1+e^{-x}}-1}{\partial x}=2\cdot\frac{\partial \frac{1}{1+e^{-x}}}{\partial x}=2\cdot e^{-x}\cdot\frac{1}{(1+e^{-x})^2}=\frac{2\cdot e^{-x}}{(1+e^{-x})^2} \tag{17}$$

Se o resultado da primeira equação, f(x) já é conhecido, é possível escrever f '(x) em função de f(x) e eliminar uma parte dos cálculos necessários para descobrir o valor de f '(x). Uma equação normalmente representa um custo muito reduzido para compensar a realização desta manipulação matemática, mas no caso de uma rede neural, que, para cada vez que a rede é treinada com um único conjunto de entradas e saídas a equação deve ser calculada uma vez para cada neurônio, este custo se torna elevado e esta manipulação passa a fazer sentido, uma vez que a variável independente teria que ser calculada novamente.

Como foi adicionada uma constante de simetria, a função f '(x) não é calculada em função de f(x), mas de g(x), que não possui a constante de simetria, com a qual este cálculo seria mais complexo, e está representada na equação (18).

$$g(x)=f(x)+1=\frac{2}{1+e^{-x}} \tag{18}$$

Depois de derivar, coloca-se o 2 em evidência, fora da divisão, para que o termo $1+e^{-x}$ apareça acima da divisão, somar (1-1) no numerador, após isto, separa-se a divisão em uma subtração, simplifica-se a primeira parte da subtração com o numerador $1+e^{-x}$, devolve-se o 2 para as divisões e adapta-se a segunda divisão com multiplicação e divisão por 2, e o resultado poderá ser escrito em função apenas de números e da equação g(x). Todo este procedimento está feito a seguir, passo a passo, e o resultado está na equação (19).

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

$$f'(x)=2\cdot\left(\frac{1+e^{-x}}{(1+e^{-x})^2}-\frac{1}{(1+e^{-x})^2}\right)=2\cdot\left(\frac{1}{1+e^{-x}}-\frac{1}{(1+e^{-x})^2}\right)=\frac{2}{1+e^{-x}}-\frac{\frac{4}{(1+e^{-x})^2}}{2}$$

$$f'(x)=g(x)-\frac{(g(x))^2}{2} \tag{19}$$

Desta forma, utiliza-se o valor da saída do neurônio (armazenada) somada a 1 para se obter o valor da derivada da função de transferência para a mesma entrada que foi calculada anteriormente.

4.3. DESENVOLVIMENTO DO CÓDIGO FONTE

Os neurônios são agrupados em camadas e as camadas agrupadas em uma RNA, cada objeto de nível superior faz os comandos aos objetos filhos com base nos comandos que recebem, desta forma o gerenciamento dos objetos criados ficam mais simples e confiável.

São implementadas três classes de objetos, *neuron* (neurônio), *layer* (camada) e *nn* (neural network, rede neural), e declarados tipos auxiliares para padronizar variáveis, tornar possível futuras alterações nos tipos e consequentemente na precisão e no uso de memória com mínima alteração no código fonte e utilizar alocação dinâmica de memória para conjuntos de números.

No caso do Object Pascal, as três classes podem ser escritas em um único arquivo pelo objetivo das três ser a criação da rede neural. O nome sugerido para a unidade é *UNN*, e o nome do arquivo, *unn.pas*. Alguns conhecimentos e práticas de programação em Object Pascal (verificar o APÊNDICE A) podem ser de grande valor para a criação de código fonte em Object Pascal ou mesmo para

Os tipos Qty, Idx, ImpVal e Weight são para quantidades, índices, valores de entrada/saída dos neurônios e os pesos, respectivamente, e todos devem ser acessíveis externamente, por outras unidades. Quantidades e índices devem ser tipos iguais entre si e inteiros, enquanto os valores de entrada/saída e os pesos, da mesma forma devem ser tipos iguais entre si, mas de ponto flutuante. Tipos ponteiro devem ser definidos para os tipos quantidade, valores de entrada/saída e pesos para que a memória para a quantidade de

elementos necessária possa ser dinamicamente alocada como um vetor (pois o tamanho da rede não é pré-definido), o Diagrama 7 ilustra esta funcionalidade de um ponteiro.

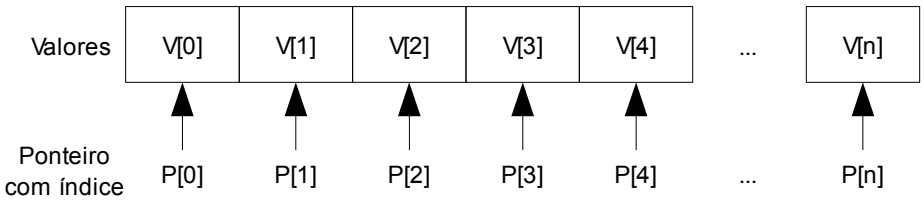


Diagrama 7: Funcionalidade de um ponteiro usado como um vetor de vários valores.

Para fins de comodidade, foi declarado e feita público um procedimento para liberar a memória de um ponteiro qualquer e depois atribuir à variável que armazena o endereço do ponteiro o valor nulo. A finalidade é padronizar a liberação de memória, evitar possíveis erros de programação de não atribuir o valor nulo à variável e permitir alterações futuras para todas as liberações de memória se houver esta necessidade.

4.3.1. Desenvolvimento da classe TNeuron

Cada objeto neuron (neurônio) é uma instância da classe TNeuron. Esta classe permite o processamento das entradas que são fornecidas, gravar o resultado em uma variável que pode ser lida quando necessário e o treinamento, que exige os mesmos valores de entrada que foram fornecidos no último processamento.

Para permitir treinamento em modo batch, o treinamento foi dividido em duas etapas, a primeira é fornecer o erro e calcular a variação de cada peso (esta variação é armazenada e somada às variações anteriores, se existirem) e a segunda é o aplicar as variações aos seus respectivos pesos. A primeira etapa pode ser repetida diversas vezes antes de executar a segunda etapa, deste modo, as variações de pesos para cada neurônio se acumulam para serem aplicadas de uma só vez.

O neurônio deve armazenar o número de entradas, os pesos, o valor de saída, a variação que deve ser aplicada durante o treinamento e, para treinamento em modo batch, a soma das variações de cada peso enquanto não são aplicadas.

Por meio de propriedades é concedido o acesso a dados internos do neurônio bem como a possibilidade de gravação de algumas delas. Os métodos que aceitam índices devem verificar se eles estão dentro da faixa permitida antes antes de realizar qualquer operação.

O construtor (método Create), representado no Diagrama 8, recebe, obrigatoriamente, a quantidade de entradas como parâmetro, e deverá alocar memória para todas as variáveis necessárias, além de inicializar todas as variáveis declaradas. Os valores iniciais dos pesos são aleatórios, e a faixa depende da quantidade de entradas do neurônio, para não haver saturação prematura do neurônio logo na inicialização da rede. Primeiro calcula-se um número aleatório entre -1 e 1 e depois se divide pelo numero de entradas.

Se uma das alocações de memória falhar, cria uma exceção. Se houver exceção seja ela disparada neste método ou qualquer chamada feita por ele, direta ou indiretamente, qualquer memória alocada dentro do método deve ser liberada e a contagem de entradas se mantém em zero, caso contrário a contagem de entradas é definida para quantidade solicitada (recebida como parâmetro).

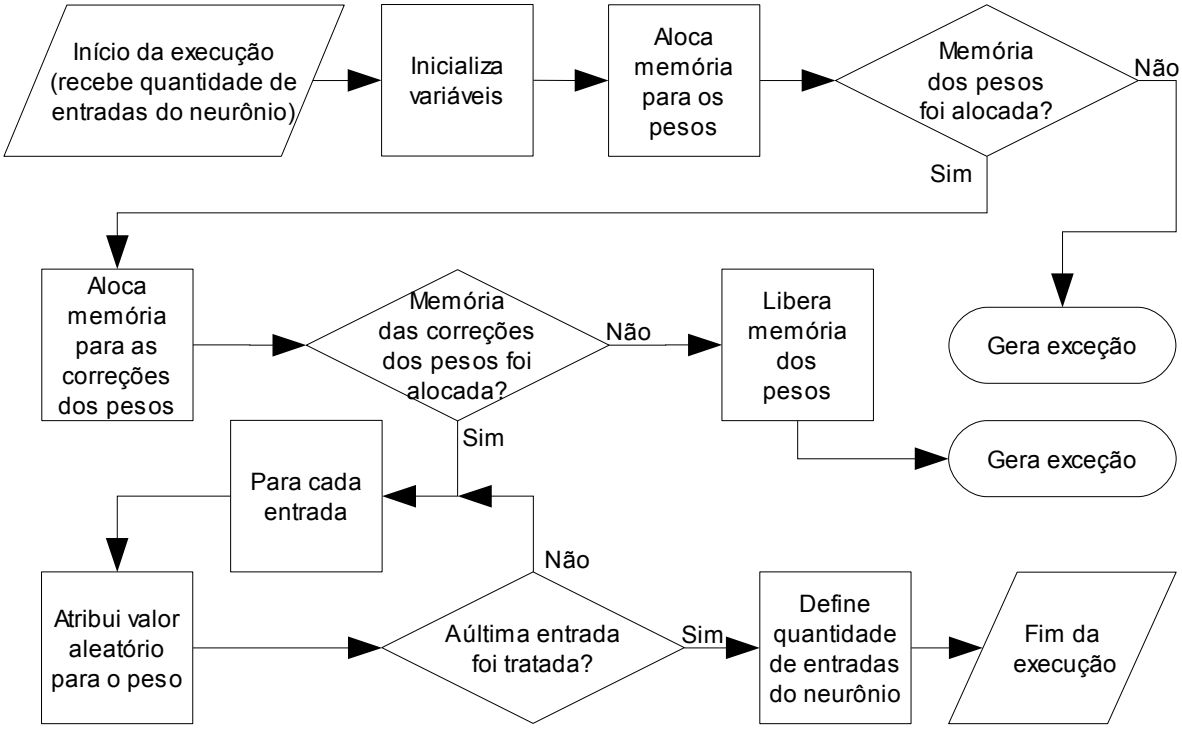


Diagrama 8: Classe TNeuron, método Create.

A taxa de treinamento, fixada em 0,35 foi experimentalmente ajustada para que a rede pudesse convergir para as saídas para as quais as redes criadas foram treinadas, um valor muito alto pode fazer com que a rede fique oscilando e não chegue próximo às respostas desejadas ou mesmo que os neurônios se saturem irreversivelmente e valores muito baixos podem fazer com que a rede demore muito para convergir. Futuramente este parâmetro pode ser definido na criação da RNA ou mesmo durante a utilização.

O destrutor (método Destroy) deve liberar a memória, mas apenas se ela está alocada, este controle é feito pela variável da quantidade de entradas, se for zero, não há

memória alocada, se for diferente de zero (obviamente maior), há memória alocada e a quantidade de elementos de cada ponteiro em questão é igual à quantidade de entradas, o Diagrama 9 ilustra o processo.

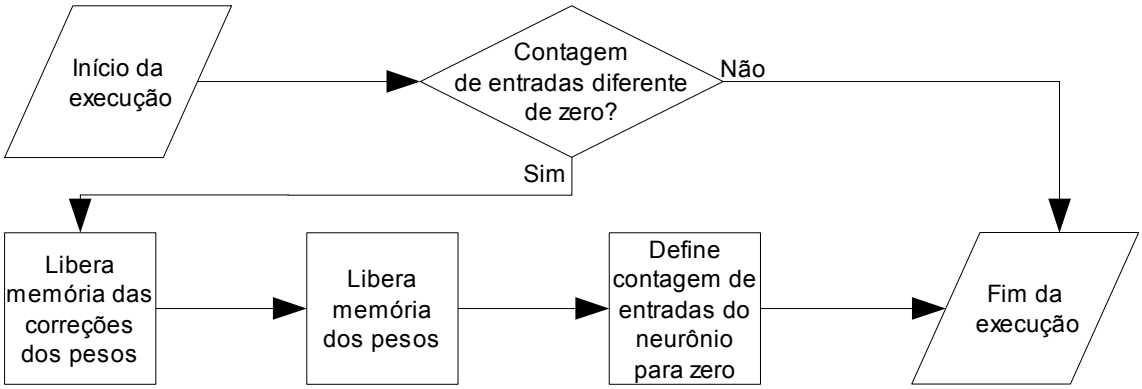


Diagrama 9: Classe TNeuron, método Destroy.

Todos os métodos que recebem o índice da entrada como parâmetro devem fazer a verificação para certificar que o índice está na faixa permitida, para isto, utilizam a função checkRangeWith, que retorna um valor booleano verdadeiro se o índice estiver na faixa permitida e gera uma exceção se não estiver. Adicionalmente esta função aceita um parâmetro não obrigatório (noException) que se não passado recebe o valor falso, se receber o valor verdadeiro, quando um índice fora da faixa é passado, a função retorna o valor falso ao invés de gerar exceção. O Diagrama 10 mostra este método.

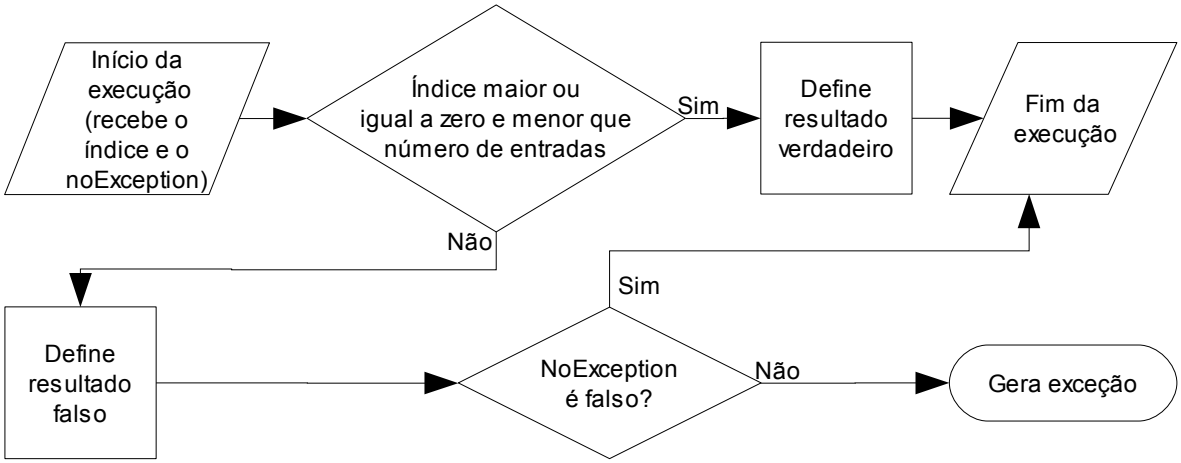


Diagrama 10: Classe TNeuron, método checkRangeWith.

Os pesos são acessados por meio da propriedade Weight, que no código chama a função getWeight para ler o valor de um peso e o procedimento setWeight para gravar. A função getWeight recebe como parâmetro o índice da entrada e retorna o peso referente. O

procedimento setWeight recebe como parâmetros o índice da entrada e o novo valor de peso. O Diagrama 11 e o Diagrama 12 ilustram o funcionamento destes métodos.

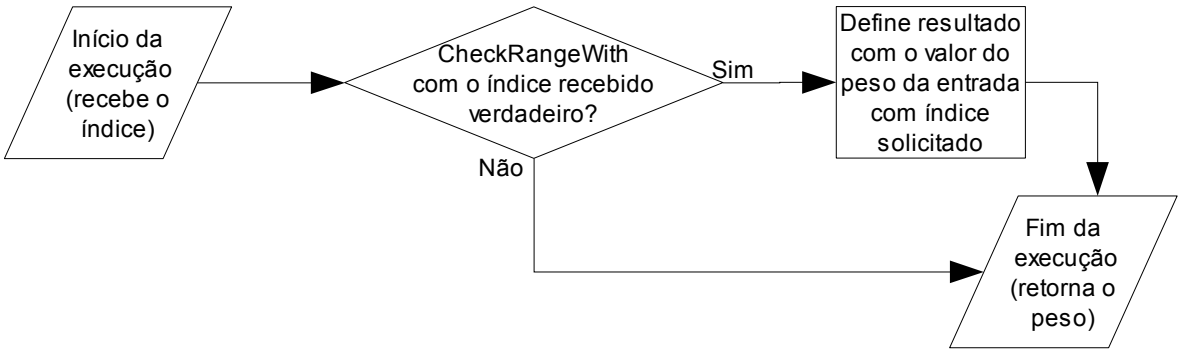


Diagrama 11: Classe TNeuron, método getWeight.

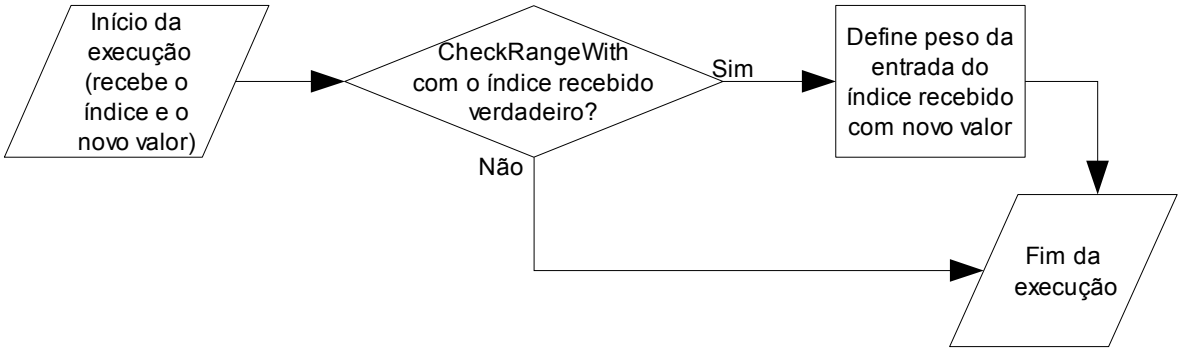


Diagrama 12: Classe TNeuron, método setWeight.

O método process faz o processamento de entradas e armazena o valor de saída do neurônio. Se não estiver havendo treinamento, é o principal código que o neurônio executa, nele está presente a função de transferência (sigmóide neste caso). Todos os valores de entrada devem ser passados para a função por meio de um ponteiro. As etapas são mostradas no Diagrama 13.

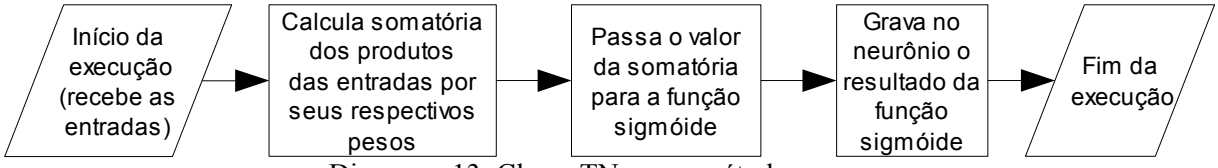


Diagrama 13: Classe TNeuron, método process.

O primeiro passo do treinamento para o neurônio é calcular uma variação desejada para a saída, que é o erro absoluto da saída do neurônio multiplicado pela derivada da saída para o mesmo valor. O método setAbsoluteError calcula a variação que a saída do neurônio deve sofrer. A Diagrama 14 mostra o procedimento.

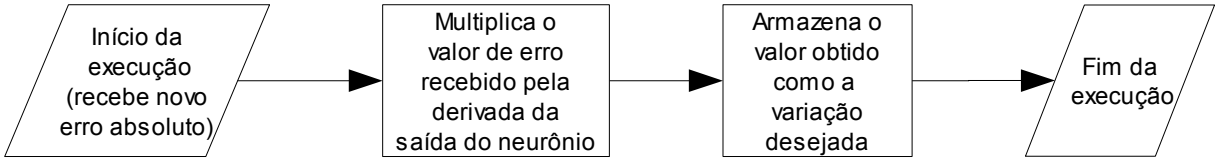


Diagrama 14: Classe TNeuron, método setAbsoluteError.

O segundo passo do treinamento é calcular as variações de cada peso. Se o valor da entrada for zero, a variação de seu peso será zero, pois não é possível determinar a variação adequada. As variações dos pesos são armazenadas e vão se acumulando até que sejam aplicadas. O método addDeltas mostrado no Diagrama 15 implementa este processo.

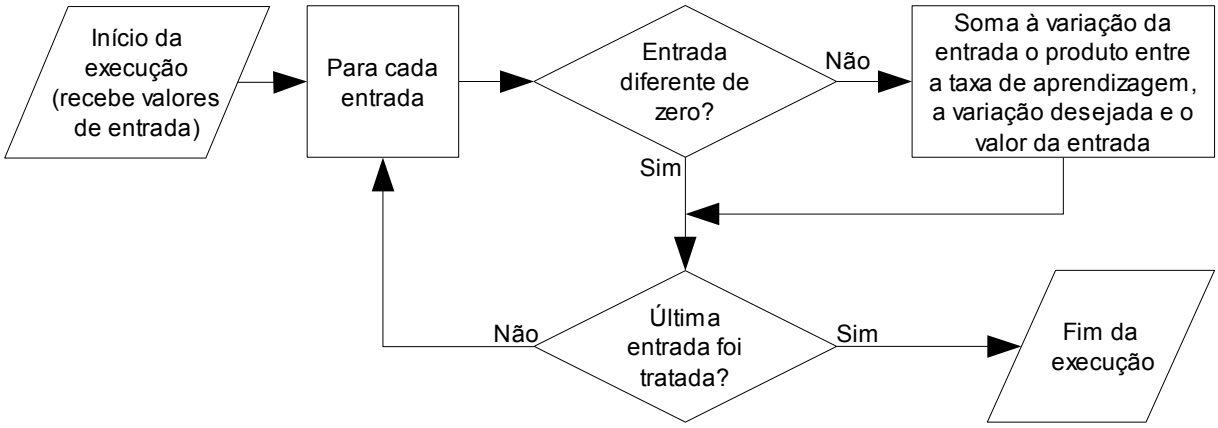


Diagrama 15: Classe TNeuron, métod addDeltas.

O terceiro e último passo para o treinamento é o ajuste dos pesos com as variações anteriormente calculadas pelos dois métodos acima. O método adjust, mostrado no Diagrama 16, aplica as variações dos pesos aos respectivos e posteriormente zera as variações para que novos treinamentos possam ser iniciados.

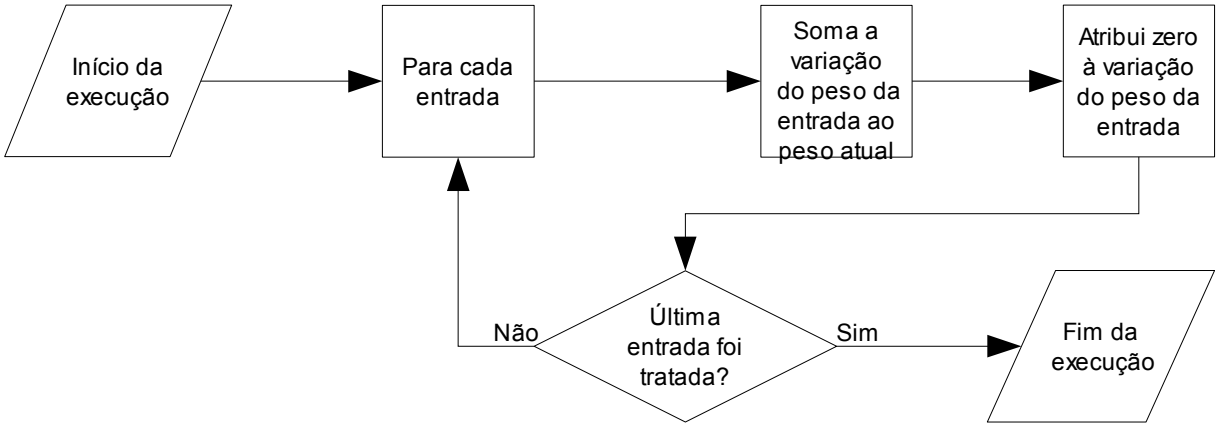


Diagrama 16: Classe TNeuron, método adjust.

