LUCAS LORENSI DOS SANTOS

# Desenvolvimento de um framework integrado de redes neurais artificiais e lógica difusa

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica, da Pontifícia Universidade Católica do Rio Grande do Sul, como parte dos requisitos para a obtenção do título de Mestre em Engenharia Elétrica.

Área de concentração: Sistemas de Energia

ORIENTADOR: Luís Fernando Alves Pereira

Porto Alegre
(2008)

# Agradecimentos

Aos professores, orientadores e amigos Luís Fernando Alves Pereira, Alessandro Manzoni e Flávio Antônio Becon Lemos pela orientação, dedicação, confiança e apoio demonstrados ao longo do desenvolvimento deste trabalho.

A todos os professores do Curso de Pós-Graduação em Engenharia Elétrica da PUCRS que, de uma ou de outra forma, contribuíram para a realização deste trabalho.

A todos os funcionários, bolsistas, colegas e amigos do mestrado pelas horas de convívio dentro e fora da universidade.

A Companhia Estadual de Energia Elétrica S.A. - CEEE e a Centrais Elétricas de Santa Catarina S.A. - Celesc pelo suporte financeiro ao desenvolvimento desta dissertação, através de seus respectivos projetos de P&D ANEEL, bem como as outras empresas que suportaram a minha formação de pesquisador ao longo dos anos no Grupo de Sistemas de Energia Elétrica - GSEE.

# Resumo

Esta dissertação apresenta a descrição do processo de desenvolvimento de um sistema integrado de redes neurais artificiais e lógica *fuzzy*, onde o objetivo criar um ambiente, de propósito geral, para a criação de soluções que possam englobar ambas as técnicas, além de agregar estas ferramentas ao *Framework para Análise de Sistemas de Energia Elétrica* (**FASEE**). Para tanto, foram criados dois *frameworks* distintos, um para cada técnica. O *framework* de redes neurais artificiais foi desenvolvido em **C++** utilizando como base o **FASEE**, devido a este apresentar um mecanismo de derivadas parciais automáticas que facilita o processo de desenvolvimento de métodos de treinamento que usam este recurso, como, por exemplo, o *back-propagation*. Porém, devido a uma restrição do **FASEE**, o *framework* de lógica *fuzzy* foi desenvolvido utilizando a linguagem **Lua**. Esta foi escolhida pela sua capacidade de interagir com programas desenvolvidos em **C++**, fazendo com que a união das duas técnicas de inteligência artificial se dê pela união das duas tecnologias, obtendo-se assim um ambiente para a elaboração de sistemas neuro-*fuzzy*.

# Abstract

This work presents the description of the development of a integrated system of artificial neural network and fuzzy logic, where the objective is to create a general purpose environment for the creation of solutions that could combine both techniques, and furthermore aggregate that functionalities to the *Framework para Análise de Sistemas de Energia Elétrica* (**FASEE**). For that it was created two distinct frameworks, one for each technique. The artificial neural network was developed in **C++** using the **FASEE** as it background, since that have an automated mechanism for the calculus of partial deviation that facilitate the development process of training methods that use this kind of resource, for instance, the back-propagation. However, because a restriction on the **FASEE**, the fuzzy logic framework was developed in **Lua**. This language was chose given it integration capabilities in others languages, such as **C++**, making the union of the two artificial intelligence techniques by the union of both languages, delivering so an environment for neuro-fuzzy systems.

# Sumário

# Lista de Figuras

# Capítulo 1

# Introdução

A inteligência artificial (IA) é uma área de pesquisa da ciência da computação dedicada a buscar métodos ou dispositivos computacionais que possuam ou simulem a capacidade humana de resolver problemas.

Recentemente a inteligência artificial ganhou meios e massa crítica para se estabelecer como ciência integral, com problemáticas e metodologias próprias. Basicamente duas técnicas de IA serão o foco deste trabalho: redes neurais artificiais (RNA) e lógica difusa, também conhecida por lógica *fuzzy*.

O objetivo principal deste trabalho é criar um *framework* para cada uma das técnicas. Um de RNAs e outro de lógica *fuzzy*, onde possam também interagir para permitir a criações de sistemas híbridos neuro-*fuzzy*.

Antes é importante conceituar o termo *framework*. Os *frameworks* são aplicações reusáveis e semi-completas que podem ser especializadas para produzir aplicações personalizadas. No desenvolvimento de *software*, um *framework* é uma estrutura de suporte definida em que um outro projeto de *software* pode ser organizado e desenvolvido. Um *framework* pode incluir programas de suporte, bibliotecas de código, linguagens de *script* e outros *softwares* para auxiliar no desenvolvimento e unir diferentes componentes de um projeto de *softwares*. Dentro do contexto deste trabalho o termo *framework* irá se referenciar a um conjunto de classes com objetivo de reutilização de *softwares*, provendo um guia para uma solução de arquiteturas em um domínio específico de problema, o que no caso deste trabalho são RNAs e lógica *fuzzy*.

Os *frameworks* são projetados com a intenção de facilitar o desenvolvimento de *software*, habilitando analistas e programadores a gastarem mais tempo determinando as exigências de um *software* do que com detalhes de baixo nível do sistema.

Nas próximas duas seções será realizado uma breve introdução sobre o assunto de redes neurais artificiais e lógica *fuzzy*, além de uma descrição dos requisitos encontrados a partir do levantamento de sistemas existentes hoje na literatura. Os embasamentos teóricos para o desenvolvimento tanto de redes neurais artificiais quanto para lógica *fuzzy* encontram-se respectivamente na primeira seção de seus capítulo. Por fim é feita uma descrição dos objetivos

a serem alcançados com este trabalho, além de como este está estruturado.

## 1.1 Redes Neurais Artificiais

As redes neurais artificiais são um método para solucionar problemas através da simulação do funcionamento do cérebro humano, incluindo seu comportamento, ou seja, aprendendo, errando e fazendo descobertas. São técnicas computacionais que apresentam um modelo inspirado na estrutura neural de organismos inteligentes e que adquirem conhecimento através da experiência [28].

Cada rede possui nós ou unidades de processamento. Cada unidade possui interconexões para outras unidades, no qual recebem e enviam sinais. Cada unidade pode possuir memória local. Estas unidades são a simulação dos neurônios, recebendo e retransmitindo informações.

Hoje em dia, concorda-se que as redes neurais são muito diferentes do cérebro em termos de estrutura. No entanto, como o cérebro, uma rede neural é uma coleção massivamente paralela de unidades de processamento pequenas e simples, onde as interligações formam a maior parte da inteligência da rede. Entretanto, em termos de escala, o cérebro é muito maior que qualquer rede neural. Além disso, as unidades usadas na rede neural são tipicamente muito mais simples que os neurônios e o processo de aprendizado do cérebro é, certamente, muito diferente do das redes neurais artificiais.

Em aplicações reais, as redes neurais são adequadas para diversas tarefas, como:

- Aproximação de funções;
- Previsão de séries temporais;
- Classificações;
- Reconhecimento de Padrões.

Embora este trabalho não tenha como objetivo a comparação entres sistemas semelhantes encontrados atualmente, foi necessário realizar uma revisão bibliográfica a fim de levantar requisitos e aspectos relevantes para o desenvolvimento deste trabalho.

Em [13], foi desenvolvido um *framework* em **Java** onde são oferecidas condições de realizar inúmeras aplicações portáveis usando uma API(*Application Program Interface*) bastante simplificada, onde o usuário poderá adaptar um conjunto de classes para as suas necessidades ou criar outros modelos. O trabalho [18], também desenvolvido em **Java**, também é um *framework* de RNAs usado para criar, treinar e testar redes. Composto por um motor central, suas aplicações podem ser criadas em máquinas locais e treinadas em um ambiente tanto local quanto distribuído, ou em qualquer dispositivo que possua uma máquina virtual **Java**. Sua característica mais relevante é que sua arquitetura é dividida em módulos, onde a idéia é criar uma base de desenvolvimento para promover novas aplicações usando o motor central.

Outros propõem a criação de bibliotecas de funções para facilitar o uso de RNAs, buscando alternativas de criar sistemas simples e flexíveis, como [21] e [5]. Em [20] a idéia foi tanto criar uma API quanto um *framework* orientado a objetos, usando como base o *toolbox* de redes neurais artificiais do Matlab [5]. O sistema apresentado em [32] é desenvolvido desde 1989 onde o objetivo principal é criar um ambiente de simulação eficiente e flexível para pesquisas e aplicações de redes neurais artificiais.

Assim, analisando-se as soluções semelhantes existentes, este trabalho buscou atingir os seguintes requisitos para o desenvolvimento do *framework* de redes neurais artificiais:

- Fornecer uma estrutura base de desenvolvimento de virtualmente qualquer RNA;
- Fornecer a implementação de modelos clássicos de redes como: *Perceptron* Multi-Camadas, Mapas Auto-Organizados, e Redes de Base Radial;
- Permitir a adaptação destes modelos;
- Permitir criar funções de ativação definidas pelo usuário;
- Permitir a definição de diferentes arquiteturas e topologias de neurônios.

## 1.2 Lógica *Fuzzy*

A lógica difusa ou lógica *fuzzy* é uma generalização da lógica booleana que admite valores lógicos intermediários entre a "verdadeiro" e "falso", como o "talvez". Como existem várias formas de se implementar um modelo *fuzzy*, a lógica *fuzzy* deve ser vista mais como uma área de pesquisa sobre tratamento da incerteza, ou uma família de modelos matemáticos dedicados ao tratamento da incerteza, do que uma lógica propriamente dita. Esta metodologia geralmente está associada ao uso da teoria de conjuntos *fuzzy*, onde os estados indeterminados possam ser tratados também, como por exemplo, avaliar conceitos como morno, médio, etc.

Novamente, deve-se ressaltar que este trabalho não tem como objetivo buscar vantagens e desvantagens dos trabalhos encontrados na literatura, porém um revisão do que existe hoje foi necessário para o levantamento de requisitos.

Em [14] também foi criado um *framework* de lógica *fuzzy* em **Java**, de código aberto, no qual implementa o modelo Mamdani [4], e utiliza o paradigma de programação orientada à objetos, onde permite à reutilização de código, levando a uma facilidade na construção de novas aplicações usando o *framework*. Em [23] foi desenvolvida uma biblioteca de código aberto otimizada para operações críticas. Sua principal característica é estar em conformidade com a norma IEC 61131-7, padrão que define uma linguagem comum de controladores *fuzzy*. Em [7] foi criado uma biblioteca de lógica *fuzzy* usando como base o Simulink, ambiente de simulação de sistemas dinâmicos em diagrama de blocos. Em [2] é desenvolvido uma biblioteca em **C++** para realizar cálculos de incertezas e prioridades. Regras são avaliadas

com diferentes pesos, e seu desenvolvimento se deu de forma a facilitar sua adaptação em diversas aplicações.

Assim, para a criação do *framework* de lógica *fuzzy*, as seguintes funcionalidades foram o foco do desenvolvimento:

- Fornecer a implementação do modelo Mamdani e Sugeno;
- Possuir as principais funções de pertinência implementadas (Gaussiana, Triangular, Trapezoidal, etc.);
- Permitir que funções de pertinência definidas pelo usuário possam ser utilizadas;
- Possuir os principais métodos de *defuzzificação* (Centro de Gravidade, Ponto Central da Área, Média dos Máximos, etc.);
- Permitir que métodos de *defuzzificação* definidos pelo usuário possam ser utilizados;
- Possuir os principais operadores *fuzzy* ( Tmin, Tpro, Smax, Ssum, etc. );
- Permitir o uso de operadores *fuzzy* definidos pelo usuário;
- Implementar o operador de complemento(NOT);
- Definir uma gramática para definições de domínios que inclui declarações de variáveis de entrada, intermediárias e de saída com seus respectivos termos nebulosos e definição das regras;
- Permitir a definição de pesos para regras.

## 1.3 Objetivos e Contribuições da Dissertação

O objetivo principal deste trabalho é a criação de dois *frameworks*: um de redes neurais artificiais e outro de lógica *fuzzy*. Estes deverão permitir um certo nível de interação de forma a proporcionar meios de criação de um sistema neuro-*fuzzy*.

Ganhos esperados com este trabalhos são:

- Estudar redes neurais artificiais
  - Tipos
  - Algoritmos de aprendizagem;
  - Campo de aplicações;
  - Vantagens de desvantagens do seu uso.
- Estudar lógica *fuzzy*:

- Motores de inferência(Mandani e Sugeno);
- Operadores *fuzzy*(normas T e S);
- Funções de pertinência;
- Métodos de *defuzzificação*;
- Campo de aplicações;
- Vantagens de desvantagens do seu uso.

Para tanto, será utilizado como base de desenvolvimento o *Framework de Análise de Sistemas de Energia Elétrica* (**FASEE**) desenvolvido em [17], pelo fato deste apresentar uma crescente demanda de ferramentas de inteligência artificial, tendo em vista o atual aumento do uso destas técnicas para solução de problemas relacionados a sistemas de energia.

## 1.4 Estrutura da Dissertação

Este trabalho encontra-se dividido em seis Capítulos. No Capítulo 1 é apresentada a introdução sobre o assunto de inteligência artificial, conceituando-se redes neurais artificiais e lógica *fuzzy*.

No Capítulo 2 são apresentados os conceitos e fundamentos que servem de alicerce para o desenvolvimento deste trabalho, que são: programação orientada a objetos e o *framework* **FASEE** e a linguagem **Lua**.

No Capítulo 3 é inicialmente realizada uma descrição sobre tipos de redes neurais, suas topologias e seus métodos de treinamentos, seguidos da definição da estrutura base criada, o *framework* propriamente dito, base do desenvolvimento de redes *Multi-Layer Perceptron* e *Self-Organizing Map*. As seções seguintes deste capítulo tratam de forma mais detalhada a criação de cada rede, mostrando exemplos de uso.

O Capítulo 4 é dedicado a lógica *fuzzy*, onde é feita a descrição do sistema que foi criado. Exemplos do seu uso também são apresentados, salientado os pontos flexíveis da estrutura.

No Capítulo 5 é mostrado o resultado deste trabalho através do uso tanto do *framework* de RNAs quanto de lógica *fuzzy* em aplicações com um apelo prático, tentando salientar a facilidade de uso do sistema.

Finalmente, no Capítulo 6, são apresentadas as conclusões e trabalhos futuros. Na parte final da dissertação são apresentadas as referências bibliográficas citadas ao longo desta dissertação.

# Capítulo 2

# Estrutura Básica

Neste capítulo será realizado uma descrição da estrutura básica utilizada para o desenvolvimento do *framework* de redes neurais artificiais e lógica difusa. Inicialmente será abordado os aspectos gerais e a motivação do uso de programação orientada a objetos. Em seguida, será feita uma descrição do ambiente sobre o qual esse trabalho foi desenvolvido, seguido de uma descrição do funcionamento do mecanismo de derivadas parciais automáticas utilizado.

## 2.1 Programação Orientada a Objetos

Atualmente, a informática vêm oferecendo soluções que buscam facilitar a tarefa de programação no desenvolvimento de sistemas complexos. O novo paradigma da programação orientada a objetos rompe com antigos conceitos e apresenta uma forma inteiramente nova de abordar a tarefa de programação.

A orientação a objetos, também conhecida como Programação Orientada a Objetos (POO) ou ainda em inglês *Object-Oriented Programming*(OOP) é um paradigma de análise, projeto e programação de sistemas baseado na composição e interação entre diversas unidades chamadas de objetos. O enfoque da programação, antes centrado fundamentalmente nas funcionalidades de um programa, agora passa a priorizar os elementos conceituais do domínio do problema, os objetos.

A análise e projeto orientados a objetos têm como meta identificar o melhor conjunto de objetos para descrever um determinado sistema. O funcionamento deste sistema se dá através do relacionamento e troca de mensagens entre estes objetos.

Na programação orientada a objetos, implementa-se um conjunto de classes que definem os objetos presentes no sistema. Cada classe determina o comportamento (definidos nos métodos) e estados possíveis (atributos) de seus objetos, assim como o relacionamento com outros objetos.

Uma das maiores mudanças que se pode observar no processo de transição entre programação estruturada e programação orientada a objetos é o fato de que nessa primeira os

dados são elementos passivos dentro de um programa, sendo eles processados e transformados por funções definidas. Já em POO os dados(objetos) são partes ativa dentro do contexto de um programa, onde apresentam estados e comportamentos.

O presente trabalho usa a UML (*Unified Modeling Language*) como linguagem padrão de modelagem de objetos, e sua notação gráfica para descrever os modelos orientados a objetos. A compreensão dos conceitos básicos da modelagem orientada a objetos(MOO) e da notação gráfica utilizada é recomendável para o entendimento dos próximos capítulos.

## 2.2 Linguagem Lua

Ao longo deste trabalho foi muito utilizada a linguagem **Lua**. Por esta não ter o mesmo reconhecimento de linguagens como **C++** ou **Java**, será feita uma breve descrição sobre o que é e suas funcionalidades.

A idéia principal de **Lua** é ser uma linguagem de programação poderosa, rápida e leve, projetada para estender aplicações. Ela conta com uma sintaxe simples para programação procedural com poderosas construções para descrição de dados baseadas em tabelas associativas e semântica extensível. É uma linguagem tipada dinamicamente, interpretada a partir de *bytecodes* para uma máquina virtual baseada em registradores, e tem gerenciamento automático de memória com coleta de lixo incremental.

Usada em muitas aplicações industriais, com ênfase em sistemas embutidos, a exemplo do consagrado *Adobe's Photoshop Lightroom*. Ela ainda conta com uma vasta bibliografia, sendo as principais usadas nesta trabalho [10] [11]. Várias versões de **Lua** foram lançadas e usadas em aplicações reais desde a sua criação em 1993.

Distribuída via um pequeno pacote **Lua** compila sem modificações em todas as plataformas que têm um compilador ANSI/ISO C, rodando em todos os tipos de Unix e Windows, e também em dispositivos móveis (como computadores de mão e celulares que usam BREW, Symbian, Pocket PC, etc.) e em microprocessadores embutidos (como ARM e Rabbit) para aplicações como Lego MindStorms.

**Lua** conta com uma API simples que permite uma integração com códigos escritos em outras linguagens. É simples estender **Lua** com bibliotecas escritas em outras linguagens. Também é simples estender programas escritos em outras linguagens com **Lua**. Usada para estender programas escritos não só em **C/C++**, mas também em Java, C#, Smalltalk, Fortran, Ada, e mesmo outras linguagens de *script*, como Perl and Ruby.

Um conceito fundamental no projeto de **Lua** é fornecer meta-mecanismos para a implementação de construções, em vez de fornecer uma extensa base de diretivas de construções diretamente na linguagem. Por exemplo, embora **Lua** não seja uma linguagem puramente orientada a objetos, ela fornece meta-mecanismos para a implementação de classes e herança. Os meta-mecanismos de **Lua** trazem uma economia de conceitos e mantém a linguagem

pequena, ao mesmo tempo que permitem que a semântica seja estendida de maneiras não convencionais.

Desenvolvida em código aberto e distribuída sob uma licença MIT, ela pode ser usada para quaisquer propósitos, incluindo propósitos comerciais, sem qualquer custo.

É atualmente a única linguagem de programação de impacto desenvolvida fora do primeiro mundo, estando atualmente entre as 20 linguagens mais populares na Internet, segundo o índice TIOBE [22].

Inteiramente projetada, implementada e desenvolvida no Brasil, por uma equipe na PUC-Rio (Pontifícia Universidade Católica do Rio de Janeiro), nasceu e cresceu no Tecgraf, o Grupo de Tecnologia em Computação Gráfica da PUC-Rio. Atualmente, **Lua** é desenvolvida no laboratório Lablua. Tanto o Tecgraf quanto Lablua são laboratórios do Departamento de Informática da PUC-Rio.

## 2.3 *Framework* para Análise de Sistemas de Energia Elétrica

A plataforma computacional chamada *Framework para Análise de Sistemas de Energia Elétrica* (**C++**), apresentado em [17], consiste de um ambiente de desenvolvimento orientado a objetos para simulação e análise de sistemas de energia elétrica, compostas de diversas ferramentas matemáticas as quais suportam a estrutura de forma a deixa-la flexível, escalonável e robusta. Este ambiente define um conjunto de classes e objetos cooperantes que implementam funcionalidades comuns à diversos tipos de aplicativos na área de sistemas de energia elétrica, definindo assim, uma estrutura computacional geral que pode ser utilizada como base para a construção de um amplo conjunto de ferramentas na área de sistema elétricos de potência.

Toda a arquitetura padrão do projeto (sua estrutura geral, divisão em classes e como estas colaboram entre si) é pré-determinada pelas classes e objetos base do ambiente **FASEE**. Desta forma, o engenheiro pode concentrar-se nos aspectos específicos da sua aplicação, o que conduz a construção mais rápida e eficiente dos aplicativos. Dentre as principais características do ambiente **FASEE** destacam-se:

- Gerenciamento da descrição topológica da rede elétrica. A descrição topológica define o arranjo estrutural de um sistema (áreas, subestações, etc), seus dispositivos e equipamentos componentes (geradores, cargas, lts, etc) bem como os relacionamentos e conexões entre estes;

- Gerenciamento da estrutura resultante do processo de configuração da rede elétrica. Nesta descrição os dispositivos lógicos (seccionadoras, disjuntores, etc.) não são representados. O sistema é reduzido a nós elétricos, chamados de Barras Elétricas, e dispositivos efetivamente conectados a estes nós, determinando a configuração operativa atual do sistema;

- Utilização de uma estrutura computacional única para todos os aplicativos finais, facilitando o desenvolvimento de ambientes integrados constituídos de diversas ferramentas computacionais em um mesmo ambiente;

- Maior facilidade para o desenvolvimento, atualização e expansão dos aplicativos, permitindo agilidade na inclusão de novos modelos, equipamentos e metodologias de simulação e análise;

- Equipamentos e modelos definidos pelo usuário, independente de sua complexidade ou topologia. Esta característica permite grande flexibilidade para o desenvolvimento e inclusão de novos modelos ao sistema;

- Novas ferramentas adicionadas ao ambiente passam a ser disponíveis para todos os aplicativos desenvolvidos no ambiente. Uma ferramenta para gerenciamento de Algoritmos Genéticos, por exemplo, incorporada ao ambiente é automaticamente disponibilizada para todos os aplicativos desenvolvidos no ambiente;

- Interface gráfica única. Todos os aplicativos utilizam a mesma interface gráfica para gerenciamento da rede elétrica e visualização de resultados. Isto reduz o tempo de desenvolvimento de um aplicativo, uma vez que sua interface gráfica já está previamente implementada, e o tempo de treinamento de usuários para novas ferramentas desenvolvidas.

As funcionalidades gerais e a modularidade do ambiente **FASEE** permite que aplicativos finais sejam facilmente implementados, cabendo aos desenvolvedores simplesmente a tarefa de customização do ambiente **FASEE** para os aspectos específicos das suas aplicações, e a adição das características particulares da aplicação determinadas pelos usuários finais.