Com o código criado com as definições acima, é possível criar, a partir da classe TNeuron, um objeto neurônio com todas as funcionalidades necessárias.

4.3.2. Implementação da classe TLayer

Cada objeto layer (camada) contém um conjunto de neurônios e os controla, enviando comandos de acordo com os comandos que recebe, funcionando como uma interface entre o elemento de nível superior (RNA) e os de nível inferior (neurônios).

Para criar o conjunto de neurônios, foi utilizado um tipo auxiliar de ponteiro, como os que foram utilizados para criar os conjuntos de pesos e valores. A camada armazena, ainda, a quantidade de entradas, quantidade de neurônios solicitada e a quantidade de neurônios. Algumas propriedades redirecionam diretamente para propriedades dos neurônios.

O construtor recebe a quantidade de entradas e o número de neurônios, se encarregará de alocar a memória necessária para o conjunto de neurônios e criar cada um deles.

Para garantir que a memória alocada é liberada, algumas instruções são protegidas. Se ocorrer uma exceção durante a criação dos neurônios (trecho protegido), todos os que já foram criados são destruídos e a memória é liberada, após isto a exceção é propagada para o próximo tratador de exceção, ou seja, se não houver tratamento posterior que considere a exceção inofensiva, o programa termina. Se a exceção for de falta de memória, adiciona na mensagem da exceção o número do neurônio que seria (ou foi) criado quando a exceção ocorreu.

O Diagrama 17 mostra a implementação do construtor, o quadro tracejado da figura representa as instruções protegidas pelo tratador de exceção. O tratador de exceção é mostrado com um fluxo separado, pois não é parte do fluxo normal do programa.

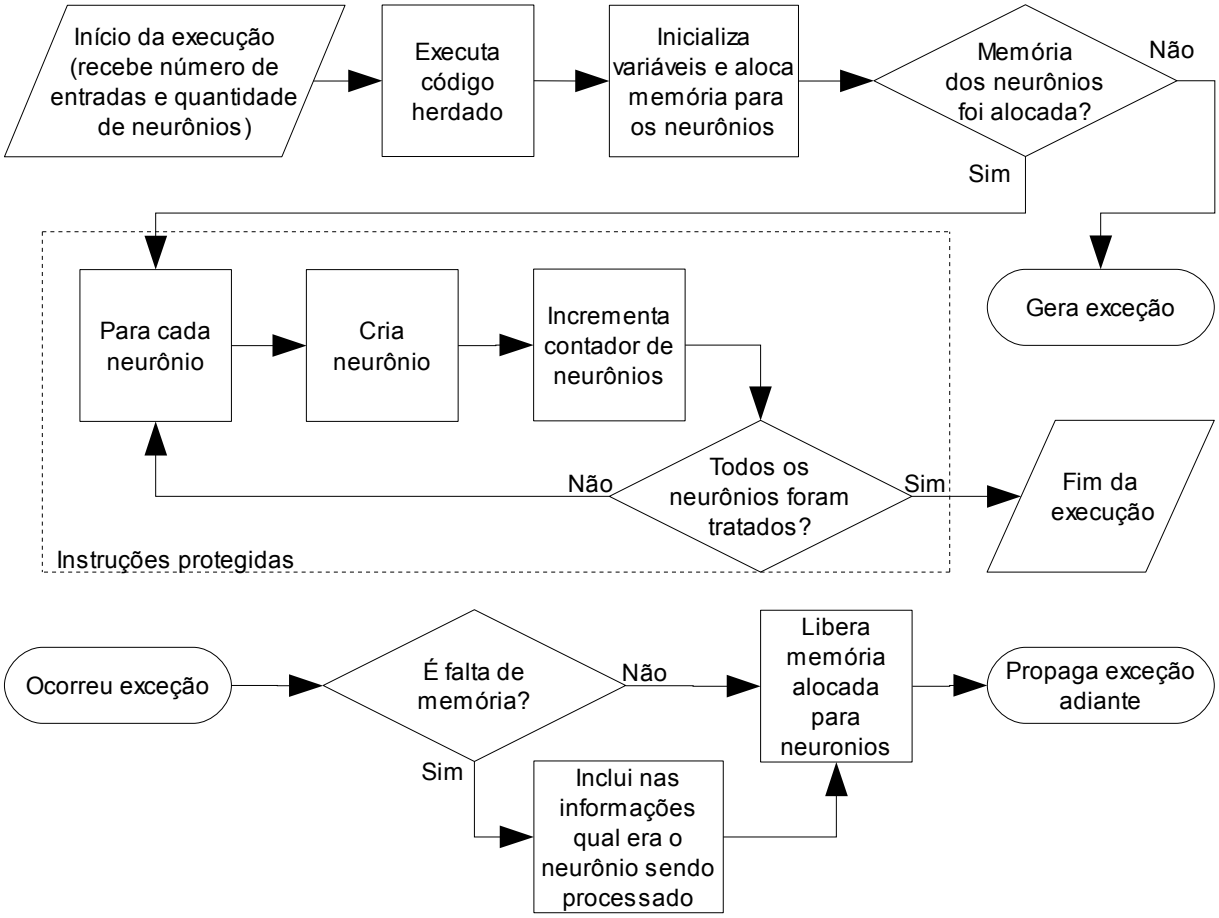


Diagrama 17: Classe TLayer, método Create.

O destrutor, mostrado no Diagrama 18, deve destruir todos os neurônios criados e depois liberar a memória alocada para o conjunto de neurônios. Para que o mesmo código pudesse ser utilizado no construtor (acima), as chamadas para destruir os neurônios e a liberação de memória são feitos por meio do método freeNeurons.

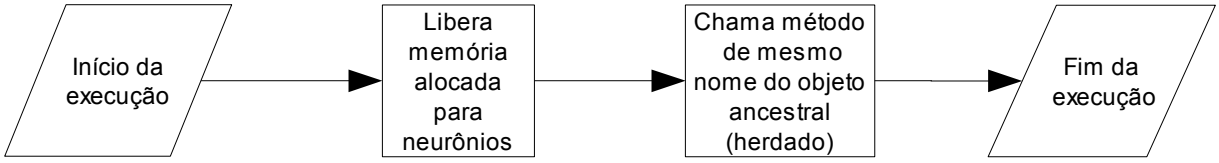


Diagrama 18: Classe TLayer, método Destroy.

O método freeNeurons verifica quais neurônios foram criados por meio do contador e os destrói, e em seguida libera a memória alocada pelo ponteiro do conjunto de neurônios, ele está representado no Diagrama 19.

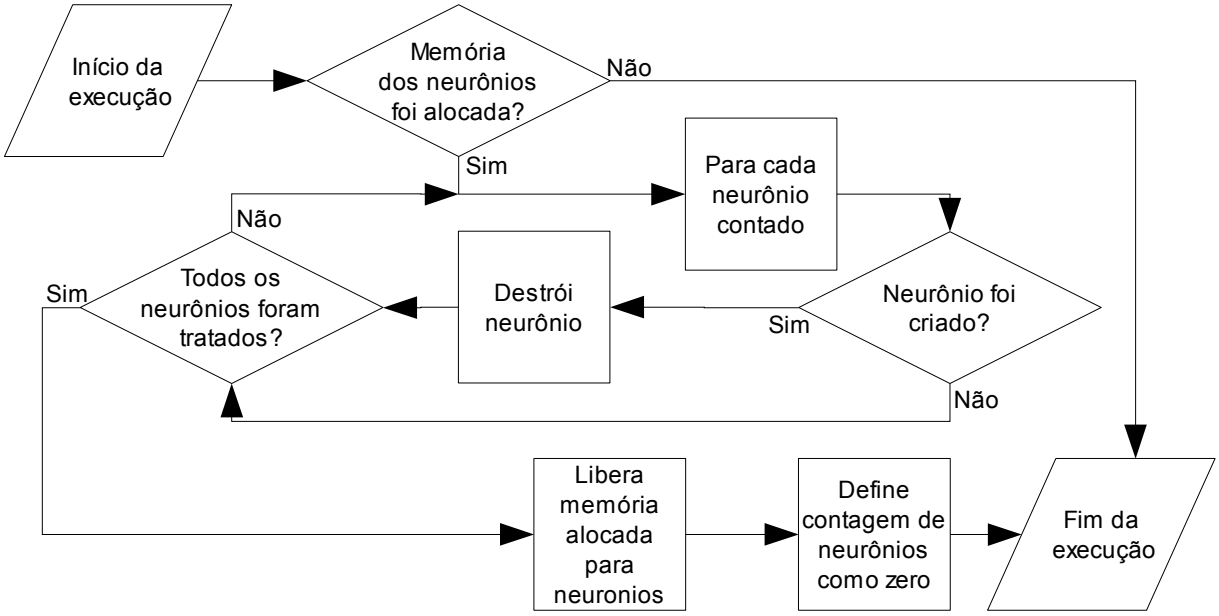


Diagrama 19: Classe TLayer, método freeNeurons.

O método checkRangeWith verifica se o índice do neurônio recebido está na faixa aceita. Ele deve ser chamado em todos os procedimentos que precisam do índice do neurônio, mostrado no Diagrama 20.

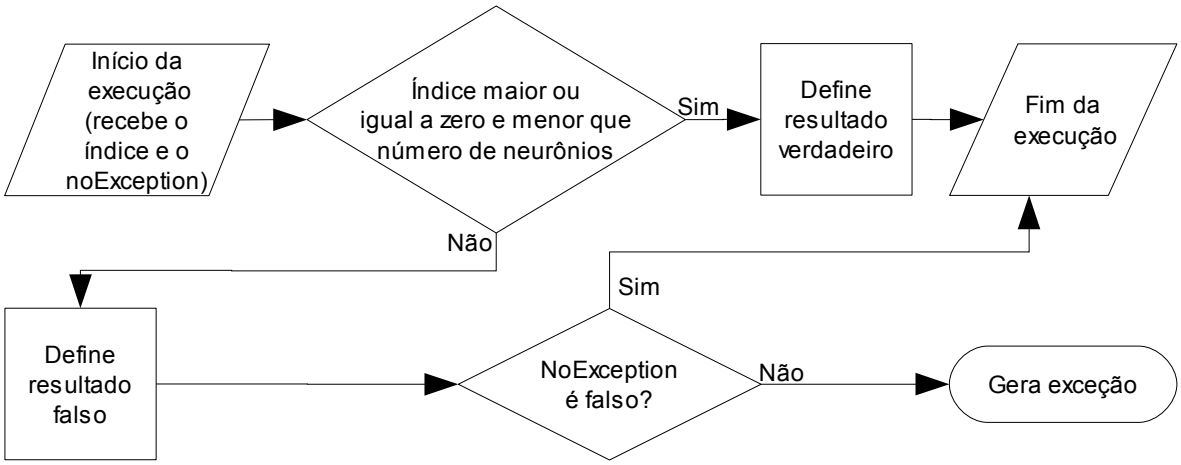


Diagrama 20: Classe TLayer, método checkRangeWith.

O método process, no Diagrama 21, recebe os valores de entrada e é responsável por chamar o processamento de cada neurônio.

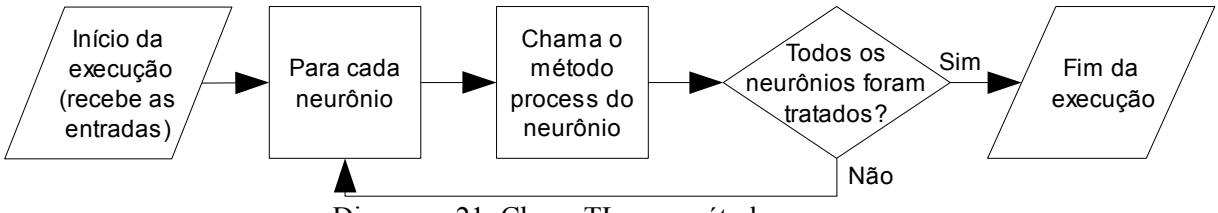


Diagrama 21: Classe TLayer, método proccess.

O método readOutputs, no Diagrama 22, copia as saídas de cada neurônio da camada na posição de memória indicada pelo ponteiro passado e nas seguintes, até que todos os valores tenham sido lidos e retorna o número de valores copiados, o que pode simplificar o tratamento de várias camadas consecutivas.

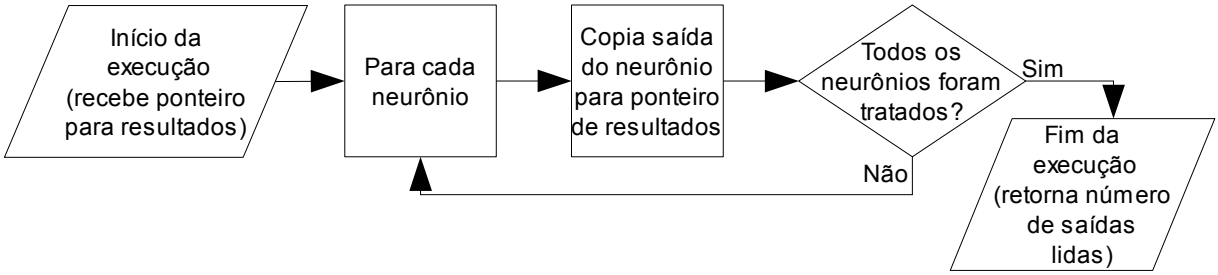


Diagrama 22: Classe TLayer, método readOutputs.

O método setErrors, no Diagrama 23, recebe os erros calculados por readPrevErrors do layer seguinte e os aplica a seus respectivos neurônios. Os índices devem coincidir.

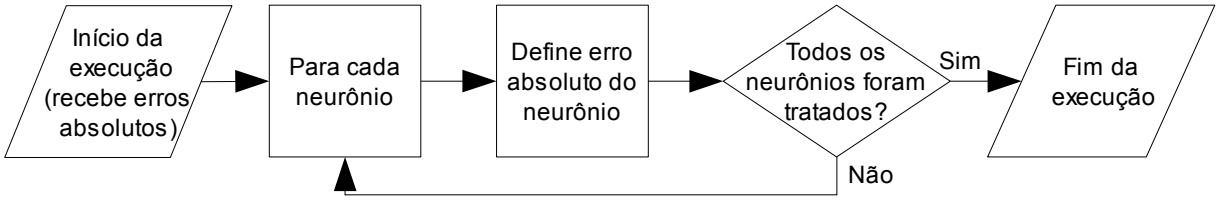


Diagrama 23: Classe TLayer, método setErros.

O método readPrevErrors, no Diagrama 24, calcula os erros que devem ser passados para o layer anterior para o cálculo dos deltas dos neurônios (do layer anterior). Os erros são copiados para a posição de memória apontada pelo ponteiro recebido como parâmetro. No caso da camada de saída (que não tem layer posterior), os erros são calculados diretamente com os valores de saída da rede e os valores que seriam corretos.

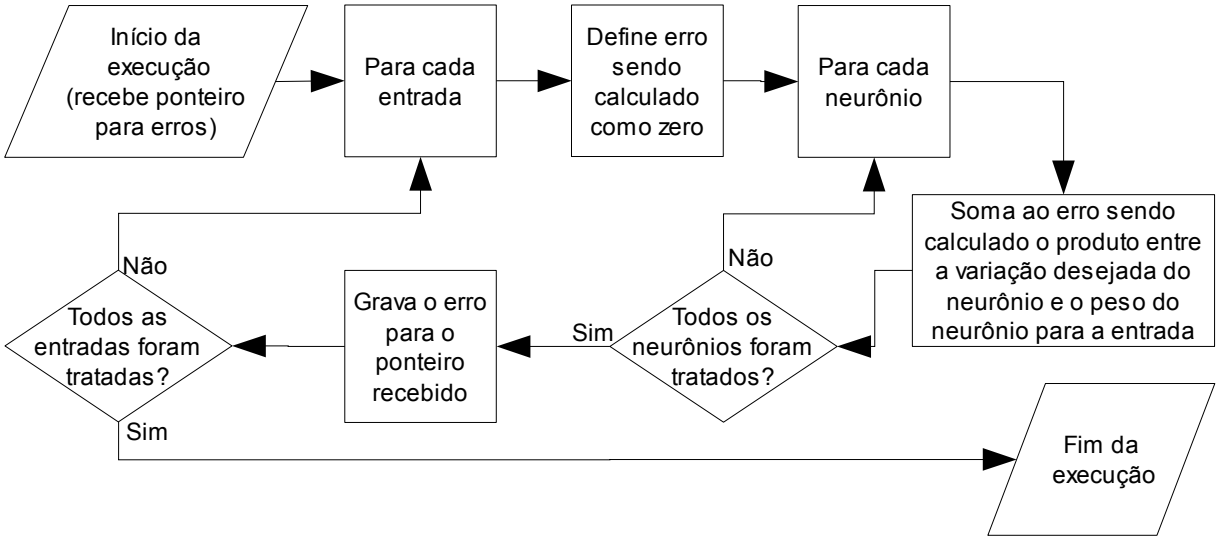


Diagrama 24: Classe TLayer, método readPrevErrors.

O método addDeltas, no Diagrama 25, recebe as entradas da rede e chama o método addDeltas de cada neurônio pertencente à camada com o mesmo parâmetro passado. Antes da chamada deste método, o processamento da rede já deve ter sido feito e os erros de cada neurônio já devem estar definidos.

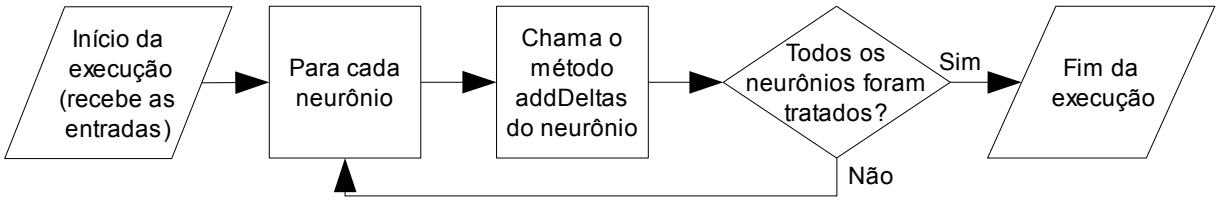


Diagrama 25: Classe TLayer, método addDeltas.

O método adjust, no Diagrama 26, simplesmente chama os métodos adjust de cada neurônio, os fatores de ajuste já devem ter sido calculados por meio do método addDeltas, eles são adicionados aos pesos.

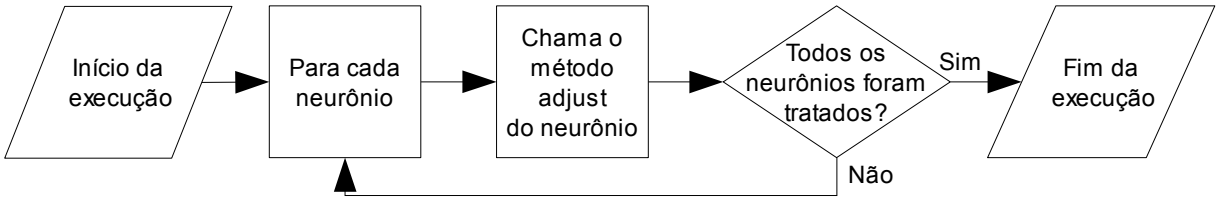


Diagrama 26: Classe TLayer, método adjust.

Os métodos getNeuronDelta (Diagrama 27), getNeuronOutput (Diagrama 28) e getNeuronWeight (Diagrama 29) são interfaces para o acesso direto às propriedades correspondentes dos neurônios. O índice do neurônio é recebido como parâmetro, o método getNeuronWeight recebe um parâmetro adicional, o índice da entrada.

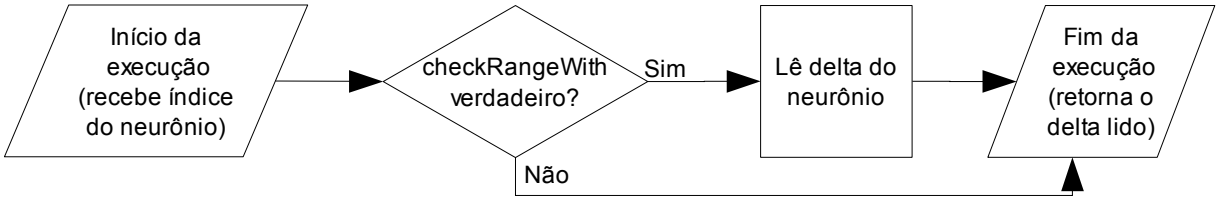


Diagrama 27: Classe TLayer, método getNeuronDelta.

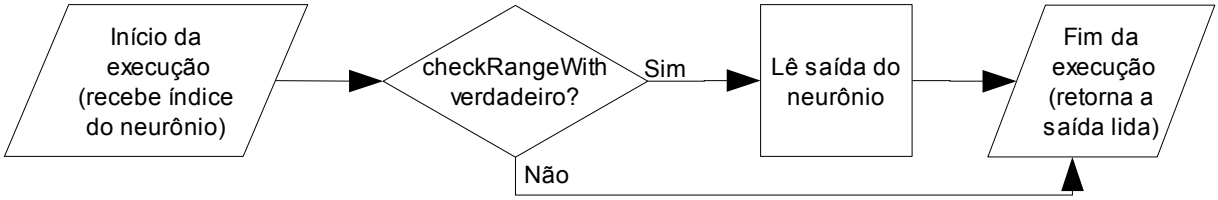


Diagrama 28: Classe TLayer, método getNeuronOutput.

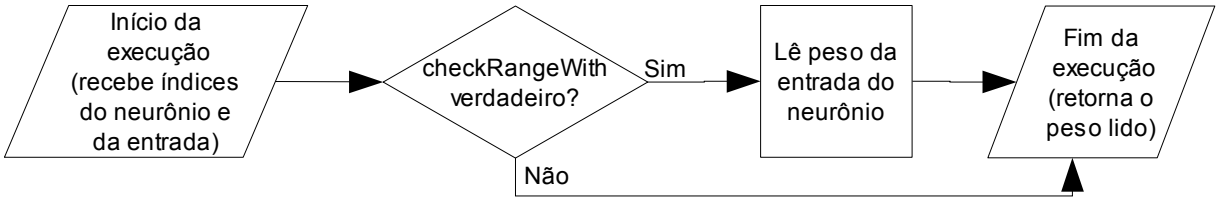


Diagrama 29: Classe TLayer, método getNeuronWeight.

O método setNeuronWeight, no Diagrama 30, é uma interface para gravação da propriedade weight de um neurônio da camada. O índice do neurônio, o índice da entrada e o novo valor são recebidos como parâmetros.

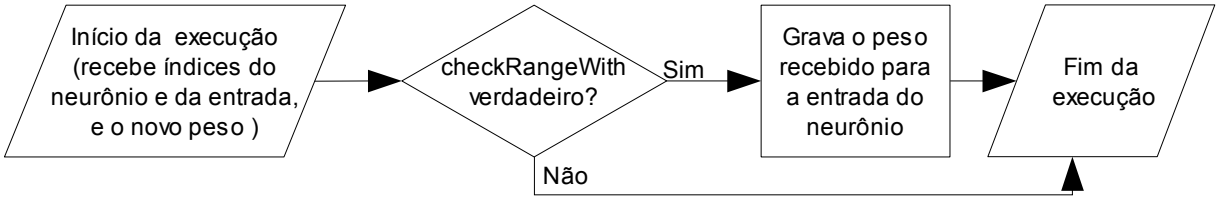


Diagrama 30: Classe TLayer, método getNeuronWeight.

4.3.3. Implementação da classe TNN

O objeto nn (RNA), instância da classe TNN, deve possuir variáveis para armazenar a quantidade de entradas da rede, quantidade de camadas, quantidade solicitada de camadas, a maior quantidade de saídas entre todas as camadas, a quantidade de saídas da rede, o conjunto de objetos TLayer, e o conjunto de saídas da rede.