Atualmente a plataforma computacional **FASEE** conta com um amplo conjunto de aplicativos já implementados, tais como:

- Configurador de Redes Elétricas;

- Fluxo de Potência (método de Newton, formulado em coordenadas polares ou retangulares, e método desacoplado rápido);

- Coeficientes de Sensibilidade;

- Fluxo de Potência Generalizado (Full-Newton, formulado em coordenadas polares ou retangulares, e injeção de potência ou corrente na rede elétrica);

- Análise Modal (cálculo de autovalores e autovetores do sistema dinâmico, fatores de participação e *mode-shapes*);

- Simulação Completa para Análise da Estabilidade Transitória (formulado pelos métodos alternado implícito ou simultâneo implícito);
- Simulação Rápida para Estudos de Estabilidade de Longo Prazo (método quase-estático com representação da dinâmica completa do sistema);
- Fluxo de Potência para Redes de Distribuição (método de soma de potências);
- Reconfiguração Ótima para Redes de Distribuição (algoritmos genéticos)
- Alocação de Bancos de Capacitores em Redes de Distribuição;
- Recomposição de Redes de Distribuição sob Distúrbio.

O diagrama de classes simplificado da estrutura do **FASEE** é apresentada na Figura 2.3. Nele, pode-se reparar que os aplicativos elétricos(fluxo de potência, simulação dinâmica,etc...) estão separados das classes de descrição da rede, fazendo com que qualquer ferramenta de cálculo nova possa ser implementada separadamente, sem precisar modificar as estruturas que descrevem a rede elétrica.

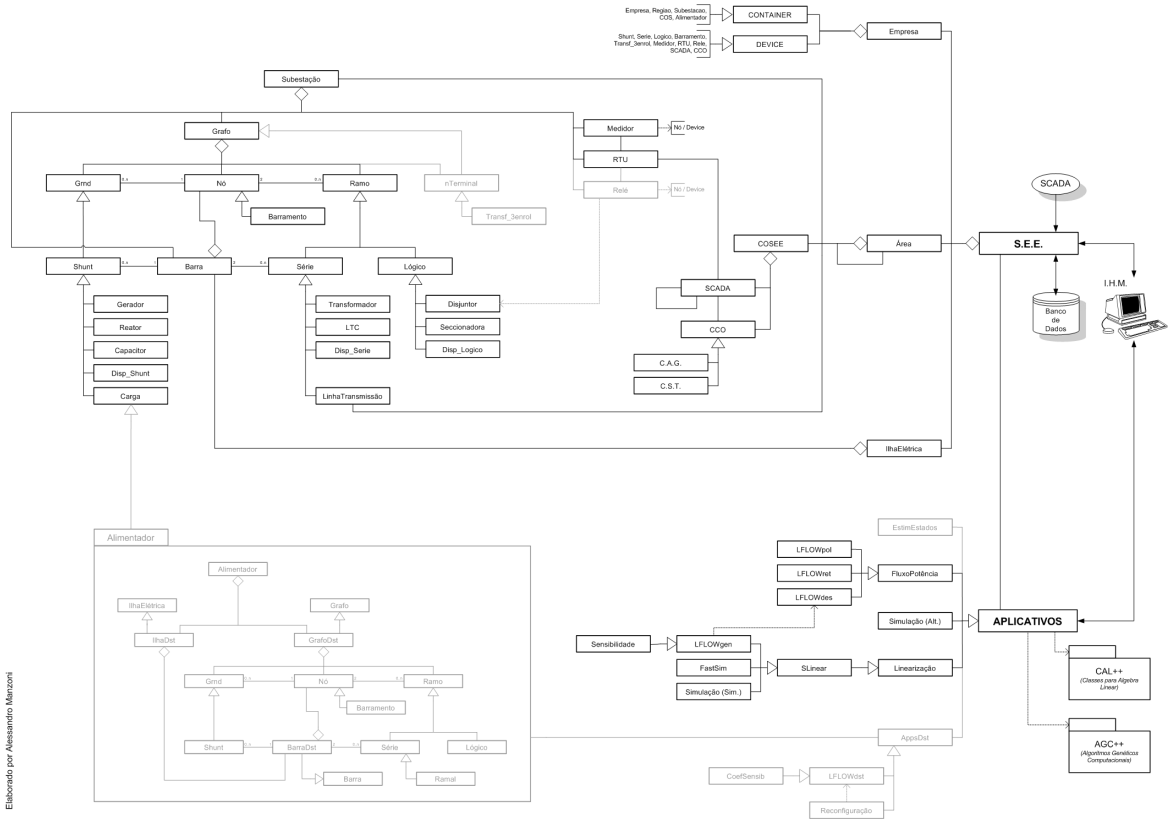


Figura 2.1: Diagrama de Classes de FASEE [17].

A seguir é feito uma descrição sobre o ambiente matemático, aqui chamada de *framework* de modelos ou simplesmente modelos, disponível no **FASEE** para a elaboração de equações e cálculo de derivadas parciais de primeira ordem.

## 2.4 *Framework* de Modelos [17]

No **FASEE** as funcionalidades dos dispositivos são altamente dependentes do aplicativo em uso. De fato, o comportamento e os dados de um dispositivo podem mudar completamente de um aplicativo para outro. Um gerador, por exemplo, possui dados e comportamento para o Fluxo de Potência completamente diferente dos dados e comportamento que possui para a Análise da Estabilidade Transitória ou Cálculo de Curto-Circuito, no entanto conceitualmente o dispositivo Gerador ainda é o mesmo para todas as aplicações.

A metodologia orientada a objetos (MOO) implementa mecanismos que permitem que dados e funcionalidades específicas sejam adicionadas aos dispositivos conforme a aplicação e removidos quando não mais necessários. A estratégia adotada para alcançar este objetivo define cada dispositivo como sendo uma composição de duas estruturas especializadas: um estado, que determina sua condição de operação e está rigidamente acoplado ao dispositivo, e um modelo, que atualiza o estado deste dispositivo e pode ser alterado conforme as necessidades do aplicativo em uso. A classe genérica *DEVICE* contém então apenas funções de caráter geral, todos os dados e funcionalidades específicas são deslocados para o modelo. De fato, deve-se evitar que características específicas de uma ou outra aplicação sejam adicionadas a esta classe. A Figura 2.2 e Figura 2.3 mostram, respectivamente, a estrutura geral de um dispositivo genérico e o diagrama de classes que o representa.

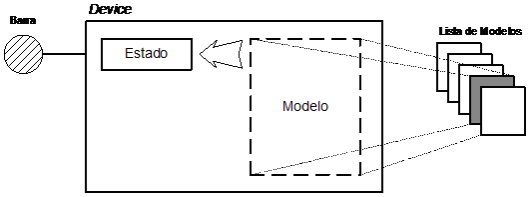


Figura 2.2: Estrutura geral de um dispositivo.

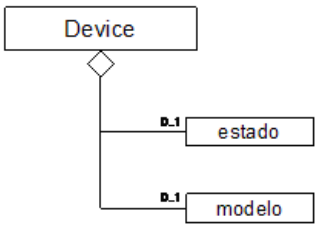


Figura 2.3: Diagrama de classes de um dispositivo.

A ilustração da Figura 2.2 mostra que o modelo de um dispositivo pode ser livremente substituído, alterando assim o comportamento deste dispositivo de acordo com as necessidades dos aplicativos (dados e funcionalidades específicas). Entretanto, o dispositivo agregador (descendente da classe *DEVICE*) permanece fixo entre as aplicações, concordando com a premissa de que o elemento em si não muda entre aplicativos, mas apenas o seu comportamento, ou seja, seu modelo.

A decomposição de cada dispositivo em estado e modelo evita que sejam necessárias

alterações nos dispositivos para cada nova aplicação implementada, sendo simplesmente adicionado um novo modelo ao dispositivo com os dados e funcionalidades específicas para o aplicativo. De maneira geral, é possível dizer que, com a abordagem proposta, apenas os dados e as funcionalidades dedicadas do dispositivo são alterados conforme muda a aplicação, sendo que o elemento conceitual (classe agregadora) continua o mesmo. Além disso, em uma representação integrada (dispositivos representados através de uma única classe com dados e funcionalidades) os dispositivos tenderiam a se tornar classes com um amontoado de métodos e atributos, resultantes da implementação de vários aplicativos sobre a mesma estrutura.

Um modelo contém as características específicas de um determinado dispositivo, sempre voltadas para um aplicativo ou conjunto de aplicativos. O modelo engloba, então, os dados, equações e ações individuais dos dispositivos para um determinado aplicativo. No entanto, é possível identificar um conjunto de funcionalidades para os dispositivos (ou para os modelos) que são comuns a uma grande diversidade de aplicativos. As funcionalidades são:

- Armazenamento de Dados e Equações: os dados, parâmetros, variáveis e equações que definem um modelo devem ser armazenados e gerenciados no interior do modelo;
- Determinação das Condições Iniciais: valendo-se do estado do dispositivo e do conjunto de equações que definem o modelo, este deve ser capaz de inicializar suas variáveis e parâmetros internos de forma a reproduzir o mesmo estado anterior, porém agora para outros dados e com outras funcionalidades;
- Solução e Derivação das Equações: esta talvez seja a funcionalidade mais importante de um modelo, uma vez que é responsável por ações como resolver o conjunto de equações do modelo, atualizar o estado do dispositivo e calcular derivadas parciais das equações que definem o modelo em relação ao conjunto de variáveis de estado;
- Definição de Funcionalidades Específicas: aplicativos que requerem ações particulares de um determinado dispositivo podem ainda introduzir tais ações em classes derivadas da classe modelo (através do mecanismo da herança).

A identificação do conjunto de funcionalidades citadas acima permite definir uma interface de utilização comum para estas funcionalidades, de tal forma que todos os aplicativos reconheçam e saibam como tratar estas funções, independente do modelo específico que está associado ao dispositivo. Isto permite que um dispositivo conserve o mesmo modelo para vários aplicativos, desde que o modelo seja adequado ao aplicativo (o modelo $\pi$ para uma linha de transmissão, por exemplo, pode manter-se o mesmo para uma grande quantidade de aplicativos). Além disso, a padronização da interface de acesso às funcionalidades comuns generaliza os aplicativos que utilizam esta interface, uma vez que tal aplicativo sabe como tratar o modelo sem conhecer necessariamente sua estrutura interna. Por esta razão, os aplicativos passam a tratar novos modelos automaticamente, sem a necessidade de alterações no código do

aplicativo. Assim, um programa de fluxo de potência e um programa de simulação dinâmica completa, por exemplo, incorporam automaticamente qualquer novo modelo adicionado a um sistema de energia elétrica(SEE) sem qualquer alteração no código do programa.

Para alcançar o grau de generalidade descrito acima, o modelo deve armazenar e gerenciar eficientemente o conjunto de parâmetros, variáveis e equações que definem o comportamento do dispositivo. Uma estrutura computacional especialmente projetada para este fim foi implementada em [17] e será descrita a seguir.

### 2.4.1 Uma Equação em Diagrama de Blocos

Uma equação matemática qualquer pode ser representada através de um conjunto de blocos elementares organizados na forma de um diagrama de blocos, onde cada bloco elementar representa uma operação matemática singular. A Figura 2.4.1 mostra a representação em diagrama de blocos da equação 2.1.

$$f(x, y) = sen(3.x + y^2) \tag{2.1}$$

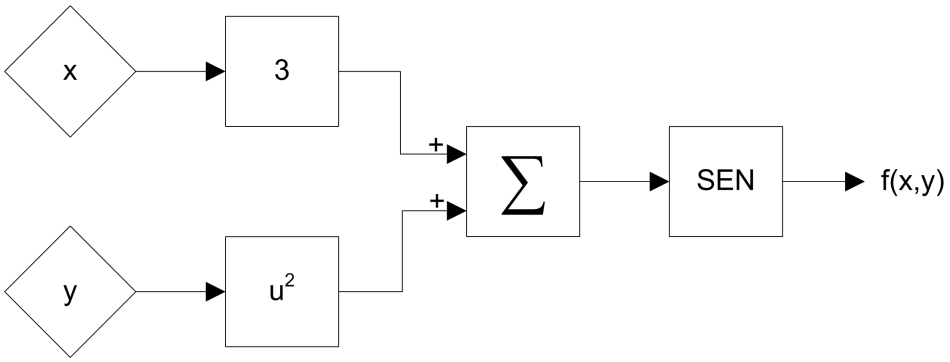


Figura 2.4: Diagrama de blocos que representa a Equação 2.1.