Antes da declaração da classe TNN, o tipo auxiliar ponteiro para camada é definido para representar conjunto de camadas, como feito anteriormente para os neurônios.

Vários dos métodos desta classe aceitam o parâmetro workArr (matriz de trabalho) de tipo PimpValArr e valor padrão nil, que é um ponteiro para um conjunto de valores para realizar armazenamento de valores durante o trabalho. O valor padrão implica em eliminar a obrigação de fornecer este parâmetro. Se não for recebido, a memória será alocada para uso e após, destruída pelo método. Se recebido, deve haver memória alocada para, no mínimo uma quantidade de valores igual ao máximo número de saídas entre todas as camadas da rede. Os valores armazenados nos elementos apontados por workArr não devem ser considerados previsíveis e de nenhuma forma devem ser associados aos valores anteriores à chamada do método.

A presença deste parâmetro adicional permite que o programa execute com menos alocações e liberações de memória, pois, no caso da biblioteca sendo implementada, são muito numerosas e podem ter um impacto significativo no custo de processamento durante a execução.

O construtor, método Create mostrado no Diagrama 31, deve receber como parâmetro a quantidade de entradas, a quantidade de camadas e o conjunto com a quantidade de neurônios para cada camada, alocar memória e criar cada camada, passando o respectivo número de neurônios. Se uma exceção ocorrer, todos os elementos criados devem ser destruídos e a memória deve ser liberada.

A camada de entrada da rede, por não possuir neurônios, não é considerada no construtor, assim, o parâmetro layerCount deve incluir apenas as camadas ocultas mais a camada de saída.

O número de saídas de uma camada é igual à quantidade de neurônios, e o número de entradas de uma camada é igual ao número de saídas da camada anterior.

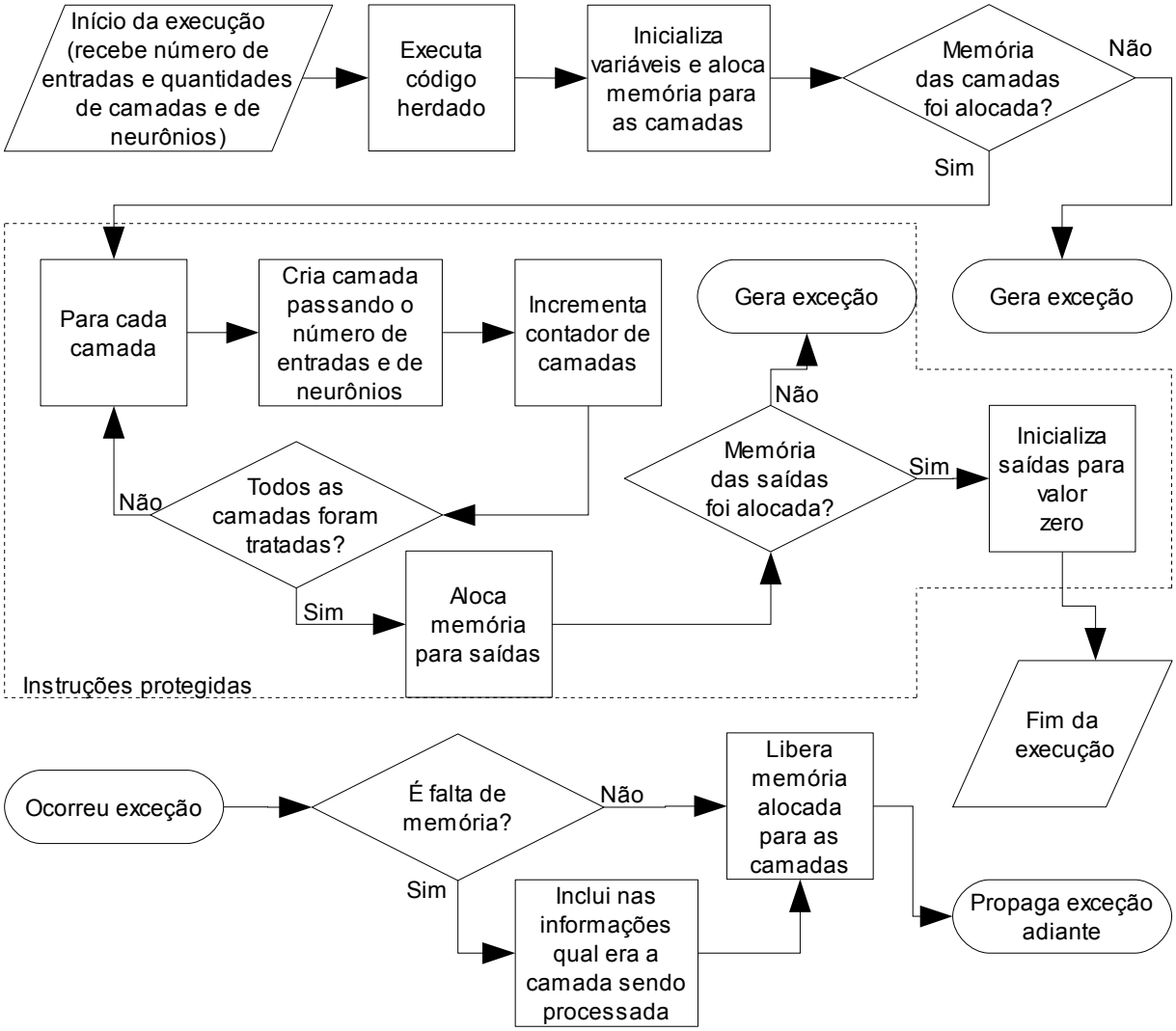


Diagrama 31: Classe TNN, método Create.

O destrutor, método Destroy no Diagrama 32, deve verificar se a memória foi alocada e liberá-la conforme necessário.

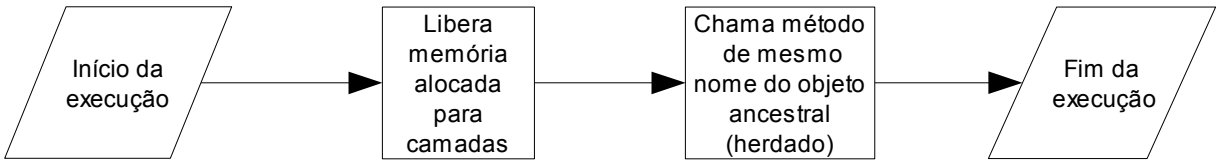


Diagrama 32: Classe TNN, método Destroy.

O método freeLayers, no Diagrama 33, verifica se há memória alocada e, se houver, destrói os objetos camada e libera a memória alocada para o conjunto delas.

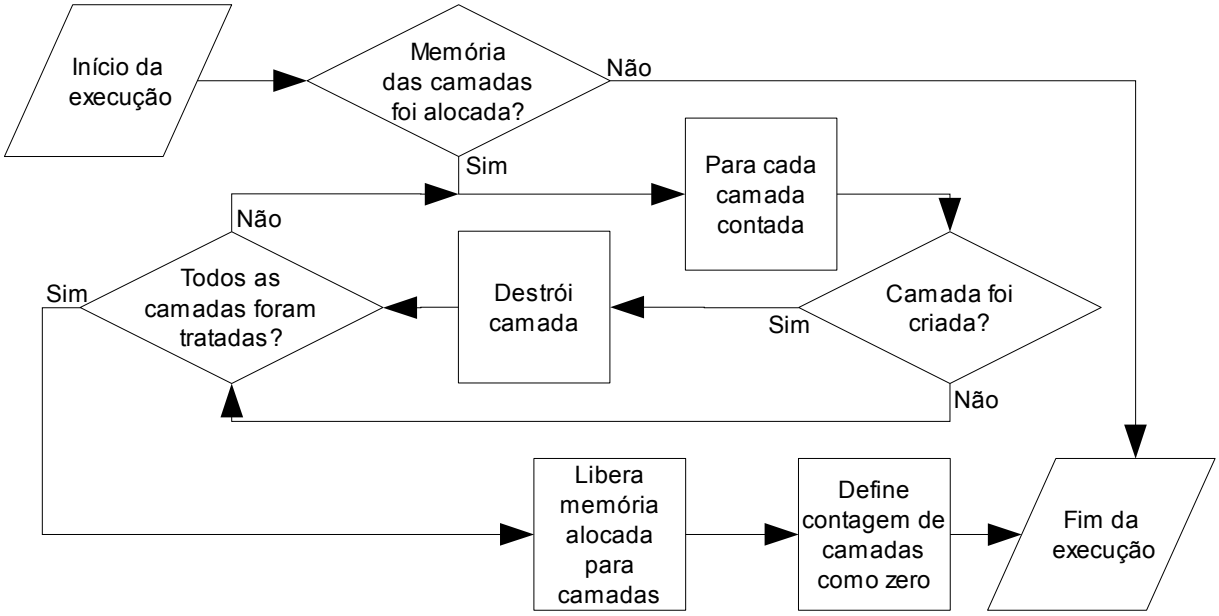


Diagrama 33: Classe TNN, método freeLayers.

O método checkRangeWith, no Diagrama 34, verifica se um índice de layer recebido está ou não na faixa permitida.

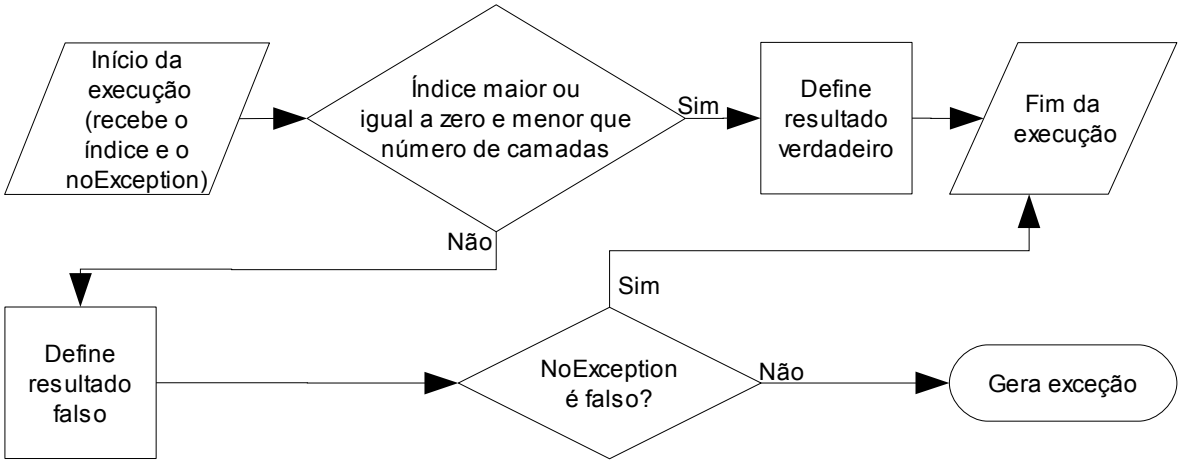


Diagrama 34: Classe TNN, método checkRangeWith.

O método process, no Diagrama 35, recebe como parâmetro as entradas e executa o processamento da RNA chamando o método process de cada camada. Os valores de entrada para cada camada são as saídas da camada anterior.

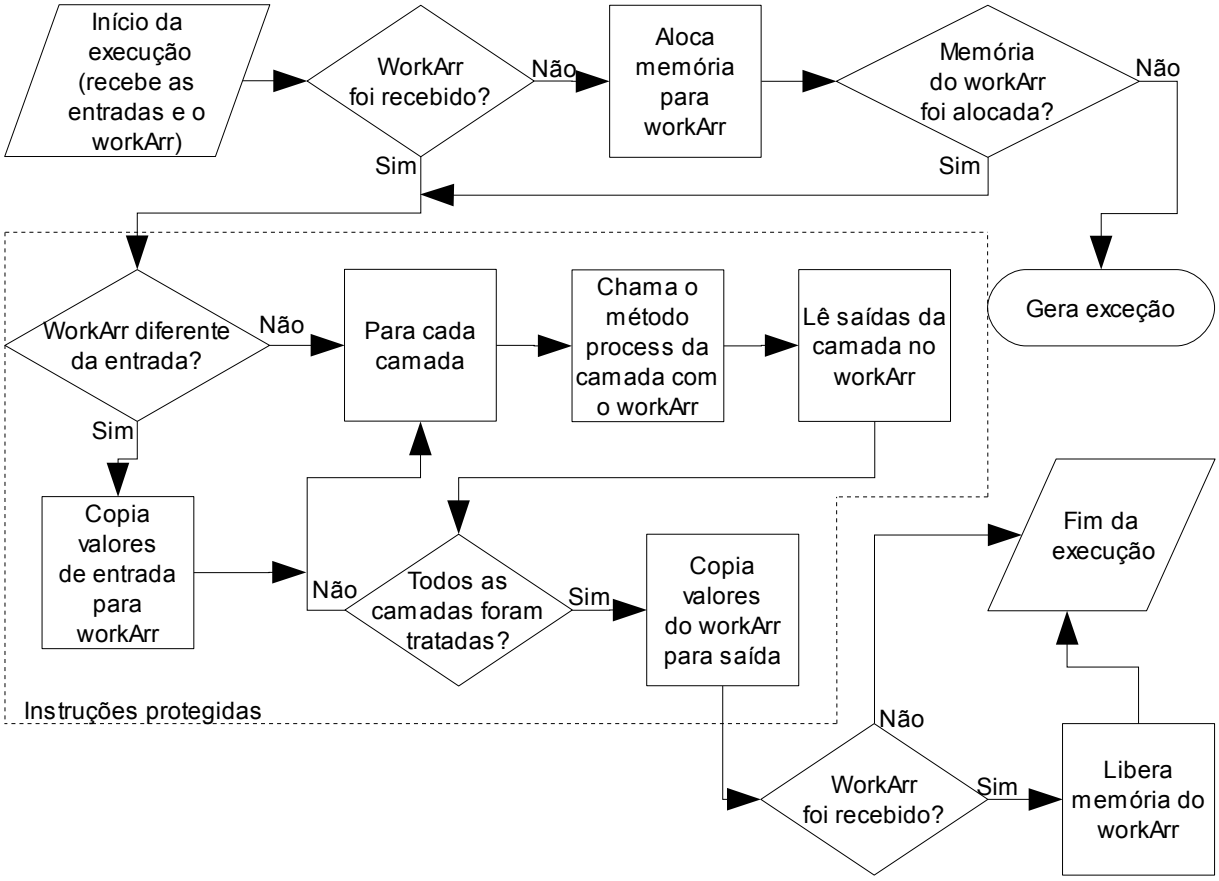


Diagrama 35: Classe TNN, método process.

O método readOutputs, no Diagrama 36, copia todas as saídas para a posição de memória apontada pelo ponteiro passado como o parâmetro target (alvo). A memória alocada para o parâmetro deve ser, no mínimo, suficiente para gravar o número de saídas da rede.

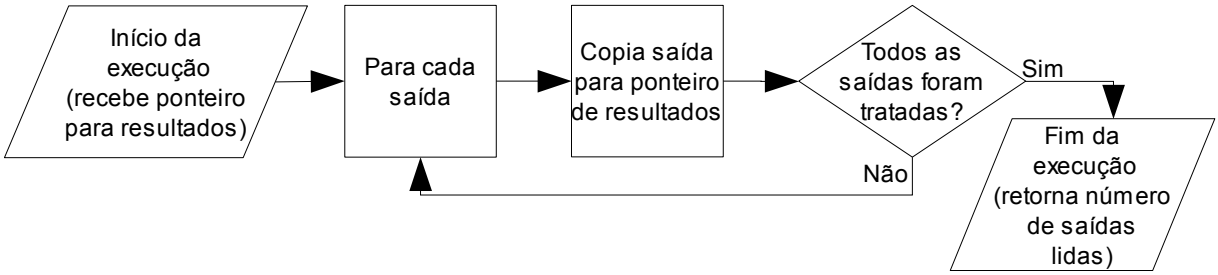


Diagrama 36: Classe TNN, método readOutputs.

O método getOutput, no Diagrama 37, retorna uma das saídas da RNA especificada pelo índice passado como parâmetro.

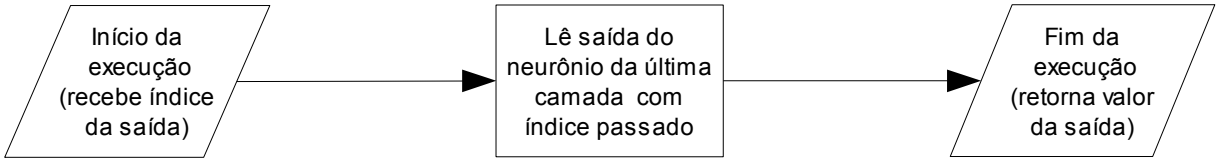


Diagrama 37: Classe TNN, método getOutput.

O método computeErrorsTo, no Diagrama 38, calcula os erros dos resultados do processamento de um conjunto de entradas para as saídas desejadas e envia para cada camada, com propagação reversa do erro, sendo que cada camada que recebe os erros calcula e retorna os erros da camada anterior.

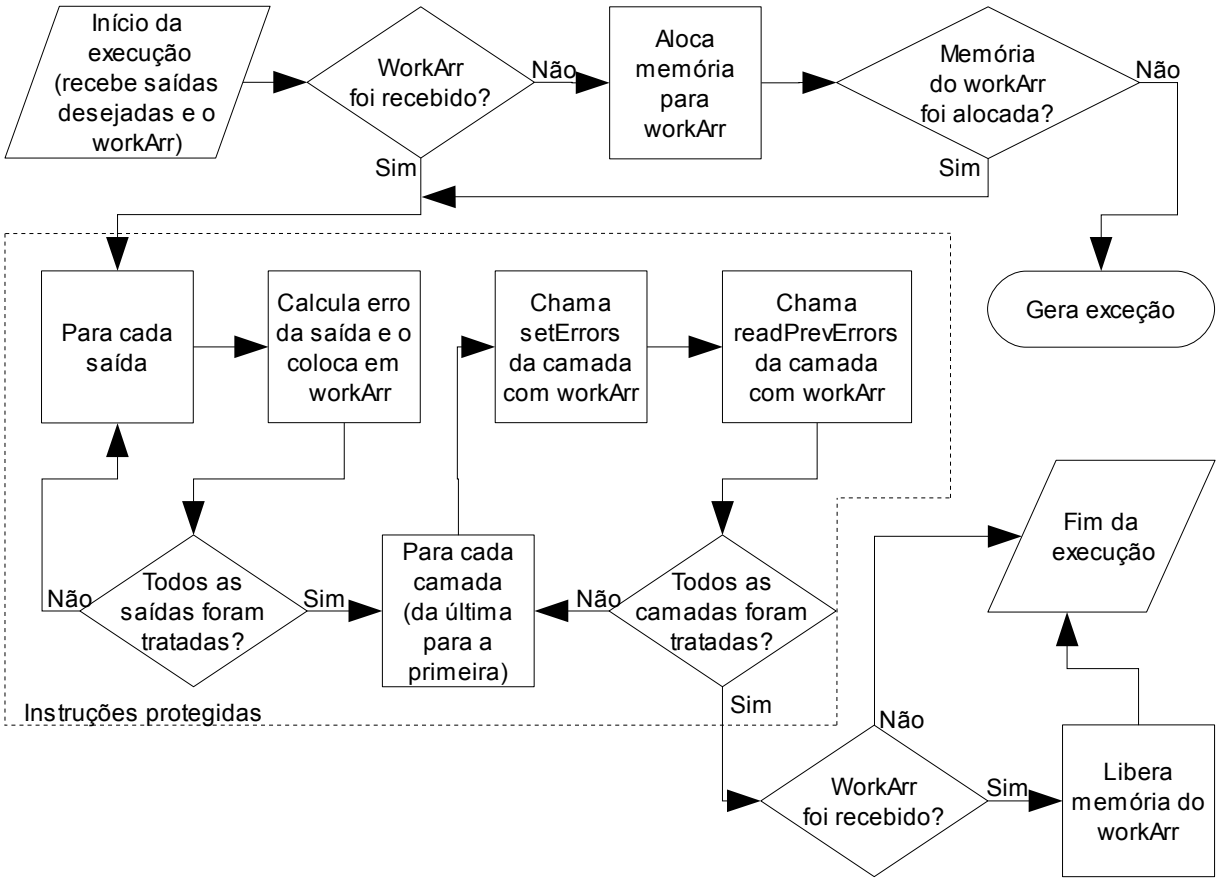


Diagrama 38: Classe TNN, método computeErrosTo.

O método addDeltas, no Diagrama 39, deve receber as entradas da rede que já foram processadas e aplicar a cada camada, sendo as entradas de cada camada a saída da camada anterior. Desta forma, acumulando as variações de peso dos neurônios para todas as camadas. O cálculo dos deltas já deve ter sido feito antes da chamada deste método.

O método adjust, no Diagrama 40, aplica os valores de variação de peso que já foram calculados para todas as camadas.

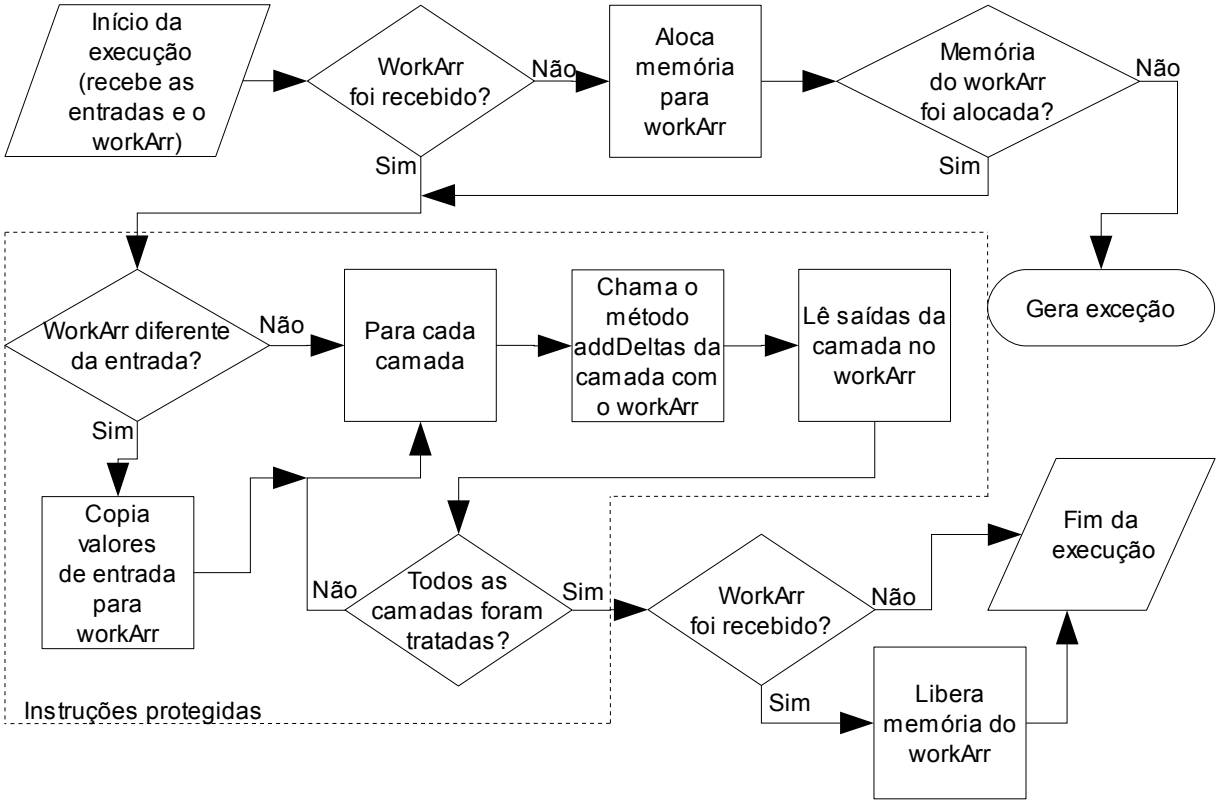


Diagrama 39: Classe TNN, método addDeltas.