A decomposição de uma equação em seu correspondente diagrama de blocos permite definir uma estrutura orientada a objetos especial onde blocos construtivos elementares (objetos no contexto da modelagem orientada a objetos) são interligados para formar o conjunto de equações que definem o modelo de um dispositivo. Esta forma de representar as equações do modelo adiciona flexibilidade à estrutura computacional, uma vez que permite ao modelo conhecer e manipular a estrutura da equação armazenada. Além disso, os parâmetros e equações do modelo podem ser construídos em tempo de execução do programa, permitindo aos usuários definirem seus próprios dispositivos (geradores, linhas de transmissão, CAG, etc.) em um conceito denominado **Dispositivo Definido pelo Usuário**.

O elemento central da estrutura orientada a objetos implementada para descrever equações é um bloco genérico (classe *BLC*) sem operação matemática específica associada, e que será utilizado como base para a construção de todos os demais blocos elementares (através do

mecanismo da herança). Abstraindo-se o tipo específico de operação matemática associada ao bloco, pode-se definir um conjunto de funcionalidades que serão comuns a todos os blocos elementares que definem um modelo. Destas a mais fundamental trata da conectividade entre os blocos, uma vez que esta funcionalidade dita a estrutura da equação armazenada. Para este fim, o bloco base gerencia um conjunto de classes especiais auxiliares (denominadas variáveis) responsáveis pela interconectividade dos blocos. Cada bloco pode possuir "n" variáveis de entrada, denominadas *VARINP* ($u_1 ... u_n$), e "m" variáveis de saída, denominadas *VAROUT* ($y_1 ... y_m$), sendo o número de entradas e saídas de cada bloco dependente da operação matemática que este executa (1:1 para um bloco ganho, n:1 para um bloco somador, etc). A variável de saída (*VAROUT*) armazena o valor numérico resultante da operação matemática elementar, podendo conectar-se a uma ou mais variáveis de entrada de outros blocos. As variáveis de entrada (*VARINP*), por sua vez, podem conectar-se a somente uma variável de saída de outro bloco, evitando assim ambigüidades (1 entrada associada a diversas saídas). A Figura 2.4.1 mostra a estrutura geral do bloco base e suas variáveis de entrada/saída, bem como as conexões possíveis entre as variáveis.

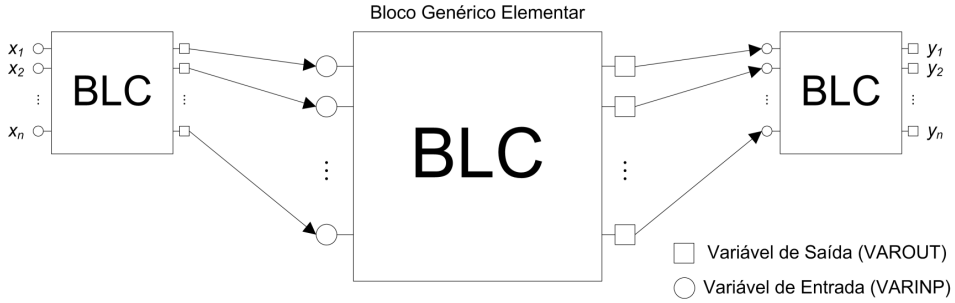


Figura 2.5: Estrutura de conexão do bloco elementar *BLC*.

Cada bloco derivado do bloco base implementa uma operação matemática específica (operação virtual no bloco base) que relaciona uma saída a uma ou mais entradas do bloco. Por exemplo, um bloco ganho (1 entrada "u" e 1 saída "y") executa uma função do tipo $y = K.u$, onde $K$ define o ganho do bloco.

O modelo de um determinado dispositivo é geralmente composto de diversos blocos e parâmetros que definem seu conjunto de equações, além disso o modelo deve permitir ainda a inclusão de sub-modelos em sua estrutura. O diagrama de classes que descreve a estrutura computacional do modelo é mostrado na Figura 2.6.

O diagrama de classes da Figura 2.6 mostra o relacionamento do bloco base (classe *BLC*) com o seu conjunto de variáveis de entrada/saída (classes *VARINP* e *VAROUT*, respectivamente) através de uma relação de agregação, e a conectividade entre as variáveis de entrada/saída através de uma relação de associação. Dois grupos de classes derivadas de *VAROUT* constituem o conjunto de parâmetros (classes *PARM* e *REF*) e variáveis de estado do modelo (classes *VARSTT*, *VARDIF* e *VARALG*). Parâmetros e referências são um tipo especial de

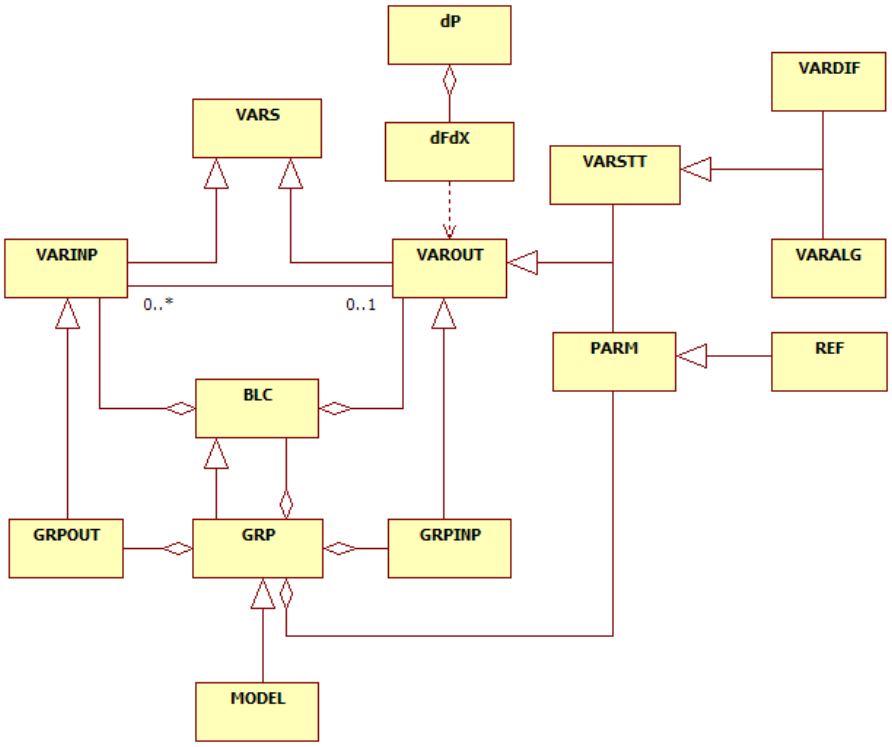


Figura 2.6: Diagrama de classes do *framework* de modelos.

variáveis de saída sem bloco associado, ou seja, um parâmetro pode ser entendido como uma variável de saída com valor numérico fixo e que pode ser conectado às entradas dos demais blocos do modelo. Variáveis de estado, por sua vez, representam um conjunto especial de variáveis de saída de determinados blocos (integradores, por exemplo) e definem os estados internos de um modelo. Assim, qualquer bloco de natureza dinâmica (integrador, *washout*, etc) necessariamente possuirá variáveis de saída do tipo diferencial (classe *VARDIF*). De forma semelhante, um bloco algébrico (somador, ganho, seno, etc) pode ter sua saída promovida ao "*status*" de variável algébrica (classe *VARALG*). Esta promoção deve ser especificada na construção do modelo.

### 2.4.2 Cálculo das Derivadas

Muitas vezes, as derivadas parciais de uma equação são de fácil determinação de forma manual, e é razoável esperar que o usuário construa um código para computá-las. Em outros casos, as funções são muito complicadas, então busca-se por outros meios, o cálculo ou mesmo uma aproximação das derivadas de maneira automática. Há várias maneiras de se obter as derivadas de primeira ordem de uma função. São elas:

- Calculo Manual: Cálculo de derivadas onde procura-se determinar a derivada de funções

de forma manual. Este tipo de cálculo é bastante utilizado, quando o número de funções a serem diferenciadas é pequeno e de simples diferenciação. Para funções mais complexas, a eficiência e a confiabilidade do cálculo de derivadas e na codificação dos resultados passa a ser de responsabilidade do programador, tornando o processo mais suscetível a erros.

- Calculo da Diferenciação Finita: Forma de cálculo onde as derivadas são aproximações do coeficiente angular de uma função em determinado ponto. A partir de pequenos incrementos no entorno de um dado ponto $x$, observa-se a mudança nos valores da função, podendo-se assim estimar a resposta a variações infinitesimais. A grande vantegem desse método é que não é necessário o conhecimento da função derivada. Por outro lado esse método pode apresentar problemas numéricos quando, por exemplo, em determinado ponto $x$ de uma função $f(x)$ ao realizar um pequeno incremento em $x$ obter como resposta uma grande variação de $f(x)$.

- Calculo da Diferenciação Simbólica: Técnica onde a função é especificada e trabalhada por ferramentas de manipulação simbólicas, a fim de produzir novas expressões algébricas para cada componente da derivada.

- Calculo da Diferenciação Automática: Técnica que leva em consideração que a função pode ser quebrada e composta por operações aritméticas elementares, no qual a regra da cadeia pode ser aplicada.

Este trabalho faz amplo uso do mecanismo de derivadas automáticas desenvolvido em [17].

### 2.4.3 Derivadas Automáticas do FASEE

O **FASEE** apresenta uma estrutura de armazenamento de equações que permite definir um mecanismo automático de solução e cálculo das derivadas parciais das equações, conferindo aos aplicativos um alto grau de generalização. Este mecanismo faz uso da regra da cadeia, na qual uma função relativamente complexa pode ser quebrada em funções mais simples para facilitar o cálculo da derivada. A regra da cadeia pode ser descrita conforme a Equação 2.2.

$$\frac{\partial f(u(x))}{\partial x} = \frac{\partial f(u)}{\partial u}.\frac{\partial u(x)}{\partial x} \tag{2.2}$$

As equações abaixo mostram um pequeno exemplo do uso da regra da cadeia.

$$\begin{aligned} f(x) &= sin(x^2) \\ f(g) &= sin(g) \\ g(x) &= x^2 \end{aligned}$$

$$\frac{\partial f(x)}{\partial x} = \frac{\partial f(g)}{\partial g}.\frac{\partial g(x)}{\partial x}$$
$$\frac{\partial f(x)}{\partial x} = cos(x^2).2.x$$

Para o cálculo das derivadas fez-se necessário implementar um conjunto de classes específicas (classes *dP* e *dFdX*) para armazenar e gerenciar as derivadas parciais de equações em relação ao seu conjunto de variáveis de estado (classes *VARDIF* e *VARALG*). O diagrama de classes da Figura 2.6 mostra o relacionamento entre as classes *dP* e *dFdX*, e com as demais classes que definem o modelo. Nesta estrutura, a classe *dFdX* armazena a derivada da equação em relação a uma única variável de estado, sendo constituída de um coeficiente numérico e uma associação a variável de saída correspondente ao estado (*VAROUT*). A classe *dP*, por sua vez, representa o conjunto de derivadas parciais de uma determinada equação em relação a todas as variáveis de estado que a equação depende, sendo implementada como um conjunto de classes *dFdX*. Assim os aplicativos dispõem de uma estrutura genérica de tratamento de derivadas parciais que é independente do modelo a ser utilizado no problema. A seguir serão apresentados detalhes do algoritmo de cálculo das derivadas parciais através de um exemplo para facilitar o entendimento do mecanismo.

### 2.4.4 Exemplo de Derivada Automática no FASEE

Tomando como exemplo a equação mostrada na Figura 2.4.1, aqui reapresentada na Figura 2.7, e atribuindo as variáveis $x$ e $y$ os valores 0.2 e 0.4, respectivamente, a equação exemplo e suas derivadas parciais em relação as variáveis de interesse assumem os valores apresentados abaixo:

$$\begin{array}{rcl} f(x,y) = sen(3.x + y^2) & & f(x,y) = 0.69 \\ \frac{\partial f(x,y)}{\partial x} = 3.cos(3.x + y^2) & \Rightarrow & \frac{\partial f(x,y)}{\partial x} = 2.17 \\ \frac{\partial f(x,y)}{\partial y} = 2.y.cos(3.x + y^2) \quad (x = 0.2; y = 0.4) & & \frac{\partial f(x,y)}{\partial y} = 0.58 \end{array}$$

Na Figura 2.7 as caixas em cinza localizadas acima das linhas que conectam os blocos representam o valor da função a partir das variáveis $x$ e $y$. Já os blocos localizados abaixo das linhas representam os valores das derivadas parciais da função, os quais são calculados a partir da variável, representadas pelos elementos $\diamond$, até o ponto de interesse, que no caso é saída do bloco SEN.

O mecanismo de solução e derivação das equações parte da variável de saída que define a equação que se deseja obter a solução e/ou o conjunto de derivadas parciais (ponto $f(x,y)$

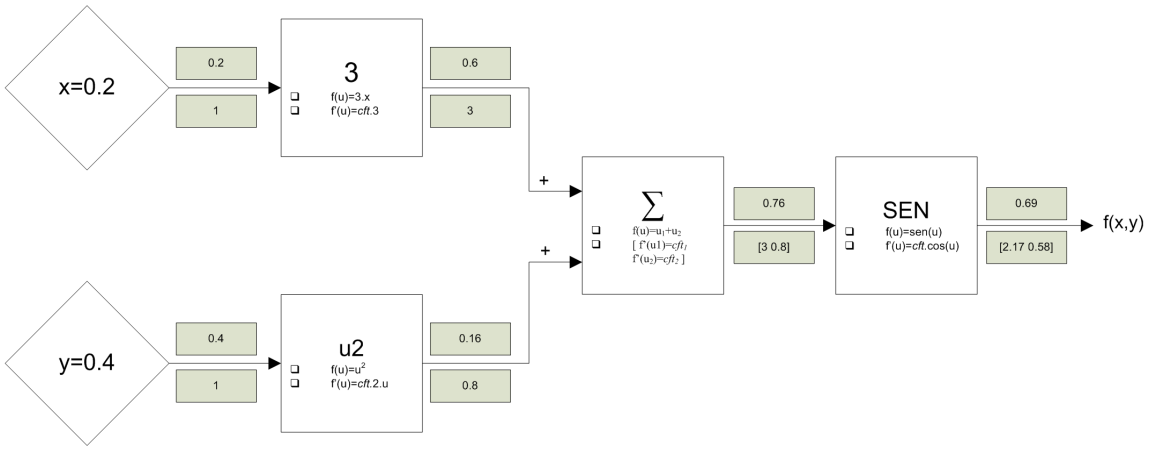


Figura 2.7: Mecanismo de solução e derivação do modelo.

na Figura 2.7). A partir deste ponto percorre-se o caminho seguindo a orientação inversa dos blocos (saída → entrada) até que uma variável de estado seja encontrada (*VARDIF* ou *VARALG*), no exemplo da Figura 2.7 todos os caminhos levam as variáveis $x$ e $y$. A escolha do caminho inverso é adequada devido a existência de apenas uma opção possível para cada variável de entrada dos blocos, uma vez que cada entrada é associada a apenas uma saída de outro bloco. Aliado a isto, é possível implementar um algoritmo de busca recursiva que define naturalmente o caminho percorrido. As variáveis de estado encontradas nesta busca determinam o conjunto de termos das derivadas parciais que serão calculados para a equação.

Após a identificação do conjunto de variáveis de estado que a equação é dependente percorre-se o caminho direto, a partir das variáveis de estado (o caminho direto já foi devidamente definido pela busca recursiva, de tal forma que não há a necessidade de buscá-lo novamente), atualizando, a cada passagem por um bloco elementar, a estrutura especial projetada para armazenamento das derivadas parciais (estrutura composta por instâncias da classe *dFdX*). A Figura 2.7 mostra os valores armazenados na estrutura auxiliar para todos os pontos do diagrama de blocos no caminho direto percorrido, sendo:

$$\left.\begin{array}{ll} [coeficiente]\ .\ variavel\ de\ estado & \rightarrow dFdX \\ [coeficiente]\ .\ variavel\ de\ estado & \rightarrow dFdX \end{array}\right\} dP$$

Cada vez que a classe *dP* passa por um bloco do diagrama uma operação particular é executada sobre esta estrutura. A operação depende do tipo de operação matemática executada no bloco, porém uma lei de formação geral pode ser formulada. A lei de formação é aplicada a todos os termos *dFdX* de *dP* e obedece a seguinte regra:

$$[coeficiente]_{antigo} \cdot \left[\frac{\partial f_{blc}}{\partial u}\right]_{u=uo} \quad \rightarrow \quad \text{novo coeficiente associado ao estado} \qquad (2.3)$$

onde o novo coeficiente de *dFdX* é obtido multiplicando-se o antigo coeficiente pela derivada da operação matemática implementada no bloco em relação a entrada. Admitindo como exemplo o bloco **sen** do diagrama de blocos da Figura 2.7 a lei de formação assume a seguinte forma:

$$[coeficiente]_{novo} = [coeficiente]_{antigo} * [cos(u)]_{u=uo} \tag{2.4}$$

O conjunto de coeficientes (termos *dFdX*) obtidos no ponto que define a equação são as derivadas parciais em relação ao conjunto de variáveis de estado identificadas pelo algoritmo, conforme mostram os resultados da Figura 2.7 concordando com os obtidos de forma analítica.

A detecção de um parâmetro, ou uma referência, durante o percurso inverso para identificação do conjunto de variáveis de estado também interrompe o processo de busca. Nesta situação é adotado o mesmo procedimento das variáveis de estado, porém a classe *dFdX* assume um coeficiente igual a 0 que determina a eliminação desta derivada (isto concorda com o fato da derivada de uma função em relação a uma constante ser nula).

### 2.4.5 Blocos Elementares

A Figura 2.8 mostra um diagrama de classes com alguns dos blocos construtivos elementares que são derivados do bloco base. Estes blocos constituem a base para a construção de qualquer modelo de dispositivo.

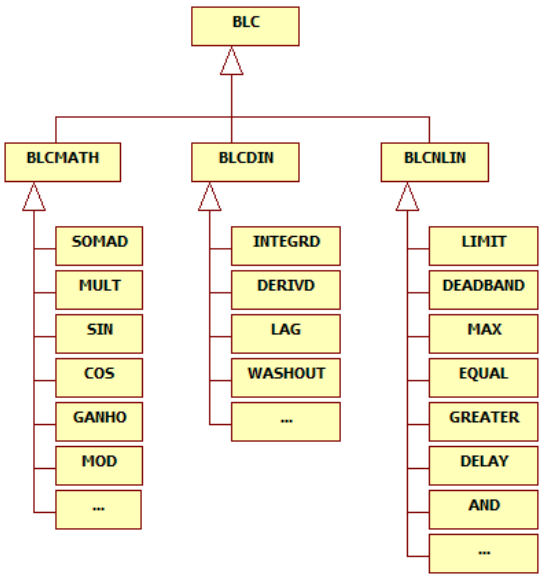


Figura 2.8: Blocos básicos para criação de modelos.

## 2.5 Considerações Gerais

Neste capítulo foi feito inicialmente uma introdução sobre programação orientada a objetos, passando por suas características gerais, vantagens e desvantagens. Em seguida realizou-se uma breve descrição sobre a estrutura base chamada *Framework para Análise de Sistemas de Energia Elétrica* (**FASEE**) [17], usa para o desenvolvimento deste trabalho. Finalmente foi descrito em detalhes a estrutura de modelos matemáticos utilizada para o desenvolvimento de redes neurais artificiais e lógica difusa, assim como suas facilidades para o cálculo de derivadas parciais.

Uma convenção que será bastante utilizada ao longo deste trabalho é a distinção entre classes derivadas de *BLC* e classes derivadas de *GRP* em diagrama de blocos da estrutura do **FASEE**, como mostra a Figura 2.9. A classe *GRP* derivada da classe *BLC* é tanto um agregador de blocos *BLC* quanto como agregador de subsistemas *GRP*, como ilustra o diagrama da Figura 2.10.

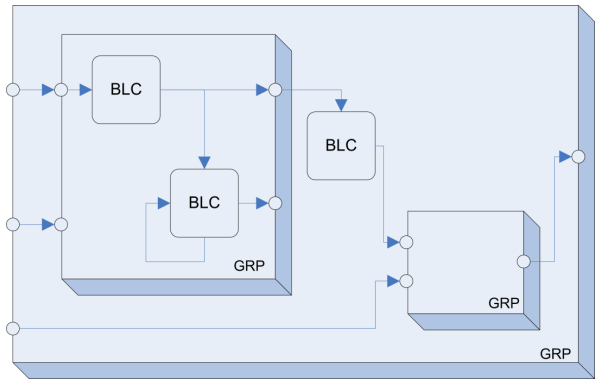


Figura 2.9: Convenção adotada para ilustrar as classes derivadas de *BLCs* e *GRPs*.

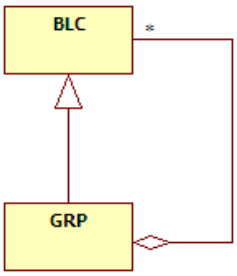


Figura 2.10: Relação entre as classes *BLCs* e *GRPs*.

Na Figura 2.9 se pode perceber que os blocos sem relevo são os blocos elementares derivados da classe *BLC*. Já os blocos com relevo são sistemas derivados da classe *GRP*.

# Capítulo 3

# Redes Neurais

Este capítulo é dedicado a redes neurais artificiais. Ele está dividido em quatro partes. A primeira descreve uma introdução de redes neurais artificiais, falando sobre conceitos e abstrações relativas a este assunto, e classes criadas para serem a base do que será o *framework* de redes neurais artificiais desenvolvido. Nas segunda e terceira partes são feitas uma descrição do que foi desenvolvido para redes Perceptron Multi-Camadas e Mapas Auto-Organizados. Finalmente são feitas algumas considerações em termos de desenvolvimento e dificuldades encontradas.

## 3.1 Introdução

Em termos biológicos redes neurais são estruturas as quais uma série de nós (neurônios) são interligados, através de conectores (sinapses) que transmitem pulsos elétricos entre os neurônios, de modo a fazer emergir o que pode ser chamada de comportamento inteligente [16] [19]. Neste contexto, redes neurais artificiais(RNAs) surgem para tentar, em um certo grau, reproduzir este comportamento, visando utilizar este conhecimento para solucionar problemas onde o nível de complexidade algébrica torna o mesmo inviável de ser resolvido através de métodos convencionais. Como exemplo, pode-se citar o reconhecimento da fala ou a segmentação de imagens, presente nos seres humanos, os quais são capazes de resolver tais problemas de forma inata. Sendo assim, podemos definir que redes neurais artificiais servem para resolver problemas de elevado grau de dificuldade, fazendo uso de um princípio simples, presente nos seres humanos, que é o aprendizado por observação, ou seja, aprender a partir de exemplos, e ainda generalizar a informação aprendida. Generalizar informações está associada à capacidade de uma rede aprender através de um conjunto reduzido de exemplos e, posteriormente, obter respostas coerentes para dados não-conhecidos.