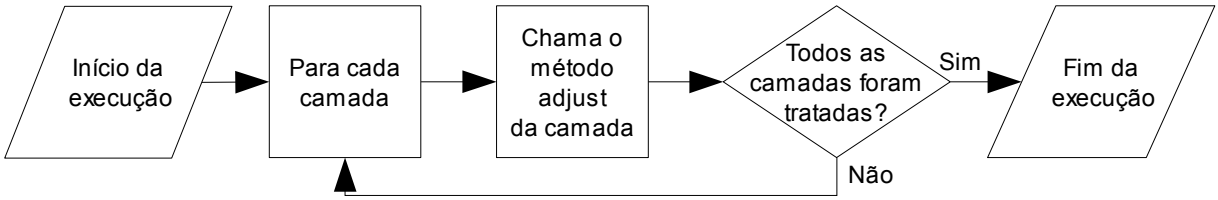


Diagrama 40: Classe TNN, método adjust.

O método train, no Diagrama 41, faz um treinamento completo para um único conjunto de entradas e saídas. Pode ser usado como modelo para uma implementação externa que pode ser desejável para melhorar o desempenho do treinamento.

O método getLayer, no Diagrama 42, retorna uma referência para uma camada qualquer, que pode ser utilizada para acesso direto a qualquer camada da rede especificada pelo índice passado como parâmetro. Ele pode ser útil para gravar ou recuperar qualquer estado da RNA.

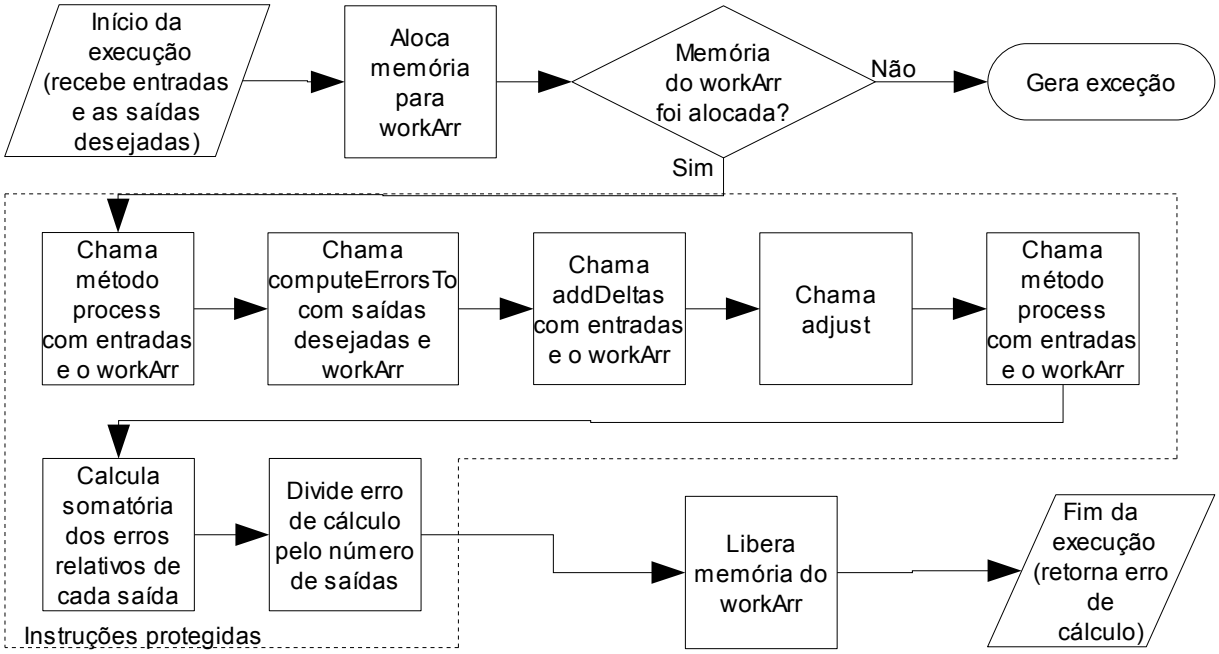


Diagrama 41: Classe TNN, método train.

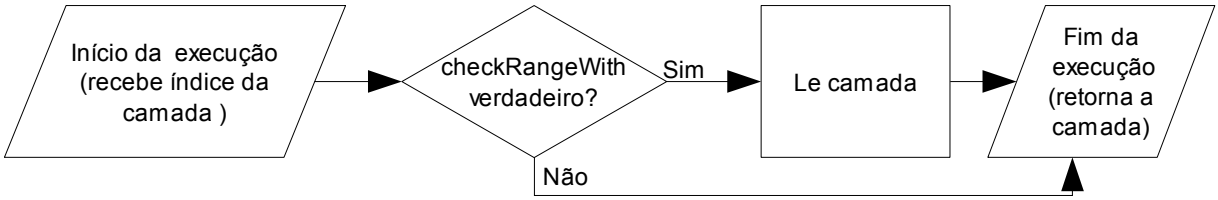


Diagrama 42: Classe TNN, método getLayer.

Os métodos getLayerInputCount (Diagrama 43) e getNeuronCount (Diagrama 44) são interfaces para acesso direto a estas propriedades das camadas. Ambas devem receber o índice da camada.

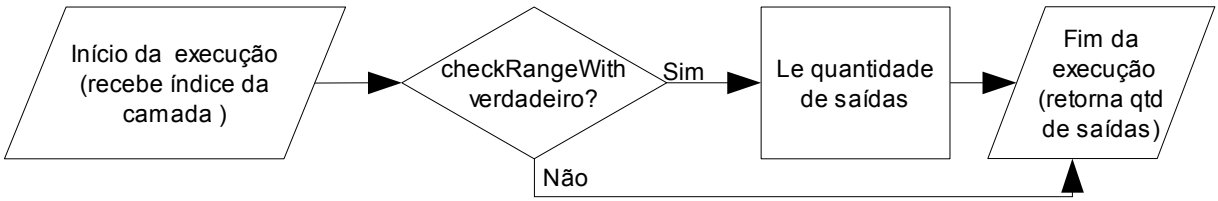


Diagrama 43: Classe TNN, método getLayerInputCount.

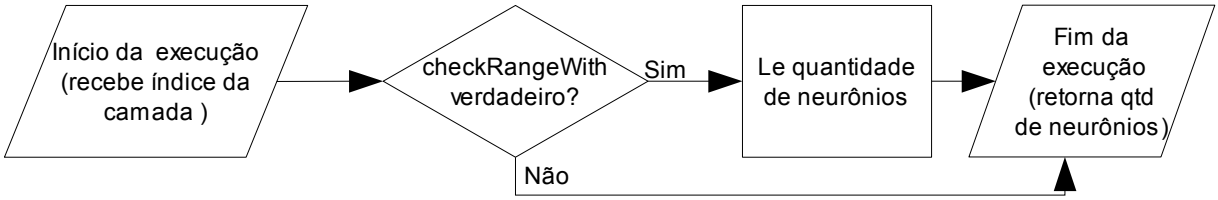


Diagrama 44: Classe TNN, método getNeuronCount.

Os métodos getNeuronOutput (Diagrama 45), getNeuronWeight (Diagrama 46) e setNeuronWeight (Diagrama 47) são interfaces para acesso direto a estas propriedades dos neurônios. Todas devem receber o índice da camada e o índice do neurônio na camada, os dois últimos, também o índice da entrada do neurônio e o setNeuronWeight deve, além do que já foi citado, deve receber o novo valor do peso a ser atribuído.

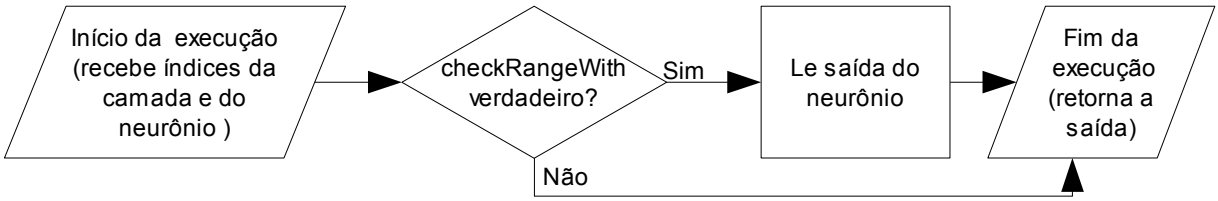


Diagrama 45: Classe TNN, método getNeuronOutput.

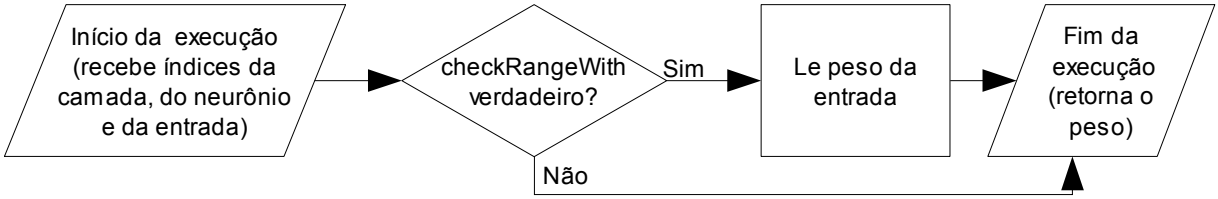


Diagrama 46: Classe TNN, método getNeuronWeight.

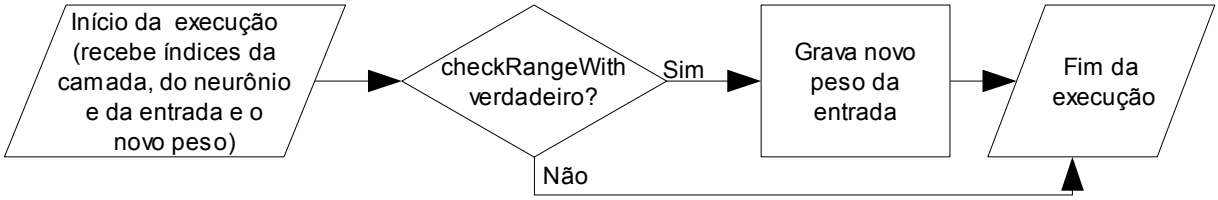


Diagrama 47: Classe TNN, método setNeuronWeight.

A classe que acaba de ser definida pode ser usada para criar instâncias de uma MLP e é suficientemente flexível para permitir outras implementações com reaproveitamento de boa parte do código.

O código completo do arquivo que implementa a unidade encontra-se no APÊNDICE B e no endereço de internet <http://www.biazi.eng.br/mack/tgi/>, neste endereço também é possível encontrar o programa de simulação apresentado a seguir neste trabalho. Para utilização de qualquer um deles, a respectiva licença deve ser examinada, completamente entendida e aceita.

5. PROGRAMA DE SIMULAÇÃO QUE UTILIZA A RNA DESENVOLVIDA

O programa foi escrito em Object Pascal na IDE Lazarus, que permite a compilação em vários sistemas operacionais sem nenhuma alteração de código. Em qualquer sistema operacional que se utilize, apesar de aparências diferentes dos controles da tela, a interface do programa é sempre similar, alterando-se muito pouco, deste modo, não há diferença de percepção das funções do programa quando usado em um ou outro sistema operacional ou interface gráfica.

O programa pode ser obtido gratuitamente em versões para Windows e Linux por meio do endereço de internet <http://www.biazi.eng.br/mack/tgi/>. Para utilização, a licença deve ser examinada, completamente entendida e aceita.

5.1. CARATERÍSTICAS E FINALIDADE DO PROGRAMA

O programa foi desenvolvido para utilizar RNA's para controlar um braço mecânico virtual, permite vários ajustes para alterar as características do sistema e visualizações e a interferência direta do usuário em vários parâmetros, mesmo que a simulação esteja em andamento.

Dois métodos foram desenvolvidos para o controle do braço mecânico, controlador PID e RNA. O controlador PID foi desenvolvido tanto para criar dados a serem utilizados no treinamento da RNA quanto para criar resultados para comparar aos da RNA.

O controlador PID é conhecido por sua grande aplicabilidade nos mais variados tipos de sistemas, mostrando-se adequado e eficiente na solução da maioria dos problemas. Seu uso é destinado a melhorar as ações de sistemas quanto aos tempos de resposta e oscilações.

É basicamente uma função de transferência que recebe um erro e retorna o valor de controle que será utilizado para corrigi-lo e consiste na soma de três termos, um proporcional, um diferencial e um integral. Cada um dos termos possui uma constante que o multiplica, enfatizando ou atenuando sua característica no valor resultante do controle.

O termo proporcional, como o nome diz, retorna um valor diretamente proporcional ao valor do erro, simplesmente multiplicado-o por uma constante normalmente chamada K_p. Este termo, por si só, não cria oscilações no sistema mas se o sistema já possuir termos proporcionais ou integrais, como inércia por exemplo, o aumento do valor de K_p pode

tornar o sistema oscilatório, se ele ainda não for, aumentar a amplitude de oscilação já existente ou simplesmente torná-lo instável.

O termo integral é o valor da integral do erro no tempo multiplicada pela constante K_i. O valor deste termo permanece constante quando o erro recebido é zero, e se altera com valores diferentes de zero. A integral é implementada como sendo a soma do seu valor de saída anterior com a multiplicação do valor de erro atual pelo tempo. O diagrama em blocos da aproximação utilizada para o cálculo da integral de um valor está representado no Diagrama 48.

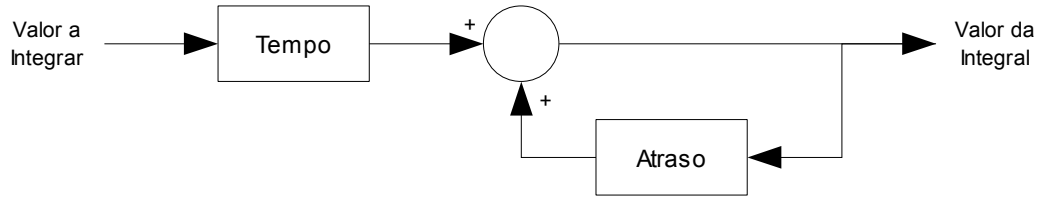


Diagrama 48: Diagrama em blocos da aproximação feita para o cálculo integral.

O termo diferencial consiste na derivada do erro no tempo multiplicada pela constante K_d. O valor deste termo é a diferença entre o valor do erro atual e o valor do erro anterior. A derivada é calculada dividindo a diferença entre o valor de erro atual e o anterior pelo tempo decorrido. O diagrama em blocos da aproximação utilizada para o cálculo da integral de um valor está representado no Diagrama 49.

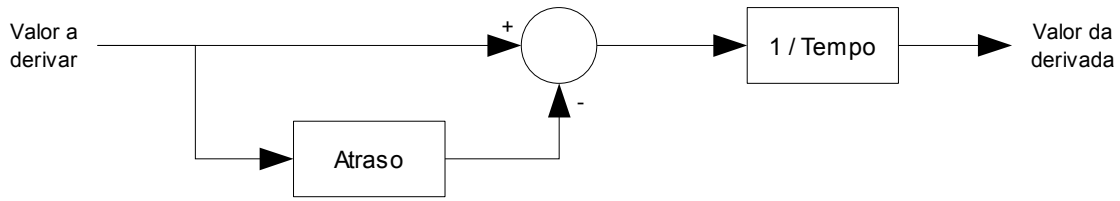


Diagrama 49: Diagrama em blocos da aproximação feita para o cálculo diferencial.

O comando enviado ao sistema implementado permanece em execução até que um novo comando seja enviado, causando um desvio do que seria o valor correto que aumenta com o passar do tempo. Este problema ocorre tanto para a componente integral quanto para a componente diferencial, mas é particularmente notável e muito mais evidente quando o sistema possui componente diferencial.

Para valores normais de constantes, o controlador PID deve minimizar o erro de entrada, mas quando o cálculo demora a ser feito, o valor calculado anteriormente continua valendo e faz a saída passar do valor que seria o correto. Para a parcela integral, isto aumenta um pouco o sobre-sinal e o tempo de amortecimento. Para a parcela diferencial gera um ruído

em forma de onda triangular cuja frequência é metade da frequência em que os comandos são calculados e a amplitude depende diretamente do valor de K_d e do erro instantâneo, e inversamente da frequência de cálculo dos comandos.

A solução para minimizar este ruído é o aumento da frequência do cálculo dos comandos, ou seja, diminuição do tempo entre os comandos. Quanto menor este tempo, melhor será a representação do sistema, que tende à perfeição quando este tempo tende a zero.

Quando o controle é feito por RNA, ela recebe a posição do alvo, a posição atual do braço e o comando anterior que ela produziu. Internamente ela calcula o erro, associa com o comando anterior e calcula o comando atual a ser enviado. Nenhuma equação é explicitamente fornecida para o treinamento, somente as amostras de entradas e saídas cujo comportamento deve ser imitado.

O valor do comando anterior é necessário para que seja possível simular oscilações na saída, pois o tipo implementado de RNA não possui memória de estados anteriores, somente de processamento das entradas.

A RNA pode ser criada com até 100 camadas de até 100 neurônios cada uma. Como cada elemento ocupa uma porção bem reduzida de memória, a criação da rede não deve provocar falta de memória. Entretanto, uma quantidade elevada de elementos aumenta significativamente o custo de processamento. Para o sistema implementado, simulações indicaram que mais de três camadas intermediárias ou mais do que 30 elementos nas camadas intermediárias (neurônios) aumentam muito o custo de processamento sem efetivamente melhorar o desempenho.

Com relação a custo de processamento, a fase de treinamento é muito afetada pelo número de camadas, para cada camada adicionada, o número de passagens necessárias para o treinamento aumenta várias vezes. A fase de execução normal da rede é mais afetada pela quantidade de elementos, mas aumenta um tempo fixo para cada elemento adicionado, que depende do poder de processamento da máquina utilizada.

O braço simulado possui apenas um segmento de comprimento fixo e uma junta composta de duas articulações rotacionais sobrepostas, uma longitudinal e uma transversal. A articulação longitudinal tem o movimento limitado entre -180° e 180°, e a articulação transversal, entre 0° e 90º sendo que em 0°, o braço encontra-se perpendicular ao plano de referência.

O objetivo do braço é atingir um alvo cuja posição é definida pelo usuário em coordenadas cartesianas. As posições do alvo são limitadas à superfície de uma semi-esfera de raio igual ao comprimento do braço, para que possa sempre ser atingido.

Como a junta é rotacional, o sistema de coordenadas utilizado é o de coordenadas esféricas (r, θ, φ) pois o comprimento do do braço é fixo e a posição pode ser descrita com somente uma coordenada esférica. Quando coordenadas cartesianas estão presentes, elas são convertidas em coordenadas esféricas antes de qualquer cálculo, os arredondamentos não interferem significativamente na simulação.

O interesse é provar a capacidade da RNA de calcular comandos para um braço mecânico, portanto o sistema simulado foi simplificado, tanto para facilitar a construção e melhorar a clareza do trabalho quanto para minimizar chance de erros na simulação dos fatores que interferem no controle. Os valores calculados para o controle são imediatamente impostos ao braço como velocidades angulares em cada articulação, assim como ocorre em sistemas reais com motores de passo.

Na simulação, os valores conhecidos para a realização do cálculo do controle são a posição atual do alvo e a posição atual do braço. A RNA recebe, ainda, o valor do comando anterior. O Diagrama 50 apresenta o diagrama em blocos simplificado do sistema. As informações enviadas de um bloco a outro são vetores com um ou mais valores.

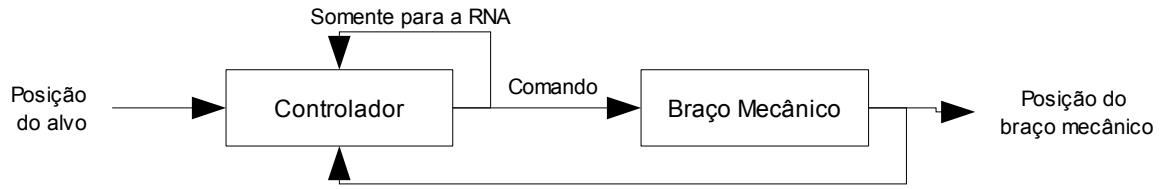


Diagrama 50: Diagrama em blocos simplificado do sistema de controle simulado.

A simulação conta também com um sistema de registro que armazena os conjuntos de entradas e saídas (log) que pode ser gravado em um arquivo para utilização no treinamento da rede neural para controlar o braço, o Diagrama 51 mostra o sistema de log ligado no diagrama apresentado no Diagrama 50.

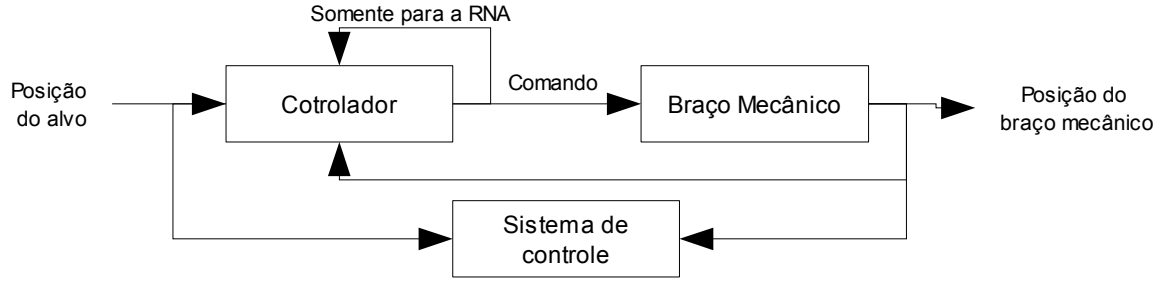


Diagrama 51: Diagrama em blocos do sistema de controle simulado com o bloco de registro dos valores.

O controlador PID aplicado ao controle do braço de um segmento é apresentado no Diagrama 52, o cálculo do comando ocorre em momentos discretos do tempo, e não continuamente, isto acaba provocando o problema descrito acima no termo diferencial do controlador. Na figura, o bloco integrador foi identificado como "1/s" e o bloco derivador como "s" em referência a sistemas de controle aplicados no domínio da frequência.

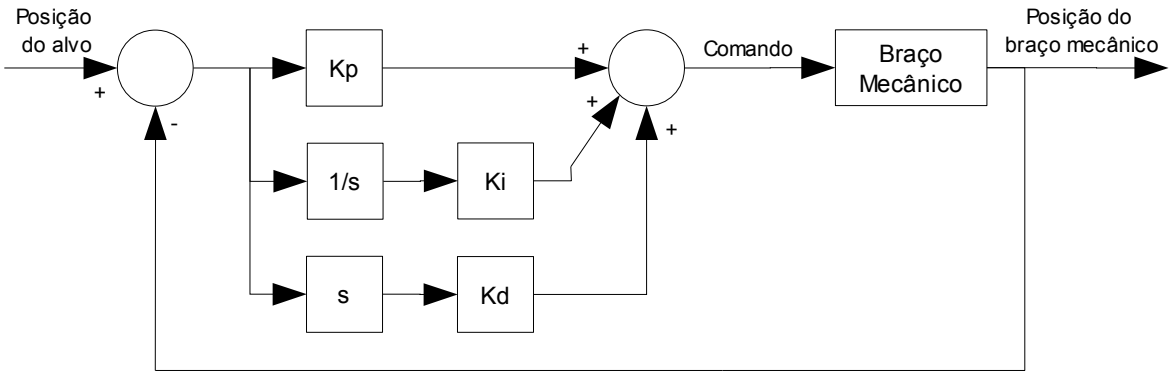


Diagrama 52: Diagrama em blocos do controlador PID no sistema simulado.

A RNA possui um algoritmo de treinamento ativado separadamente da operação do sistema colhendo dados de um log previamente gerado com controles que ela deve imitar, o treinamento se dá antes da utilização da RNA. A Diagrama 53 mostra o diagrama em blocos da RNA implementada.