### 3.1.1 Componentes de uma RNA

O elemento central de uma rede neural artificial é o neurônio. É nele que as informações aprendidas são armazenadas através dos seus pesos, isto é, suas conexões sinápticas. A Figura 3.1 mostra a generalização do modelo de McCulloch e Pitts, podendo este ser considerado o modelo geral de uma neurônio.

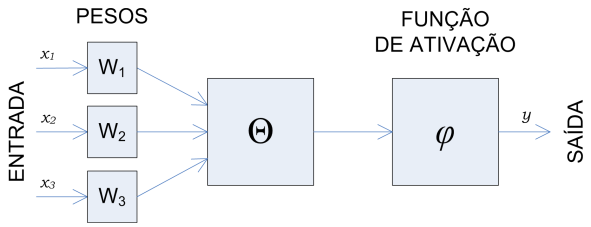


Figura 3.1: Modelo geral de um neurônio.

Nesta representação a função $\Theta$ geralmente é a soma das entradas, ou o produto das entradas. Já a função $\varphi$ é chamada de função de ativação, na qual ativa ou não a saída do neurônio, dependendo das suas entradas. Uma função de ativação é normalmente não-linear e crescente [1]. A Figura 3.2 mostra quatro exemplos de funções de ativação.

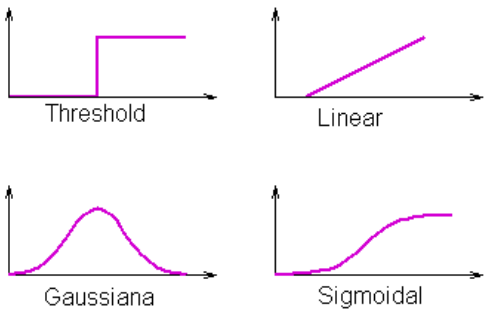


Figura 3.2: Exemplos de funções de ativação.

Quanto a arquitetura, existem alguns parâmetros que definem um RNA. Estes estão listados abaixo.

- Número de camadas: uma rede é constituída de uma camada de entrada, zero ou mais camadas escondidas ou intermediárias, e uma camada de saída. A Figura 3.3 mostra o exemplo de uma rede com duas camadas: uma escondida e a saída.

- Número de neurônios em cada camada: define o número de neurônios em uma determinada camada, os quais são ativados simultaneamente. No exemplo da Figura 3.3 a camada intermediária contém três neurônios e a camada de saída dois.

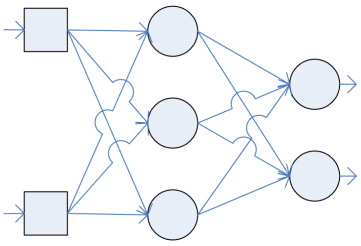

Figura 3.3: Rede com duas camadas.

- Tipo de conexão entre os neurônios: os dois principais tipos de conexões entre neurônios são: *feedfoward* ou acíclica, e *feedback* ou cíclica. Na primeira a saída dos neurônios de determinada camada não podem ser usadas como entrada de neurônios de camadas anteriores. Redes *feedback* são o oposto. Saídas de neurônios de uma camada podem ser usadas como entradas em neurônios de camadas anteriores. As Figuras 3.4 e 3.5 apresentam um exemplo de rede *feedfoward* e *feedback* respectivamente.

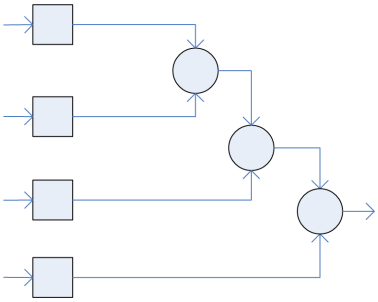

Figura 3.4: Rede *feedfoward*.

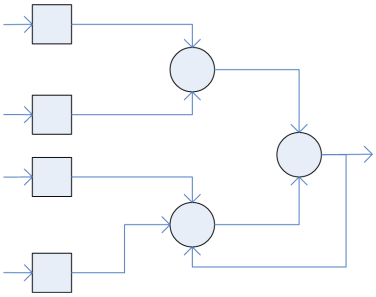

Figura 3.5: Rede *feedback*.

Existem também redes onde todas as ligações são cíclicas. Estas redes são denominadas *auto-associativas*. A Figura 3.6 apresenta um exemplo desta rede.

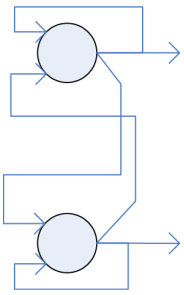

Figura 3.6: Rede *auto-associativa*.

- Topologia de rede: diz respeito a conexão existente entre os neurônios. São classificadas em redes fracamente(ou parcialmente) conectadas, Figuras 3.4 e 3.5, e redes completamente conectadas, Figura 3.6.

### 3.1.2 Estrutura Básica de Desenvolvimento de RNAs

Para que o *framework* de modelos do **FASEE** pudesse fazer uso de redes neurais, de tal forma que ficasse independente da implementação utilizada, foi necessário criar algumas classes bases, derivadas da classe *GRP*. A Figura 3.7 mostra um diagrama de classes com as classes que foram criadas.

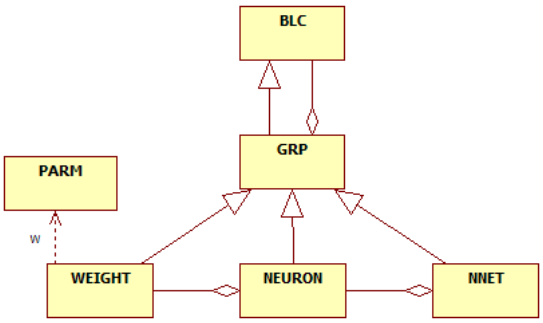


Figura 3.7: Diagrama de classes abstratas de redes neurais.

As classes *WEIGHT*, *NEURON* e *NNET* definem uma estrutura base no qual deve ser usado por qualquer implementação de redes neurais, indiferente de sua arquitetura. Isso facilita na hora de implementar uma nova rede, visto que tarefas comuns encontradas em todas implementações estão bem definidas, bastando apenas implementar os métodos virtuais. Abaixo é feita uma pequena descrição sobre cada classe.

#### Classe WEIGHT

Classe base usada para definir o peso sináptico de determinada entrada de um neurônio. Ela contém uma referência para uma instância da classe *PARM*.

#### Classe NEURON

Classe base que define o comportamento comum de um neurônio, entre diversas implementações possíveis do mesmo. Essa classe contém um lista de instâncias da classe *WEIGHT*, onde normalmente cada uma é associada a uma entrada no neurônio.

#### Classe NNET

Classe base de qualquer implementação de redes neurais. Ela contém uma lista de instâncias da classe *NEURON*.

### 3.1.3 Estrutura Básica de Treinamento de RNAs

Redes neurais artificiais possuem a capacidade de aprender por exemplos e fazer interpolações e extrapolacões do que aprendem através de um algoritmo de treinamento. Entende-se por *algoritmo de treinamento* um conjunto de procedimentos bem definidos os quais adaptam os parâmetros de uma RNA de modo que a mesma possa *aprender* uma determinada função. Existem diversos tipos de algoritmos de treinamento para os mais variados tipos de RNAs. O que difere um dos outros é a maneira na qual os pesos de uma rede são atualizados.

Os métodos para o treinamento de redes podem ser divididos em dois grupos: aprendizado supervisionado e aprendizado não-supervisionado. A seguir é feita uma breve descrição sobre cada tipo.

#### Aprendizado Supervisionado

Método no qual a entrada e a saída desejada da rede são fornecidas por um supervisor externo. Neste caso a rede tem a sua saída calculada comparada com a saída desejada, recebendo então informações do supervisor sobre o erro da resposta atual, assim direcionando o processo de treinamento. A minimização do erro é incremental de tal forma que pequenos ajustes feitos nos pesos a cada etapa de treinamento faça com que a rede caminhe para uma solução, se houver alguma. A maior desvantagem neste método é que, na ausência de um supervisor, a rede não conseguirá aprender novas estratégias para situações não contempladas pelos exemplos na fase de treinamento. A Figura 3.8 ilustra o mecanismo de aprendizado supervisionado.

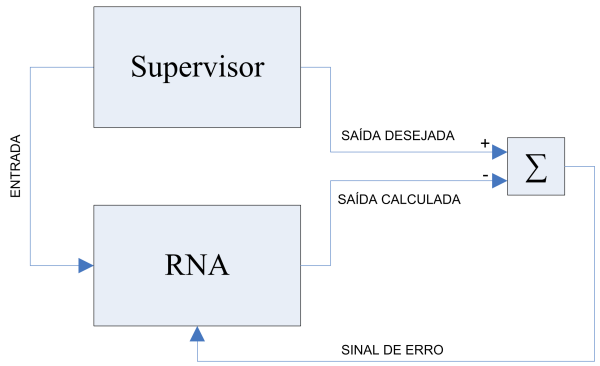


Figura 3.8: Aprendizado supervisionado.

Uma variação deste método é o aprendizado **por reforço**, onde a única informação de realimentação fornecida pelo supervisor é se determinada saída está correta ou não, ou seja, não é fornecido a resposta correta para o padrão de entrada.

### Aprendizado Não-Supervisionado

No aprendizado não-supervisionado, como o nome sugere, não há supervisor externo. Somente os dados de entrada da rede estão disponíveis. Este algoritmo serve para desenvolver a habilidade de formar representações internas para codificar características da entrada e criar novas classes ou grupos de forma automática, ou seja, segmentar o espaço dos dados de entrada em grupos específicos. A Figura 3.9 mostra o mecanismo de aprendizado não-supervisionado.

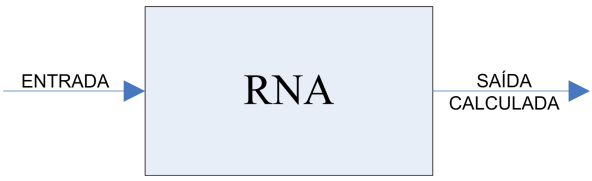


Figura 3.9: Aprendizado Não-Supervisionado.

Neste contexto de aprendizado não-supervisionado existe um caso particular chamado aprendizado **por competição**. A idéia é que a partir de um estímulo de entrada, as saídas da rede disputem entre si para serem ativadas, conseqüentemente fazendo com que os pesos da ganhadora sejam atualizados. Este algoritmo exige que as unidades de saída da rede seja conectados entre si, sendo essas conexões chamadas de conexões inibitórias. Durante o processo de treinamento os neurônios vencedores ficam cada vez mais fortes, fazendo com que o efeito inibitório sobre os neurônios adjacentes se torne mais forte também. Com o tempo somente o neurônio vencedor ficará ativo, e os demais inativos. Este processo também é chamado de *winner takes all*.

Um conjunto de classes bases foi criado de forma a abstrair conceitos e tarefas comuns a grande parte dos algoritmos de aprendizagem. A Figura 3.10 apresenta um diagrama de classes com as classes criadas para serem a base de desenvolvimento de algoritmos de treinamentos de RNAs.

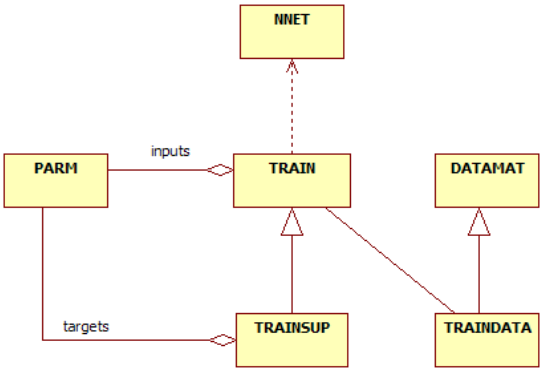


Figura 3.10: Diagrama de classes base para treinamento de RNAs.

Abaixo é feita uma breve descrição sobre cada classe.

### Classe TRAIN

Classe usada como base para os algoritmos de treinamento não-supervisionado. Ela contém uma referência para uma instância da classe *NNET*, além de diversos métodos para gerenciar instâncias da classe *PARM* durante a fase de treinamento, e métodos para se obter a saída da rede a partir de um determinado conjunto de entradas, organizado em forma de matriz.

### Classe TRAINSUP

Classe base usada para algoritmos de treinamento supervisionados. Ela contém todas as funcionalidades da classe *TRAIN*, acrescida de uma lista e métodos de gerenciamento de instâncias da classe *PARM*, as quais representam saídas desejadas do sistema.

Além das classe descritas acima, foi necessário criar algumas classes auxiliares durante o desenvolvimento de modo a facilitar a manipulação de dados de entrada e saída, tanto para ajudar no processo de codificação, quanto para auxiliar desenvolvedores durante o uso do sistema. A Figura 3.11 mostra o diagrama de classes com as classes que foram criadas.

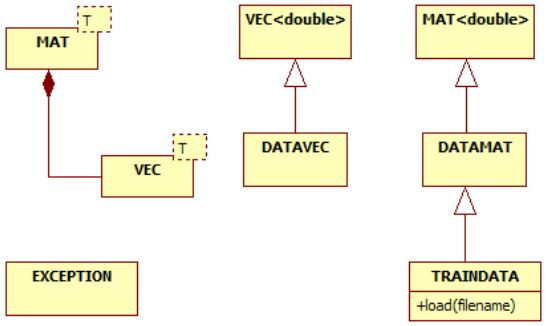


Figura 3.11: Diagrama de classes base para treinamento de RNAs.

### Classes Auxiliares

Abaixo é feita uma breve descrição sobre cada classe.

- Classe *VEC<>*: classe *template* usada para armazenar dados em forma de vetor;
- Classe *MAT<>*: classe *template* usada para armazenar dados em forma de matriz. Contém uma lista de instâncias da classe *VEC<>*;

- Classes *DATAVEC* e *DATAMAT*: são simples definições de tipos(*typedef*);
- Classe *TRAINDATA*: classe derivada da classe *DATAMAT* onde um método chamado *load(filename)* foi acrescentado de forma a facilitar a manipulação de grande quantidade de dados. Este carrega uma matriz de dados a partir de um arquivo texto;
- Classe *EXCEPTION*: classe usada para tratamento de erros. Ela deve ser usada em blocos tipo *try*{ ... } *catch( EXCEPTION e)* { ... } durante o processo de desenvolvimento para se obter mensagens detalhadas de erros, caso alguma inconsistência for encontrada durante a execução do programa.

## 3.2 MLP

As redes Perceptron Multi-Camadas, ou em inglês chamadas de *Multi-Layer Perceptron* (MLP), tem como objetivo resolver problemas não linearmente separáveis, através do uso de uma ou mais camadas intermediárias. Redes de uma única camada de neurônios Perceptron são capazes de resolver apenas problemas linearmente separáveis, limitando o seu uso a problemas cuja solução pode ser obtida dividindo-se o espaço de entrada em duas regiões através de uma reta. Com redes MLP de uma camada intermediária é possível aproximar qualquer função contínua. Com duas é possível aproximar qualquer função matemática [1].

A arquitetura de uma rede MLP é bem definida, onde se encontra uma camada de entrada de dados, uma ou mais camadas intermediárias, e uma camada de saída. Esta rede é considerada ser acíclica, também chamadas de *feedforward*, pois as saídas dos neurônios de uma determinada camada não podem ser usadas como entrada de neurônios de uma camada anterior. A Figura 3.12 mostra uma rede MLP típica, com uma camada intermediária.

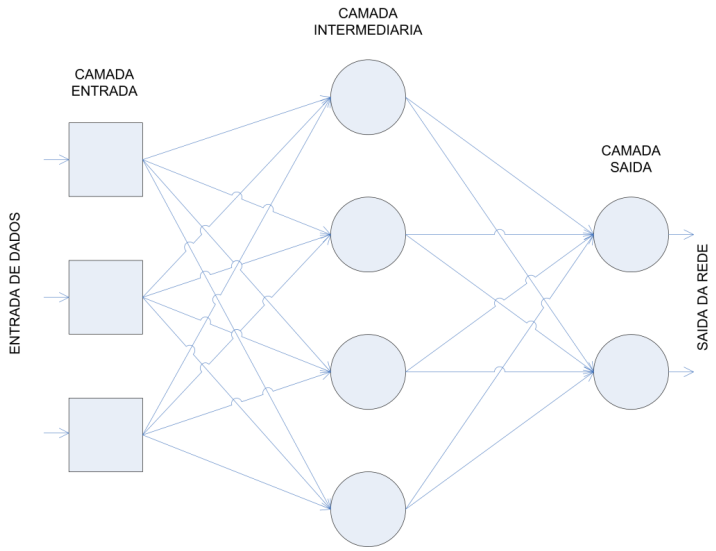


Figura 3.12: Rede MLP típica com uma camada intermediária.

A Figura 3.13 representa um neurônio típico de uma rede MLP, representada na Figura 3.12 pelo símbolo ◯.

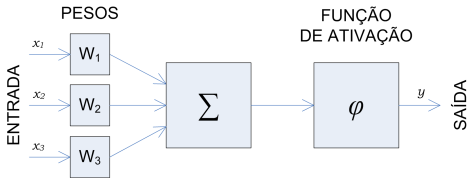


Figura 3.13: Neurônio típico de uma rede MLP.

A Equação 3.1 demonstra em termos matemáticos o comportamento de uma neurônio.

$$y = \varphi(\sum_{i=1}^{n} x_i.w_i) \tag{3.1}$$

onde $n$ é o número de entradas do neurônio, e $\varphi$ a função de ativação.

### 3.2.1 Projeto *Framework* de Redes MLP

Nesta seção é apresentado o projeto de redes MLP desenvolvido. A Figura 3.14 mostra o diagrama de classes criado para habilitar o **FASEE** a utilizar este tipo de rede.

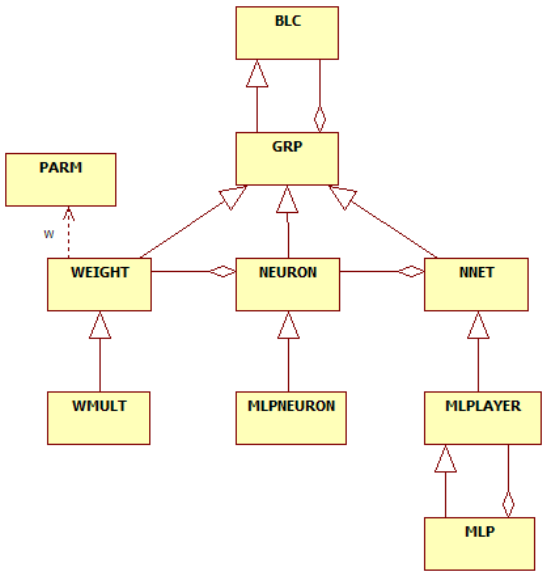


Figura 3.14: Diagrama de classes projetado para redes MLP.

A seguir é feita uma descrição detalhada sobre o que cada classe representa.

### Classe WMULT

Classe derivada da classe *WEIGHT* usada para compor uma estrutura usada nas entradas de cada neurônio. Ela conta com uma entrada e uma saída. A Equação 3.2 mostra que a operação matemática realizada por este bloco é muito simples, onde a saída é a entrada multiplicada por uma peso.

$$y = w.u \tag{3.2}$$

onde $y$ é saída do bloco, $w$ o próprio valor do peso, e $u$ a entrada do bloco.

Este bloco poderia ser substituído por um bloco tipo ganho, mas um dos requisitos do sistema é que para a etapa de treinamento, faz-se necessário obter a derivada parcial do erro da rede em relação ao peso, o que não seria possível com o bloco ganho. A Figura 3.15 mostra o diagrama de blocos resultante.

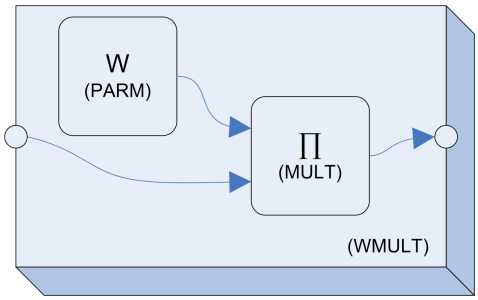


Figura 3.15: Diagrama de blocos da classe *WMULT*.

### Classe MLPNEURON

Classe derivada da classe *NEURON* que representa um neurônio genérico em uma rede MLP. A Figura 3.16 mostra o diagrama de blocos de um neurônio MLPNEURON usando a estrutura de blocos do **FASEE**.

A função que descreve seu comportamento é a mesma mostrada na Equação 3.1, aqui transcrita na Equação 3.3.

$$y = \varphi(\sum_{i=1}^{n} x_i.w_i) \tag{3.3}$$

onde $\varphi$ representa a função de ativação.

Deve-se reparar que a função de ativação do neurônio é um bloco genérico, isto é, para um neurônio *MLPNEURON* o que interessa é a saída da função de ativação, e não a sua implementação, podendo esta ser definida em tempo de execução do programa.

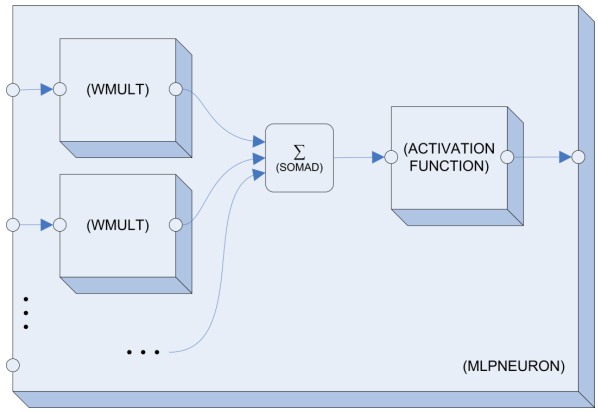


Figura 3.16: Diagrama de blocos da classe MLPNEURON usando a estrutura do **FASEE**.

### Classe MLPLAYER

Esta classe foi criada para fins de controle e facilidade de implementação. Ela contém métodos usados para a criação tanto de camadas intermediárias como a camada de saída de uma rede. Usada como um agregador de neurônios, ela nasceu com o intuito de facilitar a tarefa de programação no momento da implementação de algum algoritmo de treinamento. A Figura 3.17 mostra o diagrama de blocos de uma MLPLAYER.

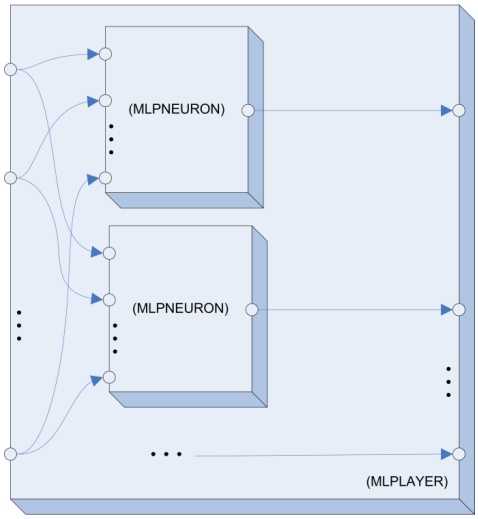


Figura 3.17: Diagrama de blocos da classe MLPLAYER usando a estrutura de blocos do **FASEE**.