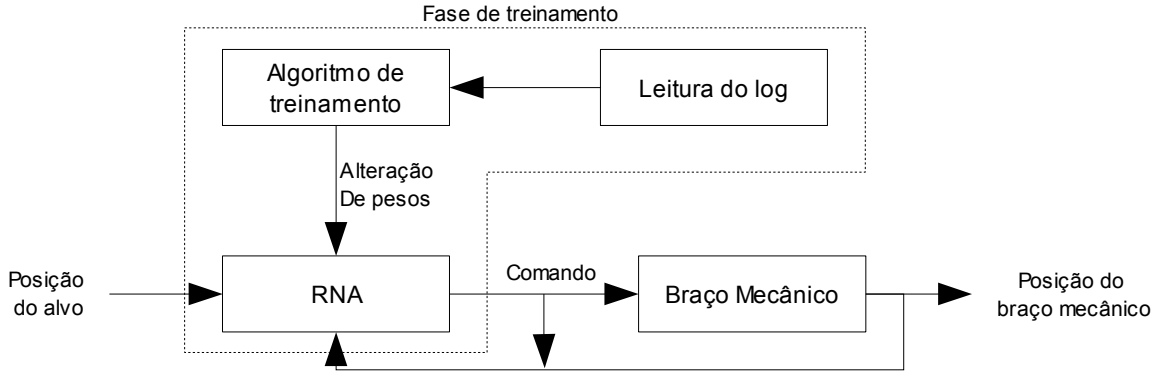


Diagrama 53: Diagrama em blocos do controle do braço de um segmento por RNA.

A finalidade do programa é permitir observar como uma RNA faz o controle de um braço mecânico depois de treinada e avaliar a viabilidade da aplicação ou não em um sistema deste tipo com base em gráficos e observação do comportamento do braço controlado.

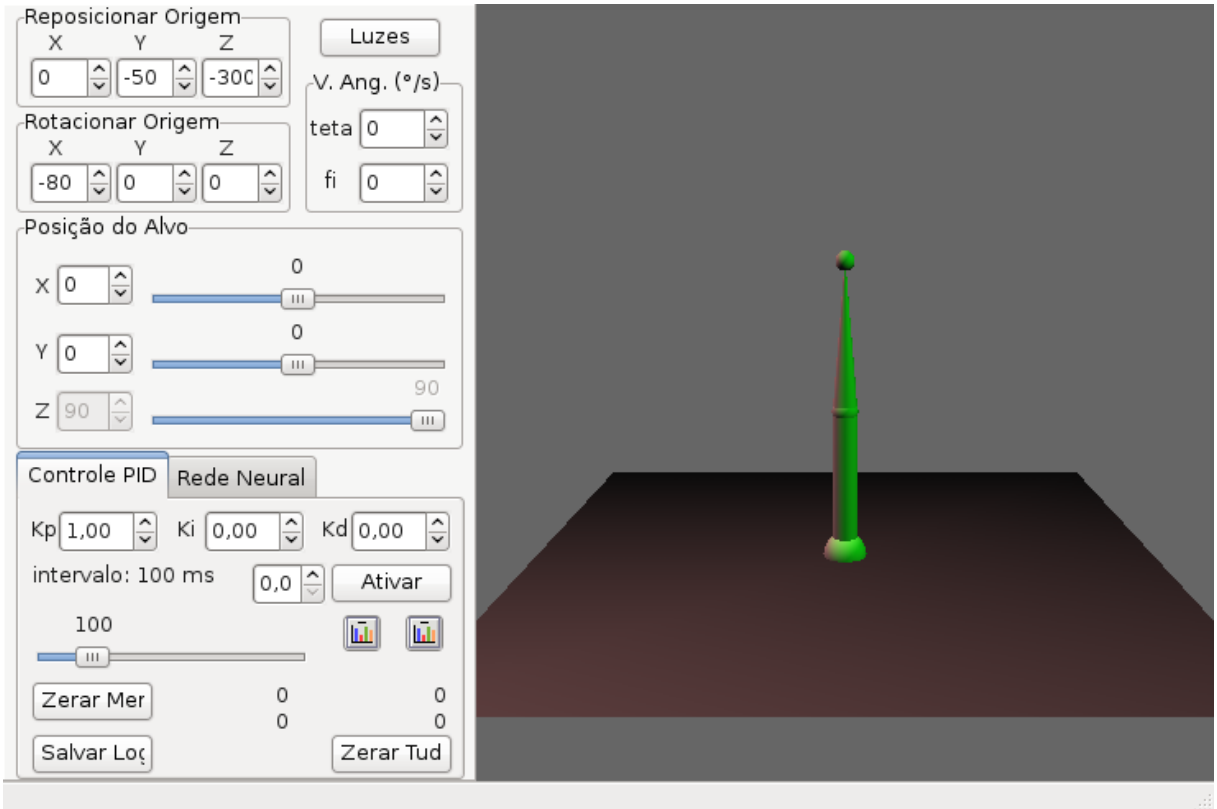
5.2. APRESENTAÇÃO GRÁFICA DO PROGRAMA

As telas do programa serão apresentadas em interface gráfica X11 com ambiente de trabalho Gnome, gerenciador de janelas metacity, decorador de janelas GTK e tema ClearLooks (as bordas das janelas não estarão inclusas), todos podem ser instalados em vários sistemas operacionais do tipo Unix.

O programa simula o controle de um braço mecânico com o único objetivo de colocar a extremidade do braço na posição que se encontra um alvo indicado por uma esfera. A representação é feita por meio de OpenGL com parâmetros variáveis de posições e rotação do sistema (base, braço e alvo) e da iluminação.

Os valores de entrada são as coordenadas do braço e do alvo e os valores de saída, os comandos de rotação do braço, em velocidade angular. O estudo dos sensores e dos atuadores não estão no escopo deste trabalho, deste modo, as posições serão sempre fornecidas corretamente e os comandos de velocidade serão aplicados diretamente ao braço. As unidades utilizadas para comprimento são irrelevantes, já que se trata de uma simulação e não terão nenhuma interação que exija unidades de medida.

Ao abrir, o programa exibe a tela de simulação, mostrada na Tela 1, com os controles à esquerda e a ilustração do sistema à direita.



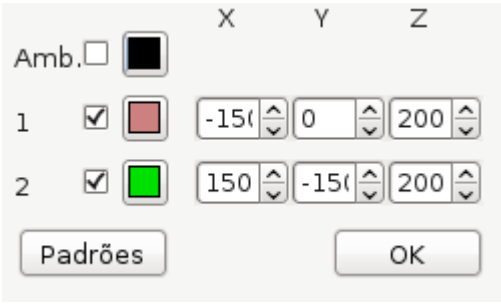
Tela 1: Programa de simulação, janela principal.

Na representação do sistema à direita, o eixo X é horizontal, o eixo Y vertical e o eixo Z decresce no sentido “entrando” na tela e aumenta no sentido oposto.

A caixa “Reposicionar Origem” permite mover todo o sistema em relação à origem do sistema de coordenadas, a posição indicada é assumida pela base do braço, onde encontra o plano XY representado. O valore de Z deve ser negativo para que o sistema seja exibido.

A caixa “Rotacionar Origem” permite rotacionar o sistema depois de ter sido deslocado dos valores da caixa Reposicionar Origem descrita acima. As rotações se dão em torno dos eixos indicados e seguem a regra da mão direita. Outra forma de mudar os valores de rotação é apertar o botão direito do mouse sobre a área de representação do sistema e mover o mouse enquanto segura o botão apertado, o movimento horizontal altera valores de rotação em torno de X e o vertical altera a rotação em torno de Z.

O botão “Luzes” permite, por meio do formulário exibido na Tela 2, ativar, desativar e alterar as cores e as posições das luzes que iluminam o sistema. Se nenhuma luz estiver ativa, todos os elementos são representados em cor sólida sem sombra, o que torna impossível distinguir os objetos representados. A luz ambiente não tem uma origem, pois ilumina iqualmente todas as superfícies do sistema, as outras duas luzes possuem um ponto que é a fonte de luz, com irradiação omnidirecional.



Tela 2: Programa de simulação, janela “Luzes”.

O quadro “V. Ang. (°/s)” permite impor uma velocidade angular de movimento, em graus por segundo ao braço representado. Tanto o controle PID quanto a RNA utilizam as caixas de valores presentes nesta caixa para enviar os comandos.

O quadro “Posição do Alvo” permite alterar a posição do alvo que o braço deve seguir. Como o braço tem comprimento fixo, ele ó pode alcançar a superfície de uma semi-esfera. Para que o alvo sempre possa ser atingido, o valore de Z não pode ser definido pelo usuário, ele é calculado com base nos valores de X e Y. Os valores de X e Y também possuem restrições e em alguns casos , a alteração do valor de um pode alterar valores do outro. Os

valores podem também ser alterados apertando o botão esquerdo do mouse sobre a área de representação e movendo-se o mouse enquanto segura-se o botão apertado. O mouse controla os valores X (para deslocamento horizontal) e Y (para deslocamento vertical).

O quadro “Controle PID” permite configurar o controle PID. Kp, Ki e Kd são as constantes proporcional, integral e diferencial, respectivamente. Para as três constantes, o programa permite a escolha de qualquer valor entre -9,99 e 9,99. Os valores são os mesmos para a coordenada teta e para a coordenada fi. Os números mais abaixo são os valores acumulados para a parcela integral e diferencial, respectivamente, com fi na primeira linha e teta na segunda linha. O botão “Zerar Mem.” define os valores acumulados integral e diferencial para zero.

O valor “Intervalo”, controlado pela barra deslizante logo abaixo, refere-se ao tempo existente entre cada comando do controle, isto é, de quanto em quanto tempo o comando deve ser calculado e atualizado, o valor padrão é 100 ms, o que significa que serão feitos 10 comandos por segundo.

O botão “Ativar” passa a executar o comando PID, a caixa de valor do lado esquerdo do botão, indica o tempo, em segundos que o sistema terá o controle ativado, exceto se o valor for zero no início, situação em que o controle fica ativo até ser interrompido, seja pressionando o botão novamente, seja mudando para a caixa de controle por RNA.

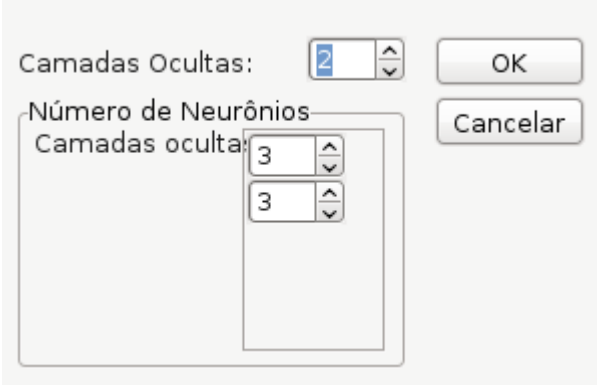
Os botões com barras coloridas, próximos ao botão “Ativar”, exibem, cada um, um gráfico com os valores de posição e controle, um em coordenadas cartesianas e outro em coordenadas polares, que por padrão não são exibidos, pois consomem uma quantidade considerável de processamento se comparado com o resto do programa, porém, os valores são atualizados mesmo quando o formulário não está sendo exibido, assim sendo, podem ser exibidos ao final de uma simulação para verificação dos últimos valores disponíveis.

O botão “Salvar Log” permite salvar em um arquivo, um log com todos os conjuntos de entradas (posições) e saídas (controle) que estão na memória. Enquanto o sistema está ativo, todos os conjuntos de valores de entrada e saída são gravados até o limite de 10240 conjuntos. Quando este número é alcançado, a entrada mais antiga é excluída para acomodar uma nova entrada.

O botão “Zerar Tudo” desativa o controle, define as velocidades angulares e as parcelas integral e diferencial do controle para zero e move o alvo e o braço para as posições iniciais, os valores das constantes Kp, Ki e cadê não são alterados.

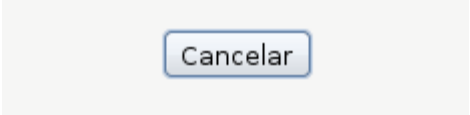
O quadro “Rede Neural”, que fica junto com o quadro “Controla PID” (só é possível exibir um de cada vez, clicando-se no título do quadro desejado), possui os campos utilizados para o uso da RNA como controlador do sistema.

O botão “Criar Nova” cria uma nova rede neural com quantidades de camadas e de neurônios de cada uma definidas pelo usuário, exceto para as camadas de entrada e saída que são definidas de acordo com o número de entradas e saídas do sistema (são quatro entradas e duas saídas). O formulário onde o usuário define as quantidades de camadas e neurônios está representado na Tela 3.



Tela 3: Programa de simulação, janela “Nova Rede Neural”.

O botão “Treinar” solicita ao usuário selecionar um arquivo com os dados de treinamento e inicia o treinamento da rede assim que o arquivo é aberto e um formulário “Processando...” como na Tela 4 é exibido com o botão “Cancelar” ao centro, este permite que o treinamento seja interrompido a qualquer momento. A cada 100 passagens pelo conjunto de treinamento, a barra de status da janela principal (parte inferior) é atualizada com informações sobre a quantidade de passagens completas do arquivo de treinamento e o erro atual. O Erro é um valor adimensional somente para referencia de quão próximo está o comportamento da RNA sendo treinada do conjunto de dados de treinamento (i.e. qual a capacidade da rede de reproduzir as saídas do treino tendo como base as entradas).



Tela 4: Programa de simulação, janela “Processando...”.

Um valor ligeiramente menor que dois é uma boa aproximação ao conjunto de treinamento. Há um limite de 50.000 passagens pelo arquivo de treinamento. Quando o treinamento é interrompido ou atinge o limite, a barra de status é novamente atualizada com a

quantidade total de passagens e o erro final obtido. O botão “Treinar (b)” é exatamente igual ao botão “Treinar”, exceto que utiliza modo batch.

O botão “Ver Pesos” exibe todos os pesos da RNA com um texto em uma janela separada, e pode ser usado a qualquer momento, exceto durante o treinamento, quando a única ação possível é clicar no botão cancelar.

Os outros controles da caixa “Rede Neural” são equivalentes aos controles da caixa “Controle PID”, o controle de intervalo entre os comandos, o botão ativar, o controle de tempo que o sistema estará ativo e os botões de gráficos têm a mesma funcionalidade. O botão “Zerar Tudo” tem funcionalidade parecida, aqui ele redefine as velocidades e posições iniciais do braço e alvo, mas não altera a RNA.

5.3. UTILIZAÇÃO DO PROGRAMA

A simulação pode ser executada com o controlador PID ou com a RNA, sendo que é necessário criar dados de treinamento antes de realizar a simulação com a RNA, os dados de treinamento possuem entradas e saídas com um comportamento que a RNA deve imitar, e poderiam ser gerados de qualquer forma, a maneira mais fácil encontrada foi gerar a partir do controlador PID.

Os parâmetros de visualização não afetam a simulação, independem do tipo de controle utilizado, e podem ser alterados durante a simulação, são eles a posição da origem, a rotação do sistema e as definições de luzes.

Os controles de posição do alvo e velocidade de movimento estão sempre presentes, e são utilizados na simulação, portanto, afetam a simulação se forem alterados. Durante a simulação, a posição do alvo deve ser modificada para observar o comportamento do braço, a velocidade angular não deve ser alterada, pois afeta o comportamento.

Os controles de velocidade estão presentes para que o usuário possa ter uma indicação fácil das saídas do controlador e para poder mexer o braço manualmente caso desejado, mas se o sistema estiver ativo, a próxima saída do controlador sobrescreve a velocidade que o usuário definir.

Para análise de resposta do sistema, a maneira mais fácil é definir os parâmetros do controlador depois, com o sistema desativado, colocar o alvo em um local diferente da posição do braço, ativar o sistema opcionalmente colocando um tempo para ele parar, desativar o sistema e depois abrir as janelas de gráficos para observar os resultados.

5.3.1. Procedimentos para Simulação com Controlador PID

A caixa de “Controle PID” deve estar sendo exibida (na janela principal, clicar na aba referente se não estiver), as etapas são:

Passo 1: Definir os parâmetros Kp, Ki e cadê conforme desejado;

Passo 2: Definir o tempo entre os cmandos;

Passo 3 (Opcional): Definir um tempo depois d qual o controle será desativdo;

Passo 4: Clicar no botão “ativar”;

Passo 5: A qualquer momento, clicar novamente no botão “ativar” para interromper o process ou aguardar o término do tempo definido no passo 3.

Passo 10: Se desejar, salvar o Log de comandos para uso para treinamento de uma RNA.

A posição do alvo pode ser alterada a qualquer momento, os gráficos podem ser exibidos também a qualquer momento, mas impactam muito no processamento exigido, portanto recomenda-se que sejam exibidos somente com o controle desativado.

Se, em algum momento o sistema entrar em um estado indesejado pelo usuário, provocado pelos valores dos cálculos das parcelas integral e diferencial, o botão “Zerar Mem” pode ser utilizado para redefinir os valores acumulados para zero.

O botão “Salvar Log” permite salvar um log com os últimos 10240 comandos (não é necessário atingir este número de comandos, podem ser usados menos comandos), e o arquivo resultante pode ser utilizado para treinar uma RNA criada nesta mesma janela. A qualidade do treinamento resultante com o arquivo criado dependerá dos parâmetros utilizados, da quantidade e diversidade de comandos e posições do alvo, normalmente, quanto maior a quantidade e a diversidade de comandos e posições, melhor o treinamento resultante.

5.3.2. Procedimentos para Simulação com Controle por RNA

A caixa de “Rede Neural” deve estar sendo exibida (na janela principal, clicar na aba referente se não estiver), as etapas são:

Passo 1: Criar uma rede neural clicando no botão “Criar Nova” e inserindo as quantidades de camadas e neurônios;

Passo 2: Treinar a rede neural clicando no botão “Treinar”;

Passo 3: Quando o erro se reduzir a um valor aceitável, interromper o treinamento clicando no botão “Cancelar” do formulário “Processando...”;

Passo 4: Definir um intervalo entre os cálculos de controle;

Passo 5 (Opcional):: definir um tempo depois do qual o controle será desativado;

Passo 6: Clicar no botão ativar;

Passo 7: A qualquer momento clicar no botão desativar ou, se houver um tempo definido no passo 4, aguardar o término do tempo.

O alvo pode ser movido e os gráficos e pesos da rede podem ser exibidos a qualquer momento, exceto quando na fase de treinamento, a exibição dos gráficos tem impacto no processamento.

No passo 3, quando a rede já está treinada a um nível desejado, o treinamento pode ser interrompido clicando-se no botão "Cancelar" da janela que aparece ao início do treinamento. Se o treinamento atinge 50.000 épocas, ele é interrompido automaticamente. O treinamento pode ser repetido, mas um conjunto de dados diferente pode causar saturação dos neurônios por ter possuir saídas muito diferentes para entradas muito semelhantes às já treinadas.

O valor de erro é a média das somas dos erros quadráticos das saídas para cada conjunto de treinamento do arquivo selecionado, não há um valor determinado pois fatores como a própria natureza dos dados podem fazer com que a rede se comporte de forma desejada mas com erros em relação aos dados de treinamento, um exemplo é quando há oscilação no conjunto de treinameno e a RNA está operando sem apresentar a oscilação.

Nas simulações executadas, quando havia oscilação presente, o erro, em algumas situações não baixou de 16, apresentando comportamento razoável, sendo que na simulação puramente proporcional, o erro teve que estar abaixo de 2 para que o comportamento fosse adequado.

É importante notar que quando há ruído presente, o treinamento pode aumentar o erro após tê-lo reduzido, portanto torna-se recomendável a interrupção do treinamento quando o erro está com um valor adequado. O treinamento pode ser interrompido para fazer testes e depois reiniciado com o mesmo arquivo, o treinamento anterior não é descartado.

6. SIMULAÇÕES

Para verificar a capacidade da RNA em aprender os comandos de acordo com as entradas, algumas simulações foram realizadas. As simulações não são focadas no controle PID, mas o utilizam como ferramenta para gerar dados para treinar a RNA.

O equipamento utilizado pode influenciar os tempos necessários para os procedimentos. Para as simulações deste trabalho, foi utilizado um computador com processador Athlon64 3700+, memória DDR400 operando em dual channel, a placa de vídeo faz processamento do OpenGL, assim, a representação do sistema não influencia no processamento ou o faz de forma desprezível. O sistema operacional (que também pode influenciar) é o Linux variante Gentoo de 64 bits com kernel compilado com configurações personalizadas.

6.1. PREPARAÇÃO E DEFINIÇÃO DE PARÂMETROS

Para a simulação e verificação da eficácia do controle por meio de RNA, foram adotados parâmetros a serem utilizados para criar dados de treinamento e comparação com controle PID.

A comparação de resultados foi feita com um único movimento do braço partindo do repouso com ponta na posição (0; 0; 90) e indo até o alvo posicionado em (-60; 65; 16), em coordenadas esféricas (90; 0°; 0°) e (89,89; 132,71°;79,75°), respectivamente. Este movimento foi acompanhado durante cinco segundos, independentemente da característica apresentada ou se o objetivo foi atingido.

Todos os gráficos são apresentados em coordenadas polares, pois neste programa, é onde se pode observar o comportamento do controle.

O log para treinamento foi feito com intervalo entre controles de 50 ms (resultando em uma taxa de amostragem de 20 Hz). O número de amostras ficou um pouco elevado mas o ruído proveniente da parcela diferencial foi reduzido. Para a exibição dos resultados, o intervalo entre controles foi definido em 20 ms (resultando em uma taxa de amostragem de 50 Hz), que resulta em boa resolução dos gráficos para análise dos resultados.

Foram considerados cinco situações de controle PID com diferentes características, representados por diferentes valores de Kp, Ki e Kd. A Tabela 1 mostra os valores adotados para cada simulação.

Característica do controle	Kp	Ki	Kd
P	3	0	0
PI	3	2	0
PD	3	0	0,9
PID	3	2	0,9

Tabela 1: Fatores utilizados nas simulações de controle PID para o braço de um segmento. P = Proporcional; I = Integral; D = Diferencial

Para cada tipo de controle foi criado um arquivo de treinamento, movendo-se o alvo e deixando que o braço o alcançasse algumas vezes durante alguns segundos em locais variados e distribuídos ao longo das faixas possíveis, incluindo os limites de teta (-180° e 180°) mas sem regularidades ou simetrias e principalmente desconsiderando o movimento utilizado para teste como referência (dados para este movimento não foram explicitamente gerados).

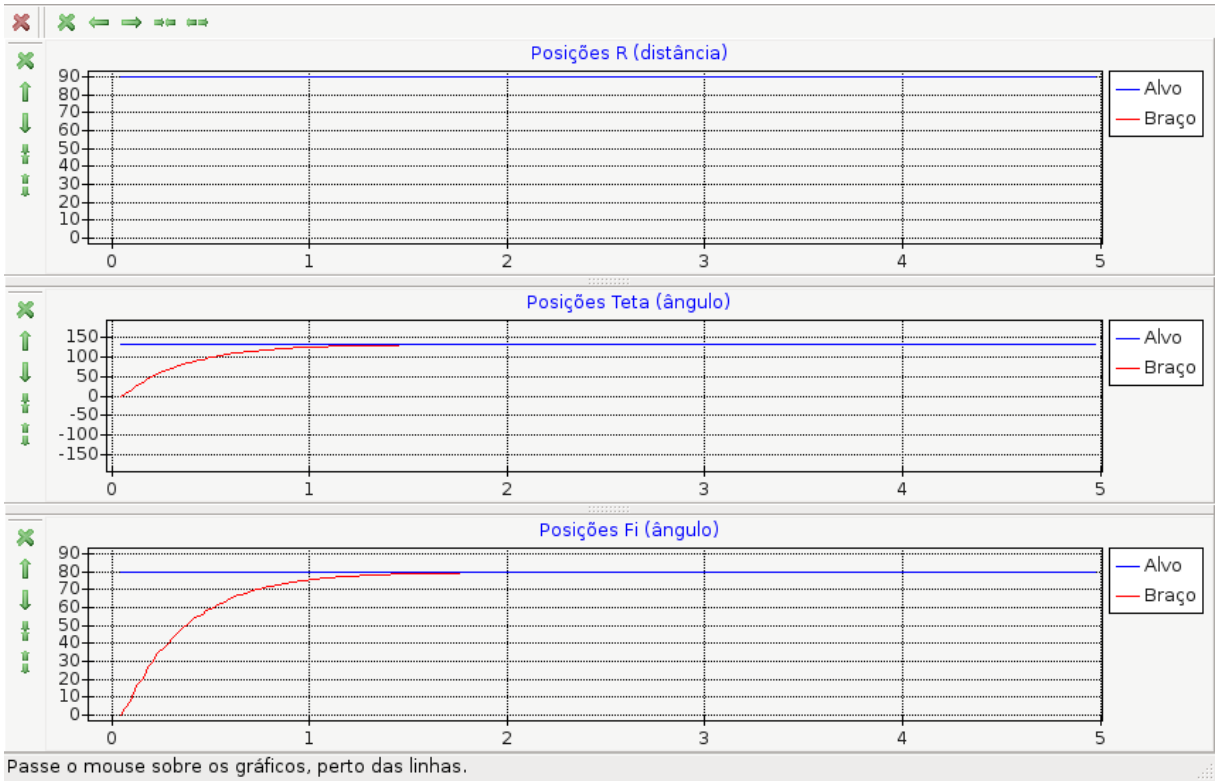
Os movimentos do alvo foram, ainda, feitos cuidadosamente, de modo a evitar ruídos provenientes do degrau de entrada do erro, para que a influência do ruído fosse a menor possível, sobretudo nos controles que incluem característica diferencial, e para haver sempre pouco sobressinal nos controles que incluem característica integral.

Uma única configuração de rede neural foi utilizada, com camadas intermediárias de 15 e 10 neurônios, a camada de entrada é fixa em 6 neurônios (ângulos das coordenadas do alvo, do braço e comando anterior) e a camada de saída fixa em 2 neurônios (comando), deste modo, o número de neurônios em cada camada é {6; 15; 10; 2}.

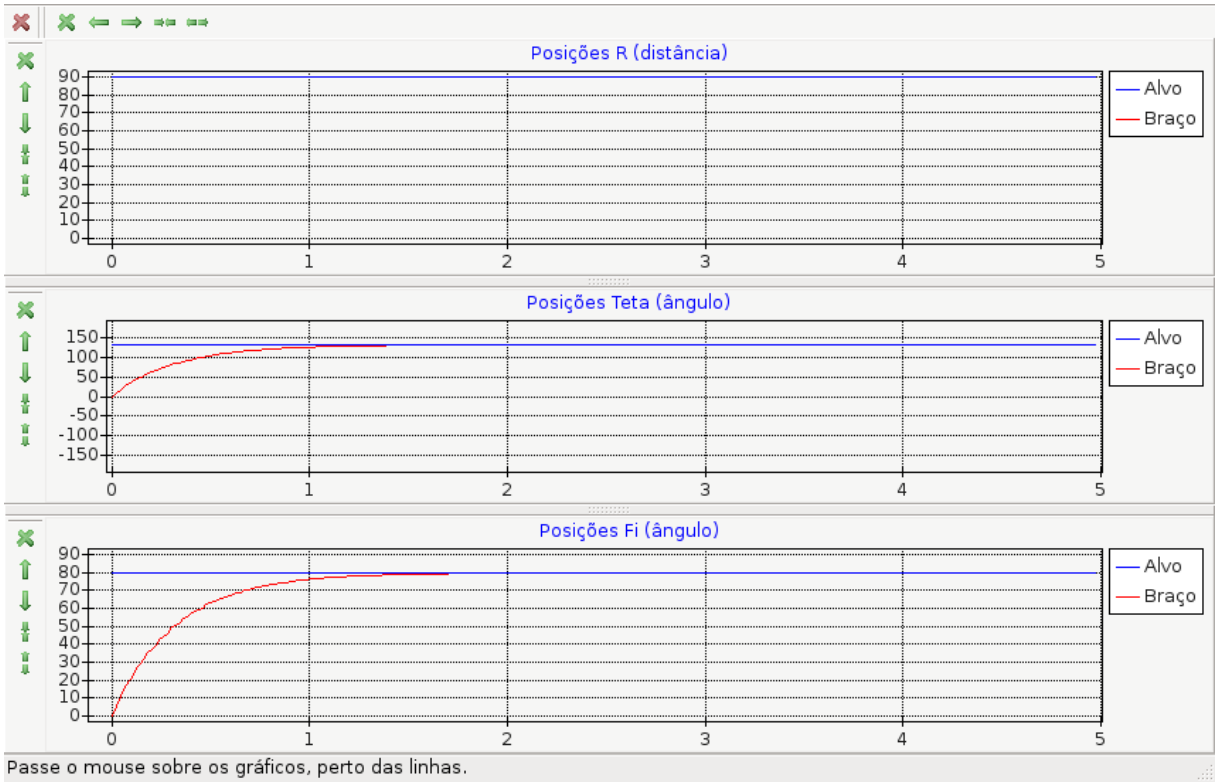
O treinamento não teve tempo fixo para cada tipo de controle, o critério para a interrupção do treinamento foi o erro mostrado na barra de status da janela principal do programa. Num primeiro treinamento o erro foi observado para determinar se ele começava a aumentar em algum momento do treinamento (sem tornar a diminuir), o menor erro era anotado e um novo treinamento foi feito. Quando o erro estava próximo ao valor anotado, o treinamento foi interrompido.

6.2. RESULTADOS E ANÁLISES

Na simulação com característica proporcional a RNA teve comportamento muito aproximado do comportamento do controle proporcional, Telas 5 e 6. O treinamento foi muito rápido e o erro exibido pelo programa ficou muito pequeno, com valor de cerca de 0,2. Como o sistema era simples, este resultado já era esperado.

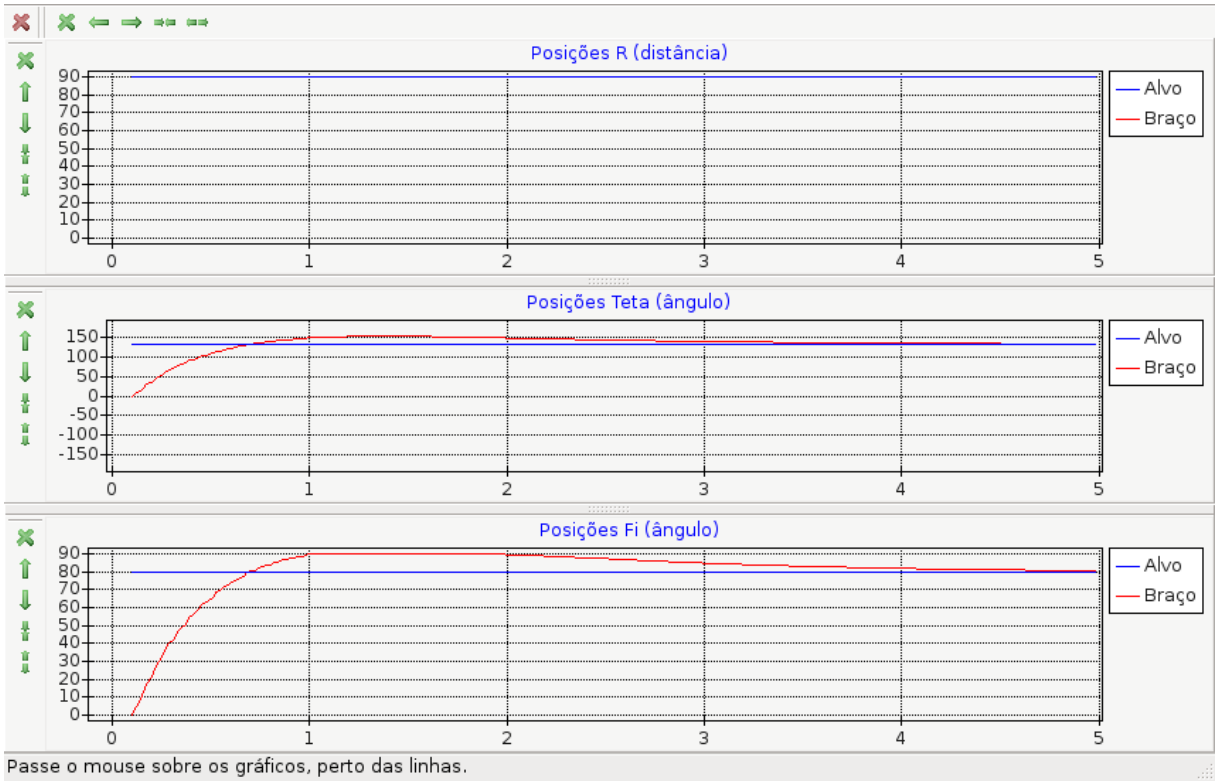


Tela 5: Resultados da simulação do braço mecânico com controle proporcional.

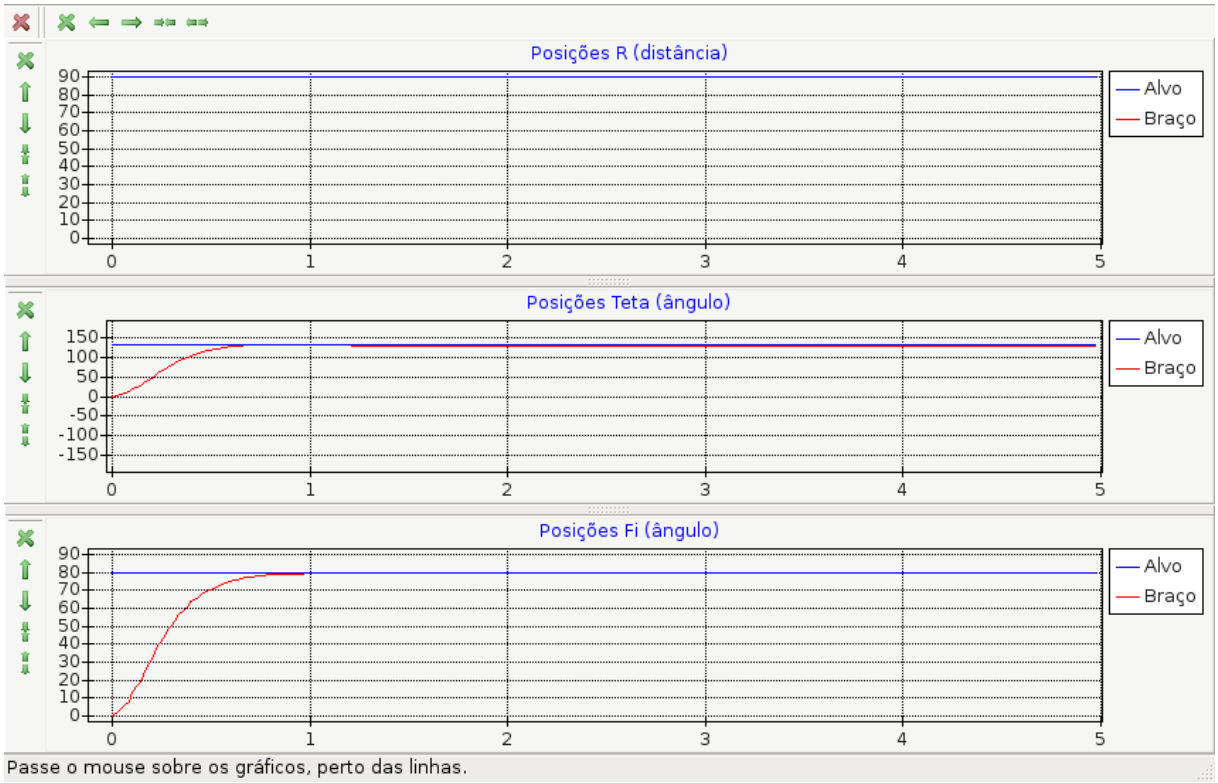


Tela 6: Resultados da simulação do braço mecânico com controle por RNA simulando controle proporcional.

Na simulação com controle com componente integral, embora a RNA parece ter eliminado o sobressinal do próprio sistema que simula, Telas 7 e 8.



Tela 7: Resultados da simulação do braço mecânico com controle proporcional-integral.



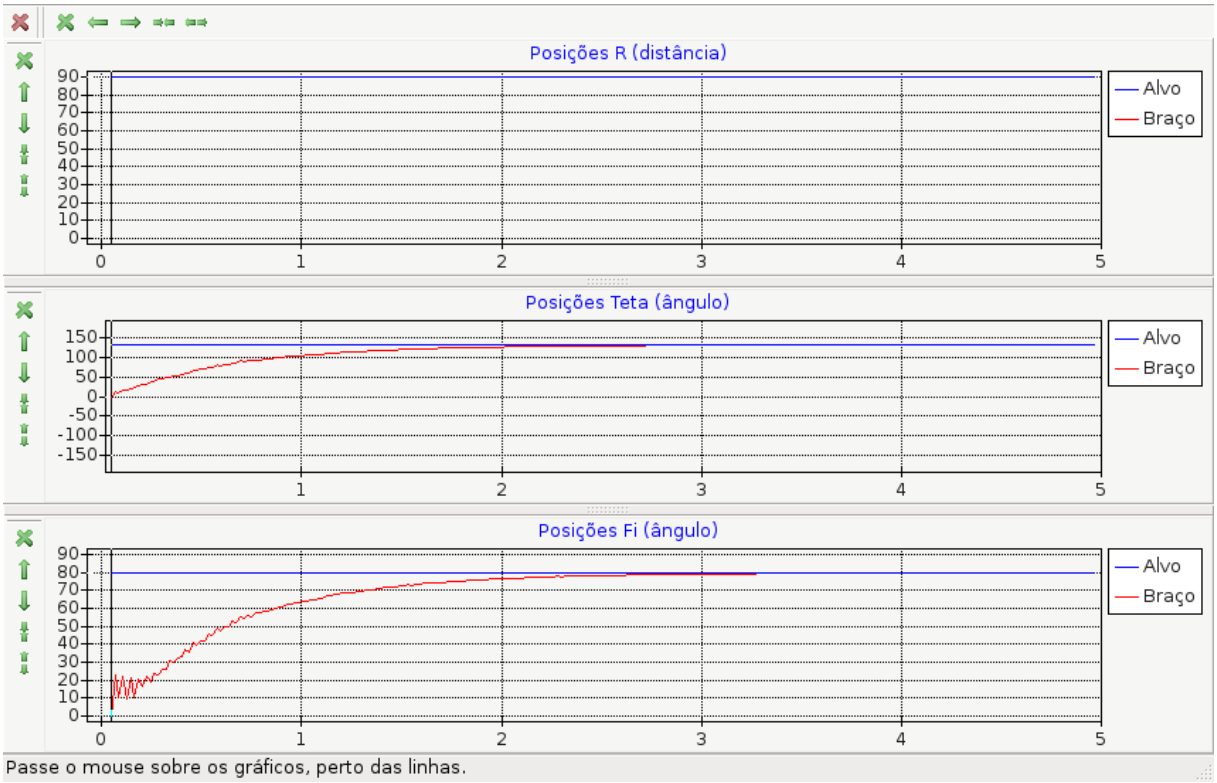
Tela 8: Resultados da simulação do braço mecânico com controle por RNA simulando controle proporcional-integral.

O comportamento da RNA neste caso foi melhor que o esperado, o motivo foi aparentemente o modo como foi criado o conjunto de treinamento. Os movimentos foram

feitos com o alvo após o braço passar por ele, no sentido em que o erro diminuía, desta forma compensando o sobressinal. O braço foi deixado próximo ao alvo por um período de tempo, assim a RNA adquiriu um comportamento de levar o braço ao alvo mas não copiar o sobressinal. O erro final que pode ser observado na Tela 8 pode também ter sido provocado pelo conjunto de treinamento, mas ainda assim ficou muito próximo do objetivo.

Para o controle proporcional-integral a rede neural definitivamente não se comporta como o controle com o qual os dados de treinamento foram gerados, mas obteve a melhor resposta de todas as simulações. Isto ocorreu provavelmente pela forma como foram gerados os dados de treinamento, Telas 9 e 10.

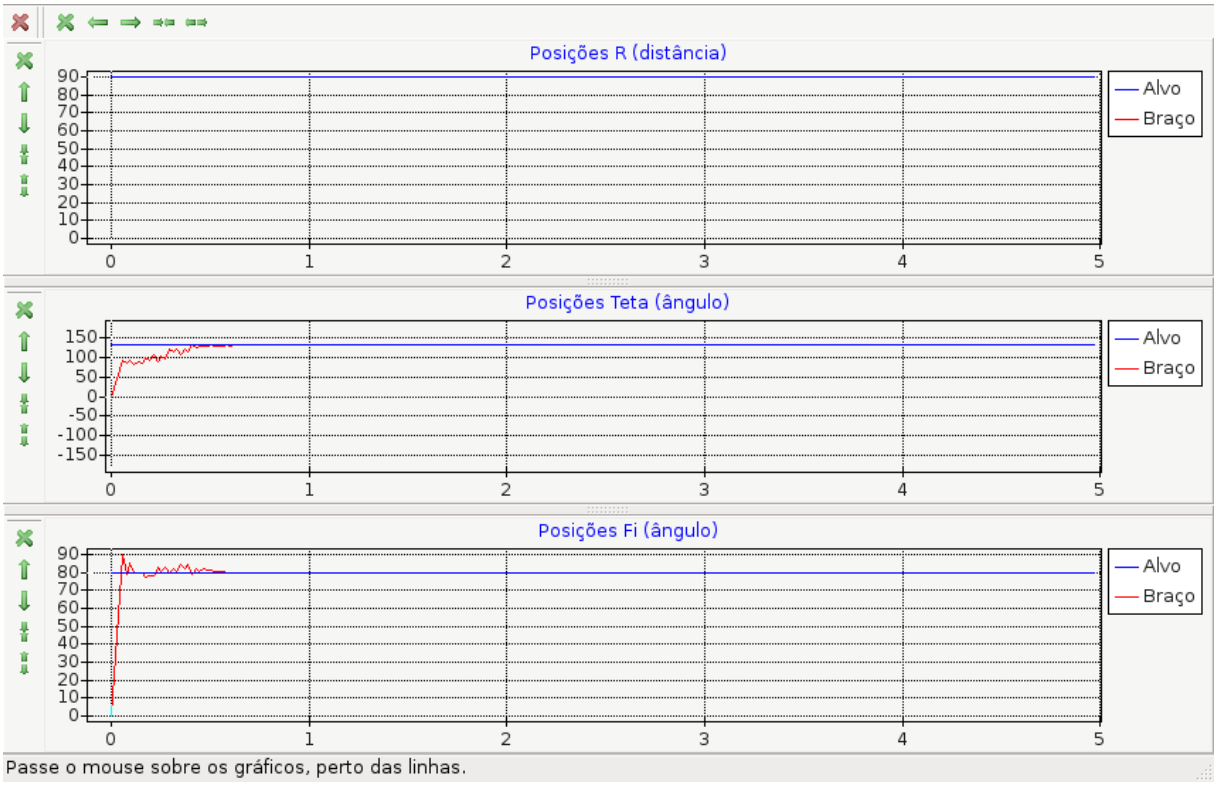
Para este tipo de controle o resultado foi ainda melhor que o anterior. Novamente o modo como o conjunto de treinamento foi criado parece ter influenciado neste resultado. Os movimentos feitos eram sempre curtos para evitar o ruído criado pela parcela diferencial, e o braço era deixado no alvo por um pequeno período de tempo, mas que resultavam em várias amostras do conjunto de treinamento.



Tela 9: Resultados da simulação do braço mecânico com controle proporcional-diferencial.

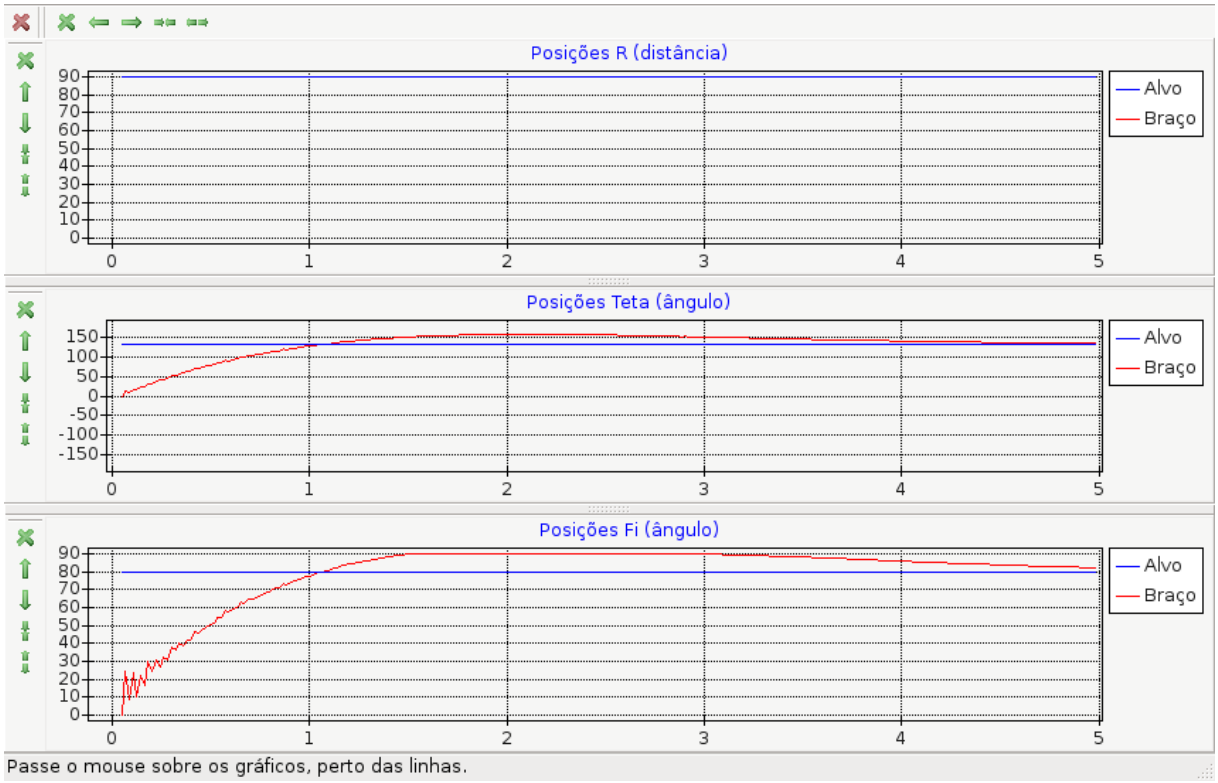
A simulação apresentada possui intervalo de 20 ms entre os comandos e os dados de treinamento foram criados com 50 ms, isto resultava em um ruído muito maior que os podem ser observados na Tela 9. Ainda é possível notar que a RNA copiou ruído presente

no conjunto de treinamento, como pode ser observado na Tela 10, está claro que ele não foi eliminado.

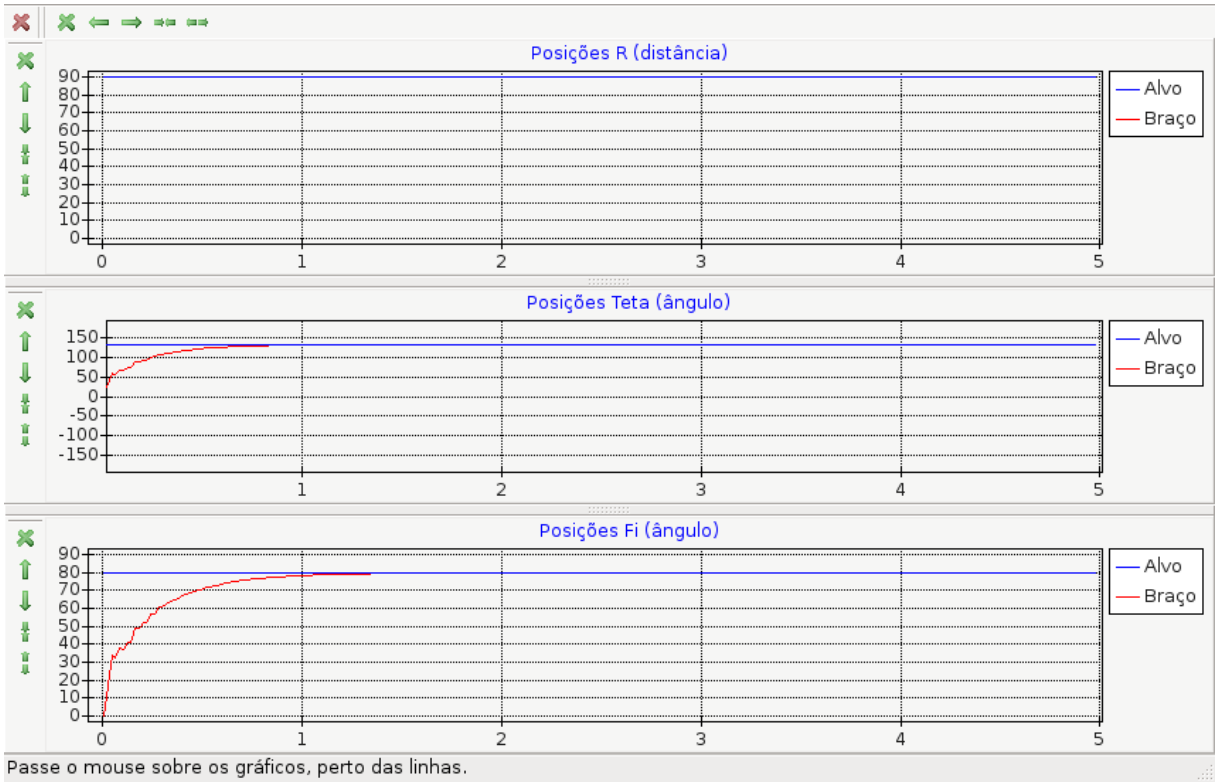


Tela 10: Resultados da simulação do braço mecânico com controle por RNA simulando controle proporcional-diferencial.

A RNA também apresentou comportamento muito bom para o controle proporcional-integral-diferencial, parecendo eliminar o ruído e o sobressinal, Telas 11 e 12.



Tela 11: Resultados da simulação do braço mecânico com controle proporcional-integral-diferencial.



Tela 12: Resultados da simulação do braço mecânico com controle por RNA simulando controle proporcional-integral-diferencial.

Em relação ao controlador proporcional-integral-diferencial, a RNA apresentou menor ruído, iniciou o comando com um movimento mais rápido e não mostrou sobressinal.

Foi um comportamento intermediário entre o que a RNA teve ao simular o controlador proporcional-integral e ao simular o controlador proporcional diferencial, o que faz sentido.

O Quadro 3 apresenta os tempos necessários para que o braço mecânico simulado atingisse a posição do alvo e para que se estabilizasse, para cada controlador utilizado e observações sobre o resultado. Considerou-se atingir a posição do alvo o primeiro momento em que a posição do braço mecânico se sobrepõe à posição do alvo, no gráfico, e estabilização quando o gráfico deixa de registrar alterações na posição do braço mecânico, sempre na coordenada que levou o maior temo.

Controlador	**Atingiu alvo em (s)**	**Estabilizou em (s)**	**Observações**
Proporcional.	1,7	1,7	
RNA simulando controlador proporcional.	1,7	1,7	
Proporcional-integral	0,6	5,0	
RNA simulando controlador proporcional-integral	1,0	1,2	Houve erro no valor final de θ após a estabilização. Não houve sobressinal.
Proporcional-diferencial	3,2	3,2	
RNA simulando controlador proporcional-diferencial	0,4	0,6	Muito mais eficiente que o controlador proporcional-diferencial utilizado para gerar os dados de treinamento.
Proporcional-integral-diferencial	1,0	> 5,0	
RNA simulando controlador proporcional-integral-diferencial	1,4	1,4	Não houve sobressinal.

Quadro 3: Tempos de ação e observações para os controles simulados.

Em todas as simulações a RNA conseguiu resultados iguais ou melhores que o próprio controle utilizado para gerar o conjunto de treinamento.

7. CONCLUSÕES

A utilização de uma linguagem de programação orientada a objeto permitiu expor a implementação das classes que executam as funções da RNA estudada de forma clara e dividindo as funcionalidades e propriedades de cada tipo de objeto, desta forma facilitando o entendimento do código e seu funcionamento.

A escolha do ambiente de desenvolvimento Lazarus foi muito benéfica. É uma interface que une uma facilidade de uso comparável aos ambientes de desenvolvimentos mais sofisticados, compatibilidade entre vários sistemas operacionais e desempenho dos programas criados equivalentes ao das ferramentas mais avançadas e eficazes, além de ser um projeto de código aberto que pode ser estudado ou mesmo modificado e redistribuído, desde que obedecendo as licenças a que está submetido. Apesar de ainda estar em desenvolvimento, apresenta uma boa estabilidade, mesmo que alguns componentes ainda necessitem aprimoramentos.

O uso do Lazarus, além de ajudar a mostrar que é adequado para o desenvolvimento rápido e fácil de aplicações e que pode ser utilizado mesmo sem um treinamento avançado, rompe a limitação criada por ferramentas destinadas a um único sistema operacional, e, desta forma, abrange maiores possibilidades de acesso ao conteúdo deste trabalho, podendo assim, destinar-se a um maior público-alvo.

A utilização de RNA's é sempre semelhante, ela recebe informações processa e devolve a resposta às entradas, e se estiver sob treinamento, os pesos são atualizados para aperfeiçoar o processamento caso seja adequado.

Existe necessidade de um processo de treinamento da RNA, que pode ocorrer antes da aplicação da RNA ao sistema ao qual é destinada ou durante o funcionamento, porém, mesmo para outros tipos de controle esta necessidade também está presente, para ajuste anterior ao funcionamento, ou durante o mesmo. Para RNA's, conjuntos de dados de treinamento podem ser utilizados para definir um comportamento que será copiado, mesmo que seja complicado obter equações que o descrevam, porém a presença de ruídos nas amostras pode dificultar ou impedir o uso de RNA's.

O controle do braço mecânico foi executado com sucesso pela rede neural proposta. Os resultados mostram que é possível fazer o controle de braços mecânicos por meio de RNA's do tipo MLP, neste trabalho mostrado sem considerar inércia, atrito e gravidade. Mesmo que o comportamento melhorado tenha outras causas associadas

desconhecidas, os resultados mostram que RNA's do tipo MLP podem apresentar comportamento melhor do que o controlador proposto e implementado.

Embora um controle PID tenha sido utilizado para gerar os dados de treinamento, nada impede a utilização de um outro tipo de controle ou simplesmente valores desejados em posições determinadas do braço e do alvo, desde que sejam em suficiente número para o treinamento da RNA. A quantidade de informação para treinamento não pode ser determinada, pois depende de características e da complexidade do sistema em que será aplicada.

Prosseguindo na evolução deste trabalho, iniciou-se o estudo de controle de braço mecânico de dois segmentos (dois graus de liberdade a mais) por RNA, mas este estudo ainda está em fase inicial, e os dados são insuficientes para publicação.

Este trabalho foi desenvolvido em plena concordância com o código de ética do IEEE disponível em <http://www.ieee.org/portal/pages/iportals/aboutus/ethics/code.html> (site em inglês).

A licença definida para os códigos-fonte, embora, à primeira vista, possa parecer violar o primeiro item deste código, tem exatamente o efeito contrário, vetando o uso em aplicações que poderiam causar danos quaisquer. Considera-se também que não há forma de impedir maus usos embora sejam explicitamente desautorizados, portanto, está presente a isenção de responsabilidade. Espera-se que todos sigam este código de ética, inibindo o mau uso do conhecimento e incentivando a evolução da humanidade.

Como proposta para trabalhos futuros sugere-se adicionar mais graus de liberdade ao braço, incluir atrito, inércia ou outros fatores ignorados para as simulações deste trabalho, utilizar a RNA desenvolvida para outras aplicações ou estudar a possibilidade de utilizar outros tipos de RNA para esta mesma aplicação.

REFERÊNCIAS

ABB, *ABB IRB 2400 - Robots (Robótica),* [S.l.]: c2010. il. color. Disponível em: <http://www.abb.com.br/product/seitp327/657d58e39c804f64c1256efc002860a7.aspx>. Acesso em: 19 mai. 2010.

ASANO, M.; YAMAMOTO, T.; OKI, T.; KANEDA, M. A Design of Neural-Net Based Predictive PID Controllers. *Systems, Man, and Cybernetics, 1999. IEEE SMC '99 Conference Proceedings. 1999 IEEE International Conference on,* Tokyo, v. 4, p. 1113-1118 vol.4, out. 1999.

BONACORSO, N. G. *Automação Eletropneumática.* 4. ed. São Paulo: Érica, 1997.

CAEIRO, C. M. et al. *Estudo sobre Inteligência Artificial.* [S.l.]: [2002?]. Disponível em: <http://www.citi.pt/educacao_final/trab_final_inteligencia_artificial/index_centro.html>. Acesso em: 19 mai. 2010.

CHRISTODOULOU, C.; GEORGIOPOULOS, M. *Application of Neural Networks in Electromagnetics.* Norwood: Artech House, Inc., 2001.

FAHLMAN, S. E. *An Empirical Study of Learning Speed in Back-Propagation Networks.* [S.l]: 1988. Disponível em: <http://www.foretrade.com/Documents/quickprop%20qp-tr.pdf>. Acesso em: 19 mai. 2010.

FARRELL, J. A. Neural Control. In: LEVINE, W. S. (Ed.). *The Control Handbook.* 3. ed. Boca Raton: CRC Press, Inc., 1996. p. 1017-1030.

FAUSETT, L. V. *Fundamentals of Neural Networks:* architectures, algoritms, and applications. New Jersey: Prentice-Hall, 1994.

FRÖHLICH, J. *Neural Net Components in an Object Oriented Class Structure.* Fachhochschule Regensburg, Regensburg, 1996. Disponível em: <http://fbim.fh-regensburg.de/~saj39122/jfroehl/diplom/e-index.html>. Acesso em: 19 mai. 2010.

FU, L. *Neural Networks in Computer Inteligence.* Singapore: McGraw-Hill, 1994.

GUPTA, K. C. *Mechanics and Controls of Robots.* New York: Springer, 1997.

HÄGGLUND, T.; ÅSTRÖM, K. J. Automatic Tuning of PID Controllers. In: LEVINE, W. S. (Ed.). *The Control Handbook.* 3. ed. Boca Raton: CRC Press, Inc., 1996. p. 824-825.

HARVEY, R. L. *Neural Networks Principles.* New Jersey: Prentice-Hall International inc., 1994.

HOMERO. *Iliada de Homero em Verso Portuguez.* Tradução: Manoel Odorico Mendes, Edição e revisão: Henrique Alves de Carvalho. Rio de Janeiro: Typographia Guttenberg, 1874.

HORGAN, J.The Consciousness Conundrum. *IEEE Spectrum,* New York, v. 45, n. 6, p. 28-33, jun. 2008.

INTELIGÊNCIA ARTIFICIAL. In: Wikipédia: A Enciclopédia livre. [S.I.]: 2010. Disponível em: <http://pt.wikipedia.org/wiki/Intelig%C3%AAncia_artificial>. Acesso em: 19 mai. 2010.

KOCH, C.; TONONI, G. Can Machines Be Conscious?. *IEEE Spectrum,* New York, v. 45, n. 6, p. 47-51, jun. 2008.

LEMES, N. H. T. ; BRITO, N. S. Um estudo preliminar sobre a aplicação de redes neurais artificiais na química. In: *III Amostra de Iniciação Científica da UNINCOR.* Três Corações. 2001.

LEWIS, F. L. Optimal Control. In: LEVINE, W. S. (Ed.). *The Control Handbook.* 3. ed. Boca Raton: CRC Press, Inc., 1996. p. 763.

NEURÔNIO. In: Grande Enciclopédia Larrouse Cultural. [S.l.], v. 17, Nova Cultural, 1998. p. 4194. Venda permitida somente em conjunto com as edições dos jornais Folha de S. Paulo ou O Globo.

OGATA, K. *Engenharia de Controle Moderno.* Tradução: Bernardo Severo. 3. ed. Rio de Janeiro: Prentice-Hall do Brasil, 1998. 813 p.

PASSINO, K. M. Intelligent Control. In: LEVINE, W. S. (Ed.). *The Control Handbook.* 3. ed. Boca Raton: CRC Press, Inc., 1996. p. 994-1001.

PIRES, J.N. *Robótica:* Das Máquinas Gregas à Moderna Robótica Industrial. Coimbra: Universidade de Coimbra, 2002. Disponível em: <http://robotics.dem.uc.pt/norberto/nova/pdfs/gregosxxi.pdf>. Acesso em: 19 mai. 2010.

SMAGT, P.; SCHULTEN, K. *Control of pneumatic robot arm dynamics by a neural network.* Acimenscteerdam: 1994. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.38.7943&rep=rep1&type=pdf>. Acesso em: 19 mai. 2010.

SPONG, M. W. Motion Control of Robot Manipulators. In: LEVINE, W. S. (Ed.). *The Control Handbook.* 3. ed. Boca Raton: CRC Press, Inc., 1996. p. 1339-1351.

TATIBANA, C. Y.; KAETSU, D. Y. *Uma Introdução às Redes Neurais.* [S.l.]: [2000?]. Disponível em: <http://www.din.uem.br/ia/neurais/index.htm>. Acesso em: 19 mai. 2010.

TESLA MEMORIAL SOCIETY OF NEW YORK. *Nikola Tesla (1856-1943).* New York: c2009. Disponível em: <http://www.teslasociety.com/tribute3.htm>. Acesso em: 19 mai. 2010.

ZAMBONI, Lincoln César. *Aplicação de redes de petri à especificação formal de sistemas, via invariantes.* São Paulo, 1997. 125 f. Dissertação (Mestrado em Ciência da Computação), Universidade Presbiteriana Mackenzie, 1997.

GLOSSÁRIO

classe – (computação) define um tipo com conjunto de constantes, variáveis e métodos que fornecem funcionalidades unidas para serem facilmente utilizadas e/ou manipuladas (os métodos fornecem funcionalidades e as variáveis, valores que podem ser manipulados internamente ou externamente). Uma classe é apenas uma definição, portanto não pode ser diretamente manipulada (embora dependendo da linguagem seja permitido o uso direto de métodos ou constantes com a condição que não haja necessidade de manipular dados que devem ser armazenados no tipo que a classe define). Utiliza-se em programação orientada a objeto (OOP, object oriented programing).

constante – (computação) é uma posição de memória que armazena um valor normalmente representada por um nome legível, muitas vezes abreviado que é definido somente uma vez durante a declaração. A partir daí o valor é somente para leitura (não pode ser sobrescrito ou atualizado).

exceção – (computação) estado especial que não pode ser tratado no fluxo normal de execução de um programa. Pode ser esperada ou não. A pilha de chamada de métodos vai sendo esvaziada até que um tratador de exceção seja encontrado. O tratador pode interromper este esvaziamento ou permitir que continue depois de sua execução.

instância – (computação) uma versão de um objeto independente de outros objetos. Duas instâncias de uma classe significa que há dois objetos criados, cada um pode ser manipulado separadamente sem interferir no outro. Utiliza-se em OOP.

função – (computação) método que retorna com um valor de saída.

método – (computação) é um trecho de código que pode ser chamado de outras pastes do programa e retorna ao término da execução (o ponto de execução do programa continua da linha seguinte à que chamou o método). Pode aceitar parâmetros e dependendo da linguagem de programação, de acordo com a declaração, modificar as variáveis que são passadas. É classificado como um procedimento ou como uma função, embora, dependendo da linguagem de programação, possam haver diferentes nomenclaturas.

objeto – (computação) tipo especial criado a partir de uma definição de classe, com métodos, variáveis e constantes. Utiliza-se em OOP.

pilha – (computação) em um processador, é uma memória que armazena as posições das instruções que mudaram o ponto de execução para métodos, para onde o ponto de execução deve retornar após o término de cada método. Ex.: em um programa sendo executado, o método A é chamado pela instrução na posição 35, o valor 35 é guardado na

pilha, após o método A terminar, o valor 35 é lido da pilha e o ponto de execução volta para a posição 35, e prossegue para a 36, seguindo o curso do programa.

procedimento – (computação) método que retorna sem um valor de saída.

tipo – (computação) definição de tipo de dado que pode ser representado por variáveis.

variável – (computação) é uma posição de memória que armazena um valor normalmente representada por um nome legível, muitas vezes abreviado, o valor pode ser gravado e lido, mas estas operações devem ser feitas considerando-se o tipo definido na declaração da variável. Dependendo da linguagem de programação, as variáveis podem ser de tipos especiais que permitem conversões automáticas como por exemplo texto para números e vice-versa.

APÊNDICE A – Normas e recomendações de programação em Object Pascal

A linguagem Object Pascal tem uma estrutura simples de arquivo. Um único arquivo é usado para criar um aplicativo, uma biblioteca ou uma unidade, com as declarações e as implementações, ao contrário do que acontece no C e no C++, que as declarações são colocadas em um arquivo h (header) e as implementações em um arquivo c ou cpp (source). Uma unidade é um conjunto de zero ou mais variáveis, métodos e classes que pode fazer parte de um programa ou uma biblioteca.

O arquivo inicia-se com um identificador, no caso de uma unidade, o identificador é unit. Existem três seções: interface, implementation e initialization. Interface é o lugar onde se faz as declarações que serão acessíveis a qualquer outra unidade ou programa que utilize esta unidade. Implementation (implementação) é onde se faz a implementação de todos os códigos, incluindo os dos métodos declarados na interface. Initialization é um lugar para um código que é executado quando a unidade é carregada na memória para uso.

Tanto na seção interface como na seção implementation, podem haver três blocos, cada um independente e não obrigatório, são eles: uses, type e var. O bloco uses contém os nomes das unidades que contém os elementos utilizados pelo código implementado, assim uma unidade pode ser usada por outra. O bloco type tem definições de tipos compostos, que podem ser classes ou outros mais simples ou mesmo tipos iguais aos já existentes com outros nomes. O bloco var possui variáveis.

Declarações de métodos na interface devem ser feitas após a última definição de tipo, não há implementação de métodos na interface. Implementações de métodos na implementation (incluindo todos os métodos das classes declaradas na interface) devem ser feitas depois das definições das variáveis, não há declaração de métodos na implementation.

A seguir, exemplo de um pequeno arquivo de unidade com todos os blocos possíveis e um método, este declarado na interface:

```
unit Unit1;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;
type
  MyPublicInt = Integer;
var
  v1: MyPublicInt;

procedure doIt;
```

```
implementation

uses
  Math;
type
  MyPrivateInt = Integer;
var
  v2: MyPrivateInt;

procedure doIt;
begin
  v2 := 1;
  v1 := v2;
end;

initialization

doIt;

end.
```

Todos os blocos de código a ser executado se iniciam em begin e terminam em end, e todas as instruções terminam com “;” exceto as que antecedem a palavra else.

Ponteiro é um tipo especial de variável que aponta uma posição de memória, A memória apontada pode ter qualquer valor. Ele pode ser tipificado, o que significa que se pode acessá-lo para ler o conteúdo da posição de memória que ele aponta como sendo de um determinado tipo. Para declarar um ponteiro tipificado utiliza-se um “^” (acento circumflexo, sem letra) antes do nome do tipo, por exemplo:

```
type
MyInt = Integer;
PMyInt = ^MyInt;
```

Para acessar a posição de memória que o ponteiro aponta, usa-se o “^” depois do nome da variável do tipo ponteiro, por exemplo:

```
var
pmi: PmyInt;
begin
pmi := new(PMyInt);
pmi^ := 5;
```

O Object Pascal não tem suporte a arrays dinâmicos (que podem mudar de tamanho) mas os ponteiros podem ser usados como arrays alocando-se memória com getmem e usando-se um índice baseado em zero no lugar do “^”:

```
var
getmem(pmi, 3 * sizeOf(MyInt))
pmi[0] := 5;
pmi[1] := 5;
pmi[2] := 5;
```

Quando um ponteiro é passado como parâmetro de método, não é possível determinar a quantidade de memória alocada para o ponteiro, deste modo, as chamadas a funções que aceitam ponteiros como parâmetros devem sempre serem feitas com cautela.

Em Object Pascal, recomenda-se que todos os nomes de classe se iniciem com “T”, que todas as variáveis de classe se iniciem com “F” e que nenhuma variável de classe seja pública, utilizando propriedades (property) para acesso externo se necessário.

A partir daqui seguem recomendações que, embora focadas no Object Pascal, podem também ser consideradas em outras linguagens de programação, as instruções podem ser diferentes, mas com funcionalidades similares.

Exceções são situações anormais durante o processamento de um programa, não são necessariamente erros. Quando bem utilizadas podem ser grandes aliadas para encontrar problemas ou até mesmo para tornar programas mais seguros. Um exemplo de exceção é divisão por zero.

Toda a memória manualmente alocada deve ser obrigatoriamente liberada. É aconselhável, também, o uso de blocos try/finally para garantir a liberação da memória. O código de finally é executado incondicionalmente ao término do bloco try. Se houver uma exceção durante a execução do bloco try, o ponto de execução vai direto para o finally pulando as instruções intermediárias restantes.

```
getmem(pmi, 3 * sizeOf(MyInt))
try
  …
finally
  freemem(pmi, 3 * sizeOf(MyInt))
end;
```

Blocos try/except são usados para tratar exceções, o código de except é executado somente se houver uma exceção durante a execução do bloco try (caso em que o ponto de execução pula as instruções restantes do bloco try e vai direto para o bloco except), se não houver exceção, o ponto de execução pula o bloco except quando o atinge.

```
try
  …
except
  showmessage('Houve exceção, a execução foi interrompida');
end;
```

Todo objeto programado deve, na medida do possível, ser escrito de forma a aceitar infinitas expansões com um mínimo de alterações em código já escrito e toda a expansão deve, na medida do possível, manter a compatibilidade com os objetos anteriores.

Se o código-fonte será impresso, pode ser útil limitar o comprimento de todas as linhas. Um valor comum herdado das impressoras matriciais com formulários contínuos de 8,5” de largura é 80 caracteres.

Mais detalhes de programação em Object Pascal podem ser obtidos em livros sobre Delphi, no site <http://www.freepascal.org/> ou em outras fontes.

APÊNDICE B – Código Fonte Completo da Unidade UNN

```
{

Copyright (C) 2009 Fernando Biazi Nascimento

Criado por Fernando Biazi Nascimento
2009-09-24

Condições e Licença de uso:

1. Se as leis locais, de alguma forma qualquer, impedir ou limitar o uso desta
   licença ou qualquer termo ou condição nela presente, o usuário não é
   elegível ao uso deste código-fonte ou do programa por ele gerado e deve
   destruir todas as cópias que tiver em seu poder. O autor não é responsável e
   não pode ser responsabilizado por quaisquer taxas, valores ou prejuízos
   decorrentes desta imposição;

2. Qualquer programa ou código que se baseie neste código, em todo ou em parte,
   ou que seja distribuído junto com este código ou o programa gerado deve ser
   submetido a esta mesma licença, incluindo este item, opcionalmente com termos
   adicionais que não entrem em conflito com nenhum dos termos aqui descritos.

3. Este código fonte é fornecido "no estado em que está" sem nenhuma garantia de
   qualquer natureza, e não pode, de forma alguma, ser utilizado comercialmente,
   em equipamentos que operam em caráter primário, equipamentos hospitalares ou
   combinação destes.

4. O autor não é responsável e não pode ser responsabilizado por qualquer uso
   que se faça ou se possa fazer do código-fonte ou do programa gerado, com ou
   sem prejuízos ou danos, mesmo que ele tenha sido avisado sobre a possibilidade
   ou certeza de causá-los;

5. Se uma outra licença diferente desta for necessária, o autor pode ser
   contatado para que a solicitação desta possa ser feita, entretanto, sem a
   garantia de que será atendida.

6. Com a condição de atender todos os itens acima, qualquer pessoa pode utilizar
   o código fonte sob os termos da GNU General Public License como publicada
   pela Free Software Foundation, na versão 2 ou (sob a responsabilidade do
   usuário) qualquer versão mais recente. Caso exista qualquer conflito
   prevalecem os itens acima.

7. Cópias digitais da licença GNU GPL podem ser obtidas por meio do endereço da
   web <http://www.gnu.org/licenses/licenses.html>, incluindo a versão 2,
   anteriores e posteriores. Traduções não-oficiais encontram-se neste mesmo site
   e, embora não sejam consideradas legais pela Free Software Foundation, podem ser
   utilizadas para melhor entender todos os termos da licença.

 ******************************************************************************
 *                                                                            *
 *   This source is free software; you can redistribute it and/or modify      *
 *   it under the terms of the GNU General Public License as published by     *
 *   the Free Software Foundation; either version 2 of the License, or        *
 *   (at your option) any later version.                                      *
 *                                                                            *
 *   This code is distributed in the hope that it will be useful, but         *
 *   WITHOUT ANY WARRANTY; without even the implied warranty of               *
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU        *
 *   General Public License for more details.                                 *
 *                                                                            *
 *   A copy of the GNU General Public License is available on the World       *
 *   Wide Web at <http://www.gnu.org/copyleft/gpl.html>. You can also         *
 *   obtain it by writing to the Free Software Foundation,                    *
 *   Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.           *
 *                                                                            *
```

```
 ******************************************************************************

}
unit UNN;

{$mode objfpc}{$H+}

interface

uses
Classes, SysUtils;

type
  { Tipos personalizados - permitem alterar o tipo real de dados sem alterar o
    programa }
  Qty = Integer; { Manter Qty e Idx com o mesmo tipo }
  Idx = Integer;

  PQtyArr = ^Qty;

  ImpVal = Single;
  PImpValArr = ^ImpVal;

  Weight = Single;
  PWeightArr = ^Weight;

  TNeuron = class(TObject) { Declaração do tipo neurônio }
    private
      FDelta: Weight; { Variação a ser aplicada aos pesos }
      FEta: Weight; { Taxa de treinamento }
      FInputCount: Qty; { Quantidade de entradas }
      FOutput: ImpVal; { Valor da saída do neurônio }
      FWeights, FDeltaWeights: PWeightArr; { Conjuntos de pesos e de deltas dos
                                                                        pesos }
    protected
      function getWeight(input: Idx): Weight;
      function checkRangeWith(input: Idx; noException: Boolean = False):
                                                                    Boolean;
      procedure setAbsoluteError(newAbsoluteError: ImpVal);
      procedure setWeight(input: Idx; newWeight: Weight);
    public
      constructor Create(inputCount: Qty);
      destructor Destroy; override;
      procedure addDeltas(inputs: PImpValArr);
      procedure adjust;
      procedure process(inputs: PImpValArr);
      property absoluteError: ImpVal write setAbsoluteError;
      property delta: Weight read FDelta;
      property inputCount: Qty read FInputCount;
      property output: ImpVal read FOutput;
      property weights[input: Idx]: Weight read getWeight write setWeight;
  end;

  PTNeuronArr = ^TNeuron;

  TLayer = class(TObject) { Declaração do tipo layer }
    private
      FInputCount: Qty; { Quantidade de entradas }
      FRequestedNeuronCount, FNeuronCount: Qty; { Quantidade de neurônios }
      FNeurons: PTNeuronArr;  { Conjunto de neurônios }
    protected
      procedure freeNeurons;
      procedure setNeuronWeight(neuron, input: Idx; newWeight: Weight);
      function checkRangeWith(neuron: Idx; noException: Boolean = False):
                                                                    Boolean;
      function getNeuronDelta(neuron: Idx): Weight;
      function getNeuronOutput(neuron: Idx): ImpVal;
      function getNeuronWeight(neuron, input: Idx): Weight;
```

```
    public
      constructor Create(inputCount, neuronCount: Qty);
      destructor Destroy; override;
      procedure addDeltas(inputs: PImpValArr);
      procedure adjust;
      procedure process(inputs: PImpValArr);
      procedure readPrevErrors(absoluteErrors: PImpValArr);
      procedure setErrors(absoluteErrors: PImpValArr);
      function readOutputs(target: PImpValArr): Qty;
      property inputCount: Qty read FInputCount;
      property neuronCount: Qty read FNeuronCount;
      property neuronDelta[neuron: Idx]: Weight read getNeuronDelta;
      property neuronOutput[neuron: Idx]: ImpVal read getNeuronOutput;
      property neuronWeight[neuron, input: Idx]: Weight read getNeuronWeight
                                                 write setNeuronWeight;
  end;

  PTLayerArr = ^TLayer;

  TNN = class(TObject) { Declaração do tipo rede neural (neural network) }
    private
      FInputCount: Qty; { Quantidade de entradas }
      FRequestedLayerCount, FLayerCount: Qty; { Quantidade de layers }
      FMaxLayerSize: Qty; { Maior número de saídas entre todos os layers }
      FOutputCount: Qty; { Quantidade de saídas }
      FLayers: PTLayerArr; { Conjunto de camadas }
      FOutputs: PImpValArr; { Conjunto de saídas }
    protected
      procedure freeLayers;
      procedure setNeuronWeight(layer, neuron, input: Idx; newWeight: Weight);
      function checkRangeWith(layer: Idx; noException: Boolean = False):
                                                                  Boolean;
      function getLayer(i: Idx): TLayer;
      function getLayerInputCount(layer: Idx): Qty;
      function getNeuronCount(layer: Idx): Qty;
      function getNeuronOutput(layer, neuron: Idx): ImpVal;
      function getNeuronWeight(layer, neuron, input: Idx): Weight;
      function getOutput(i: Idx): ImpVal;
    public
      constructor Create(inputCount, layerCount: Qty; neuronCounts: PQtyArr);
      destructor Destroy; override;
      procedure addDeltas(inputs: PImpValArr; workArr: PImpValArr = nil);
      procedure adjust;
      procedure computeErrorsTo(targets: PImpValArr; workArr: PImpValArr = nil);
      procedure process(inputs: PImpValArr; workArr: PImpValArr = nil);
      function readOutputs(target: PImpValArr): Qty;
      function train(inputs, targets: PImpValArr): ImpVal;
      property inputCount: Qty read FInputCount;
      property layer[i: Idx]: TLayer read getLayer;
      property layerCount: Qty read FLayerCount;
      property layerInputCount[layer: Idx]: Qty read getLayerInputCount;
      property maxLayerSize: Qty read FMaxLayerSize;
      property neuronCount[layer: Idx]: Qty read getNeuronCount;
      property neuronOutput[layer, neuron: Idx]: ImpVal read getNeuronOutput;
      property neuronWeight[layer, neuron, input: Idx]: Weight
                                                 read getNeuronWeight
                                                 write setNeuronWeight;
      property output[i: Idx]: ImpVal read getOutput;
      property outputCount: Qty read FOutputCount;
  end;

  procedure FreememThenNil(var p:pointer; Size:ptrint);

implementation

procedure FreememThenNil(var p:pointer; Size:ptrint);
begin
  if p <> nil then begin
```

```
    Freemem(p, Size);
    p := nil;
  end;
end;

{ TNeuron }

constructor TNeuron.Create(inputCount: Qty);
var
  i: Idx;
begin
  inherited Create;
  FDelta := 0;
  FInputCount := 0;
  FOutput := 0;
  FEta := 0.35;
  FWeights := nil;
  FDeltaWeights := nil;
  Getmem(FWeights, inputCount * Sizeof(Weight));
  { Se não puder alocar memória, criar exceção }
  if FWeights = nil then
    raise EOutOfMemory.Create('Sem memória para criar pesos do neurônio');
  try
    Getmem(FDeltaWeights, inputCount * Sizeof(Weight));
    { Se não puder alocar memória, criar exceção }
    if FDeltaWeights = nil then
      raise EOutOfMemory.Create('Sem memória para criar pesos do neurônio');
    try
      { Atribuição de pesos - o peso é dividido pela quantidade de entradas para
        evitar saturação inicial no neurônio }
      for i := 0 to inputCount - 1 do begin
        FWeights[i] := Weight((Random * 2 - 1) / inputCount);
        FDeltaWeights[i] := 0;
      end;
      FInputCount := inputCount;
    except
      on E: Exception do begin
        FreememThenNil(FDeltaWeights, inputCount * sizeOf(Weight));
        raise E;
      end;
    end;
  except
    on E: Exception do begin
      FreememThenNil(FWeights, inputCount * sizeOf(Weight));
      raise E;
    end;
  end;
end;


destructor TNeuron.Destroy;
begin
  if FInputCount <> 0 then begin
    FreememThenNil(FDeltaWeights, FInputCount * sizeOf(Weight));
    FreememThenNil(FWeights, FInputCount * sizeOf(Weight));
    FInputCount := 0;
  end;
  Inherited Destroy;
end;

procedure TNeuron.addDeltas(inputs: PImpValArr);
var
  i: Idx;
begin
  for i := 0 to FInputCount - 1 do
    if (inputs[i] <> 0) then
      FDeltaWeights[i] := FDeltaWeights[i] + (FEta * FDelta * inputs[i]);
end;
```

```
procedure TNeuron.adjust;
var
  i: Idx;
begin
  for i := 0 to FInputCount - 1 do begin
    FWeights[i] := FWeights[i] + FDeltaWeights[i];
    FDeltaWeights[i] := 0;
  end;
end;

function TNeuron.checkRangeWith(input: Idx; noException: Boolean = False):
         Boolean;
begin
  if (input >= 0) and (input < FInputCount) then
    Result := True
  else begin
    Result := False;
    if not noException then
      raise ERangeError.Create('Índice [input] fora da faixa: ' +
                               IntToStr(input) + '; faixa: [0 - ' +
                               IntToStr(FInputCount - 1) + ']');
  end;
end;

function TNeuron.getWeight(input: Idx): Weight;
begin
  if checkRangeWith(input) then
    Result := FWeights[input];
end;

procedure TNeuron.process(inputs: PImpValArr);
var
  i: Idx;
  s: ImpVal;
begin
  s := 0;
  for i := 0 to FInputCount - 1 do
    s := s + (FWeights[i] * inputs[i]);
  FOutput := ((2.0 / ( 1 + exp(-s))) - 1);
end;

procedure TNeuron.setAbsoluteError(newAbsoluteError: ImpVal);
var
  nOutput: ImpVal; { Saída sem a constante de simetria }
begin
  nOutput := FOutput + 1;
  FDelta := newAbsoluteError * (nOutput - (nOutput * nOutput / 2));
end;

procedure TNeuron.setWeight(input: Idx; newWeight: Weight);
begin
  if checkRangeWith(input) then
    FWeights[input] := newWeight;
end;

{ TLayer }

constructor TLayer.Create(inputCount, neuronCount: Qty);
var
  i: Idx;
begin
  inherited Create;
  FRequestedNeuronCount := neuronCount;
  FInputCount := inputCount;
  FNeuronCount := 0;
  Getmem(FNeurons, FRequestedNeuronCount * sizeOf(TNeuron));
  { Se não puder alocar memória, criar exceção }
```

```
  if FNeurons = nil then
    raise EOutOfMemory.Create('Sem memória para criar neurônios');
  try
    for i := 0 to FRequestedNeuronCount - 1 do begin
      FNeurons[i] := TNeuron.Create(FInputCount);
      inc(FNeuronCount);
    end;
  except
    on E: Exception do begin
        if (E is EOutOfMemory) then
          E.Message := 'Neurônio ' + IntToStr(i) + ': ' + E.Message;
        freeNeurons;
        raise E;
      end;
  end;
end;

destructor TLayer.Destroy;
begin
  freeNeurons;
  inherited Destroy;
end;

procedure TLayer.addDeltas(inputs: PImpValArr);
var
  i: Idx;
begin
  for i := 0 to FNeuronCount - 1 do
    FNeurons[i].addDeltas(inputs);
end;

procedure TLayer.adjust();
var
  i: Idx;
begin
  for i := 0 to neuronCount - 1 do
    FNeurons[i].adjust;
end;

function TLayer.checkRangeWith(neuron: Idx; noException: Boolean = False):
         Boolean;
begin
  if (neuron >= 0) and (neuron < FNeuronCount) then
    Result := True
  else begin
    Result := False;
    if not noException then
      raise ERangeError.Create('Índice [neuron] fora da faixa: ' +
                               IntToStr(neuron) + '; faixa: [0 - ' +
                               IntToStr(FNeuronCount - 1) + ']');
  end;
end;

procedure TLayer.freeNeurons;
var
  i: Idx;
begin
  if FNeuronCount > 0 then begin
    for i := FNeuronCount - 1 downto 0 do
      if FNeurons[i] <> nil then
        FNeurons[i].Free;
    FreememThenNil(FNeurons, FRequestedNeuronCount * sizeOf(TNeuron));
    FNeuronCount := 0;
  end;
end;

function TLayer.getNeuronDelta(neuron: Idx): Weight;
begin
```

```
  if checkRangeWith(neuron) then
    Result := FNeurons[neuron].delta;
end;


function TLayer.getNeuronOutput(neuron: Idx): ImpVal;
begin
  if checkRangeWith(neuron) then
    Result := FNeurons[neuron].output;
end;

function TLayer.getNeuronWeight(neuron, input: Idx): Weight;
begin
  if checkRangeWith(neuron) then
    Result := FNeurons[neuron].weights[input];
end;

procedure TLayer.process(inputs: PImpValArr);
var
  i: Idx;
begin
  for i := 0 to neuronCount - 1 do
    FNeurons[i].process(inputs);
end;

function TLayer.readOutputs(target: PImpValArr): Qty;
var
  i: Idx;
begin
  for i := 0 to neuronCount - 1 do
    target[i] := FNeurons[i].output;
  Result := neuronCount;
end;

procedure TLayer.readPrevErrors(absoluteErrors: PImpValArr);
var
  i, j: Idx;
begin
  for i := 0 to FInputCount - 1 do begin
    absoluteErrors[i] := 0;
    for j := 0 to FNeuronCount - 1 do
      absoluteErrors[i] := absoluteErrors[i] + (FNeurons[j].delta *
                           FNeurons[j].weights[i]);
  end;
end;

procedure TLayer.setErrors(absoluteErrors: PImpValArr);
var
  i: Idx;
begin
  for i := 0 to FNeuronCount - 1 do
    FNeurons[i].absoluteError := absoluteErrors[i];
end;

procedure TLayer.setNeuronWeight(neuron, input: Idx; newWeight: Weight);
begin
  if checkRangeWith(neuron) then
    FNeurons[neuron].weights[input] := newWeight;
end;

{ TNN }

constructor TNN.Create(inputCount, layerCount: Qty; neuronCounts: PQtyArr);
var
  i: Idx;
  nextInputCount: Qty;
begin
  inherited Create;
```

```
  FRequestedLayerCount := layerCount;
  FInputCount := inputCount;
  FLayerCount := 0;
  FOutputCount := 0;
  nextInputCount := 0;
  FMaxLayerSize := 0;
  FLayers := nil;
  FOutputs := nil;
  Getmem(FLayers, FRequestedLayerCount * sizeOf(TLayer));
  { Se não puder alocar memória, criar exceção }
  If FLayers = nil then
    raise EOutOfMemory.Create('Sem memória para criar Layers');
  try
    nextInputCount := FInputCount;
    FMaxLayerSize := nextInputCount;
    for i := 0 to layerCount - 1 do begin
      FLayers[i] := TLayer.Create(nextInputCount, neuronCounts[i]);
      inc(FLayerCount);
      nextInputCount := neuronCounts[i];
      if (nextInputCount) > FMaxLayerSize then
        FMaxLayerSize := nextInputCount;
    end;
    Getmem(FOutputs, nextInputCount * sizeOf(ImpVal));
    If FOutputs = nil then
      raise EOutOfMemory.Create('Sem memória para criar Saídas');
    FOutputCount := nextInputCount;
    for i := 0 to FOutputCount - 1 do
      FOutputs[i] := 0; { valor inicial para todas as saídas }
  except
    on E: Exception do begin
      if (E is EOutOfMemory) then
        E.Message := 'Layer ' + IntToStr(i) + ': ' + E.Message;
      freeLayers;
      raise E;
    end;
  end;
end;

destructor TNN.Destroy;
begin
  if FOutputCount > 0 then begin
    FreememThenNil(FOutputs, FOutputCount * sizeOf(ImpVal));
    FOutputCount := 0;
  end;
  freeLayers;
  inherited Destroy;
end;

procedure TNN.addDeltas(inputs: PImpValArr; workArr: PImpValArr = nil);
var
  i: Qty;
  createWorkArr: Boolean;
begin
  createWorkArr := workArr = nil;
  if createWorkArr then begin
    Getmem(workArr, FMaxLayerSize * sizeOf(ImpVal));
    if workArr = nil then
      raise EOutOfMemory.Create(
         'Sem memória para criar array de trabalho para propagação dos deltas');
  end;
  try
    if inputs <> workArr then
      for i := 0 to FInputCount - 1 do
        workArr[i] := inputs[i];
    for i := 0 to FLayerCount - 1 do begin
      FLayers[i].addDeltas(workArr);
      FLayers[i].readOutputs(workArr);
    end;
```

```
  finally
    if createWorkArr then
      FreememThenNil(workArr, FMaxLayerSize * sizeOf(ImpVal));
  end;
end;

procedure TNN.adjust;
var
  i: Idx;
begin
  for i := 0 to FLayerCount - 1 do
    FLayers[i].adjust;
end;

function TNN.checkRangeWith(layer: Idx; noException: Boolean = False): Boolean;
begin
  if (layer >= 0) and (layer < FLayerCount) then
    Result := True
  else begin
    Result := False;
    if not noException then
      raise ERangeError.Create('Índice [layer] fora da faixa: ' +
                               IntToStr(layer) + '; faixa: [0 - ' +
                               IntToStr(FLayerCount - 1) + ']');
  end;
end;

procedure TNN.computeErrorsTo(targets: PImpValArr; workArr: PImpValArr = nil);
var
  i: Qty;
  createWorkArr: Boolean;
begin
  createWorkArr := workArr = nil;
  if createWorkArr then begin
    Getmem(workArr, FMaxLayerSize * sizeOf(ImpVal));
    if workArr = nil then
      raise EOutOfMemory.Create(
           'Sem memória para criar array de trabalho para propagação de erros');
  end;
  try
    for i := 0 to FOutputCount - 1 do
      workArr[i] := targets[i] - FOutputs[i];
    for i := FLayerCount - 1 downto 0 do begin
      FLayers[i].setErrors(workArr);
      FLayers[i].readPrevErrors(workArr);
    end;
  finally
    if createWorkArr then
      FreememThenNil(workArr, FMaxLayerSize * sizeOf(ImpVal));
  end;
end;

procedure TNN.freeLayers;
var
  i: Idx;
begin
  if FLayerCount > 0 then begin
    for i := FLayerCount - 1 downto 0 do
      FLayers[i].Free;
    FreememThenNil(FLayers, FRequestedLayerCount * sizeOf(TLayer));
    FLayerCount := 0;
  end;
end;

function TNN.getLayer(i: Idx): TLayer;
begin
  if checkRangeWith(i) then
    Result := FLayers[i];
```

```
end;

function TNN.getLayerInputCount(layer: Idx): Qty;
begin
  if checkRangeWith(layer) then
    Result := FLayers[layer].inputCount;
end;

function TNN.getNeuronCount(layer: Idx): Qty;
begin
  if checkRangeWith(layer) then
    Result := FLayers[layer].neuronCount;
end;

function TNN.getNeuronOutput(layer, neuron: Idx): ImpVal;
begin
  if checkRangeWith(layer) then
    Result := FLayers[layer].neuronOutput[neuron];
end;

function TNN.getNeuronWeight(layer, neuron, input: Idx): Weight;
begin
  if checkRangeWith(layer) then
    Result := FLayers[layer].neuronWeight[neuron, input];
end;

function TNN.getOutput(i: Idx): ImpVal;
begin
  Result := FLayers[layerCount - 1].neuronOutput[i];
end;


procedure TNN.process(inputs: PImpValArr; workArr: PImpValArr = nil);
var
  i: Idx;
  createWorkArr: Boolean;
begin
  createWorkArr := workArr = nil;
  if createWorkArr then begin
    { aloca espaço para o array de trabalho }
    Getmem(workArr, FMaxLayerSize * sizeOf(ImpVal));
    if workArr = nil then
      raise EOutOfMemory.Create(
            'Sem memória para criar Array de trabalho para o processamento');
  end;
  try
    { Se o mesmo ponteiro foi passado nas duas variáveis, não é necessário
      copiar os valores }
    if workArr <> inputs then
      for i := 0 to FInputCount - 1 do
        workArr[i] := inputs[i];
    for i := 0 to FLayerCount - 1 do begin
      FLayers[i].process(workArr);
      FLayers[i].readOutputs(workArr);
    end;
    for i := 0 to FOutputCount - 1 do
      FOutputs[i] := workArr[i];
  finally
    if createWorkArr then
      FreememThenNil(workArr, FMaxLayerSize * sizeOf(ImpVal));
  end;
end;

function TNN.readOutputs(target: PImpValArr): Qty;
var
  i: Idx;
begin
  for i := 0 to FOutputCount - 1 do
```

```
    target[i] := FOutputs[i];
  Result := FOutputCount;
end;

procedure TNN.setNeuronWeight(layer, neuron, input: Idx; newWeight: Weight);
begin
  if checkRangeWith(layer) then
    FLayers[layer].neuronWeight[neuron, input] := newWeight;
end;

{ Treina para apenas um conjunto de entradas e saídas, implementação externa
  pode ser desejável para melhor desempenho (evitando, por exemplo, consecutivas
  alocações e liberações de memória ou usando treinamento em modo batch) }
function TNN.train(inputs, targets: PImpValArr): ImpVal;
var
  i: Idx;
  workArr: PImpValArr;
  calcError: ImpVal;
begin
  workArr := nil;
  Getmem(workArr, FMaxLayerSize * sizeOf(Weight));
  if workArr = nil then
    raise EOutOfMemory.Create(
         'Sem memória para criar array de trabalho para treinamento');
  try
    process(inputs, workArr);
    computeErrorsTo(targets, workArr);
    addDeltas(inputs, workArr);
    adjust;
    { processa o mesmo conjunto de dados depois do treinamento e calcula a
      média do quadrado dos erros das saídas }
    process(inputs);
    calcError := 0;
    for i := 0 to FOutputCount - 1 do
      calcError :=  calcError + (abs((FOutputs[i] - targets[i]) /
                                     targets[i]));
    calcError := calcError / FOutputCount;
    Result := calcError;
  finally
    FreememThenNil(workArr, FMaxLayerSize * sizeOf(Weight));
  end;
end;

end.
```

ANEXO A – Licença GNU GPL versão 2 Publicada pela Free Software Foundation

GNU GENERAL PUBLIC LICENSE

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
c) Accompany it with the information you received as to the offer to distribute corresponding source

code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is

permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and an idea of what it does.
Copyright (C) yyyy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.
```

```
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type `show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type `show c'
for details.
```

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program `Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License.

Fonte: <http://www.gnu.org/licenses/old-licenses/gpl-2.0.html>.

ANEXO B – Nota de direitos autorais do site da ABB (exigida por figura utilizada)

Fonte: <http://www.abb.com.br/Copyright.aspx?text=copyright>.