### Classe MLP

Esta é a classe principal de redes MLP. Derivada de *MLPLAYER*, onde também contém uma lista de *MLPLAYER*, constituindo uma padrão de projetos chamado *Composite*. Ela contém métodos usados para o gerenciamento das camadas de uma rede. A Figura 3.18

mostra o diagrama de blocos de uma rede MLP exemplo com duas camadas: a primeira, intermediária, com dois neurônios, e a segunda, a de saída, com três neurônios.

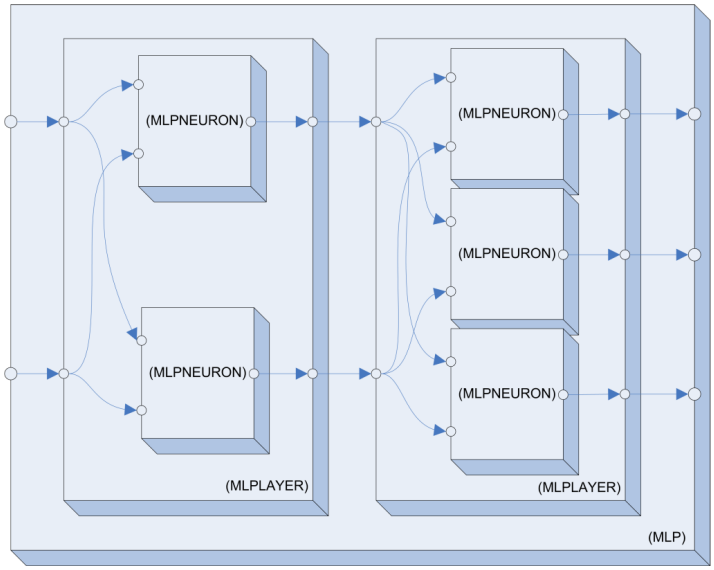


Figura 3.18: Diagrama de blocos da classe MLP usando a estrutura de blocos do **FASEE**.

Abaixo é apresentado um trecho de código usado para exemplificar a criação de uma rede MLP.

```
1  #include "mlp.h"
2
3  void main()
4  {
5      // cria instancia da rede
6      MLP* mlp = new MLP( 2 /*numero de entradas*/ );
7
8      // adiciona camada escondida
9      mlp->addlayer( 2 /*numero de neuronios*/ );
10
11     // adiciona camada de saida
12     mlp->addlayer( 3 /*numero de neuronios*/);
13
14
15     // usa a rede...
16
17
18     // apaga instancia
19     delete mlp;
20 }
```

### 3.2.2 Treinamento de Redes MLP

O grande problema de redes de mais de uma camada é o seu treinamento. A inexistência ou desconhecimento de algoritmos para treinar redes com uma ou mais camadas intermediárias

foi uma das causas da redução das pesquisas em RNAs na década de 70.

Atualmente existem diversos algoritmos de treinamento de redes MLP, os quais a maioria é supervisionado. O algoritmo mais conhecido para o treinamento destas redes é o *back-propagation*, sendo que a maioria dos métodos de aprendizado de redes MLP são variações desse algoritmo.

O treinamento ocorre em duas fases: *forward* e *backward*. A fase *forward* é usada para definir a saída da rede para um determinado padrão de entrada. Já a fase *backward* é usada para atualizar os pesos das conexões da rede a partir da diferença entre a saída desejada da rede e a saída calculada. A Figura 3.19 ilustra as duas fases do treinamento.

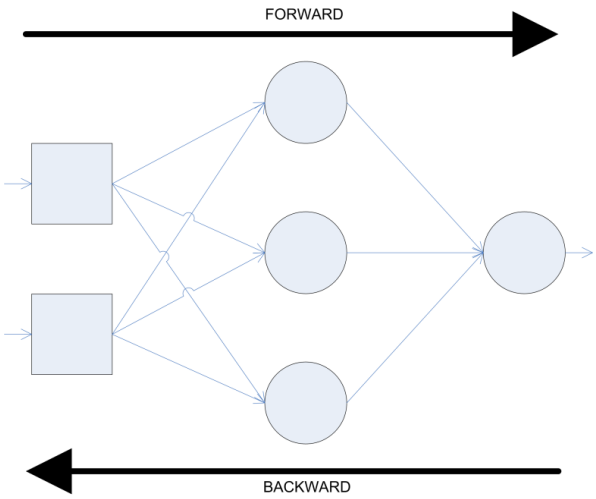


Figura 3.19: Fases do treinamento de uma rede MLP.

Durante o processo de treinamento diversos padrões de entrada e saída são apresentados a rede na fase *forward*. A partir de uma critério que define o erro da rede, a fase *backward* faz com que os pesos da redes sejam atualizados de forma proporcional ao erro resultante de cada um dos padrões.

Foram criadas duas classes para a codificação do processo de treinamento, as classes *BP* e *SQRERR*, que representam respectivamente o algoritmo *back-propagation* e o erro médio quadrático. A Figura 3.20 mostra o diagrama de classes para treinamento de redes MLP.

Inicialmente será feita uma descrição do critério que define o erro de uma rede, e a seguir sobre o algoritmo de treinamento em si, o *back-propagation*, fazendo-se referências as classes criadas.

### Erro Médio Quadrático

O Erro Médio Quadrático, em inglês *Mean Square Error*(MSE), classe *SQRERR*, é usado para avaliar o quão longe de uma solução a rede se encontra a cada passo do processo de treinamento. Este valor é calculado a partir da diferença ponderada entre as saídas desejadas

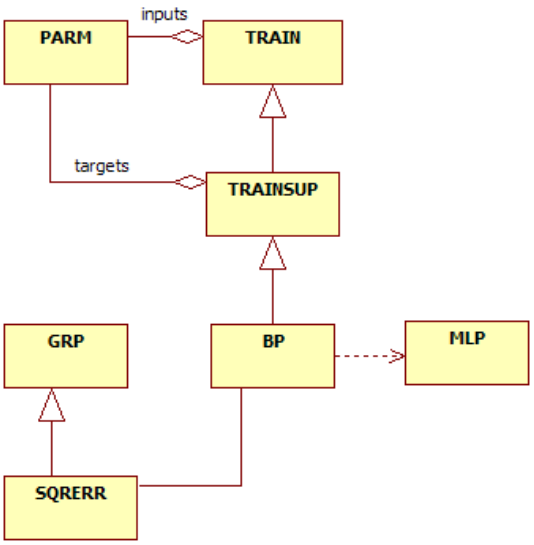


Figura 3.20: Diagrama de classes para treinamento de redes MLP.

e as saídas calculadas da rede. A Equação 3.4 mostra a expressão do MSE.

$$E = \frac{1}{2}\sum_{p}^{P}\sum_{i=1}^{k}(d_i^p - y_i^p)^2 \tag{3.4}$$

onde $E$ é a medida total do erro, $P$ é o número de padrões, $k$ é o número de neurônios de saída, $d_i$ é a $i$-ésima saída desejada e $y_i$ a $i$-ésima saída gerada pela rede. Este valor de erro $E$ quantifica o erro cometido pela rede para todos os padrões $P$ de entradas e saídas.

Neste trabalho foi desenvolvida uma classe, um bloco na estrutura do **FASEE**, chamado *SQRERR* o qual representa o MSE usando a estrutura de diagrama de blocos do **FASEE**. Seu desenvolvimento foi feito usando os blocos do **FASEE** uma vez que era necessário o uso da derivação automática. A Figura 3.21 mostra o diagrama de blocos do MSE.

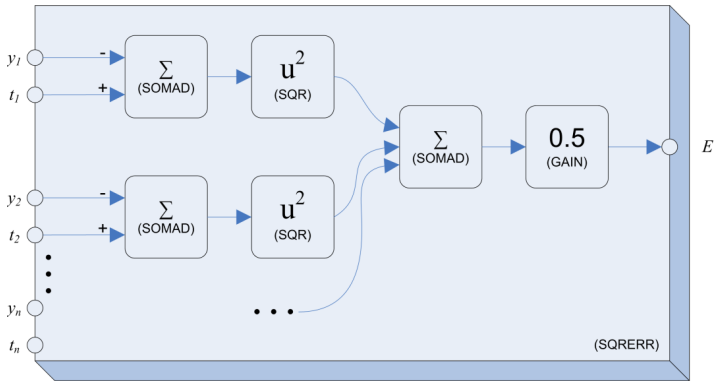


Figura 3.21: Diagrama de blocos do MSE usando a estrutura do **FASEE**.

### Back-Propagation

Como colocado anteriormente, o *back-propagation*, *BP*, é um algoritmo que utiliza pares de dados – entradas e saídas desejadas – para que através de um mecanismo de correção de erros, ajustar os pesos de uma rede. O algoritmo *back-propagation* foi, essencialmente, inventado e popularizado por Rummelhart, Hilton e Williams [27], resolvendo uma das limitações para o treinamento de redes complexas. Baseado na regra delta proposta por Widrow na década de 60 [31], ele propõe uma forma de definir o erro dos neurônios das camadas intermediárias, possibilitando assim os seus ajustes através do método gradiente descendente.

A Figura 3.22 mostra o exemplo de uma superfície de erro e os pontos dos gradientes descendentes calculados até atingir o valor de mínimo da superfície.

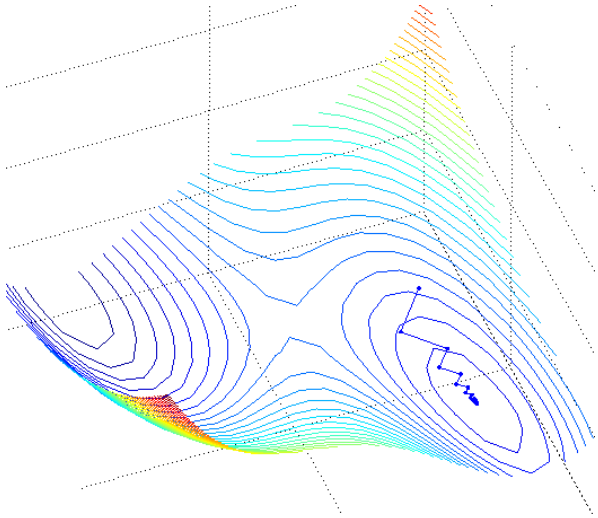

Figura 3.22: Superfície de erro de uma rede MLP e a trajetória do gradiente descendente.

No algoritmo *back-propagation*, considera-se que a minimização do erro para cada padrão entrada/saída levará a minimização do total $E$, Equação 3.4. Assim a equação do erro passa a ser definida como na Equação 3.5.

$$E = \frac{1}{2}\sum_{i=1}^{k}(d_i^p - y_i^p)^2 \tag{3.5}$$

A regra delta propõe que os pesos sejam atualizados de forma proporcional ao gradiente do erro em relação a determinado peso. Assim a variação dos pesos para cada dado padrão é definido pela Equação 3.6

$$\Delta w_{ij} \propto -\frac{\partial E}{\partial w_{ij}} \tag{3.6}$$

onde $w_{ij}$ é o peso da entrada $i$ do neurônio $j$, podendo este ser de qualquer camada da rede MLP.

A Figura 3.2.2 mostra o diagrama de blocos de uma rede MLP exemplo com duas camadas: a primeira é a camada escondida com 2 neurônios e a segunda é a camada de saída com 3 neurônios. Esta saída da rede está conectada ao erro total, conforme a Equação 3.5.

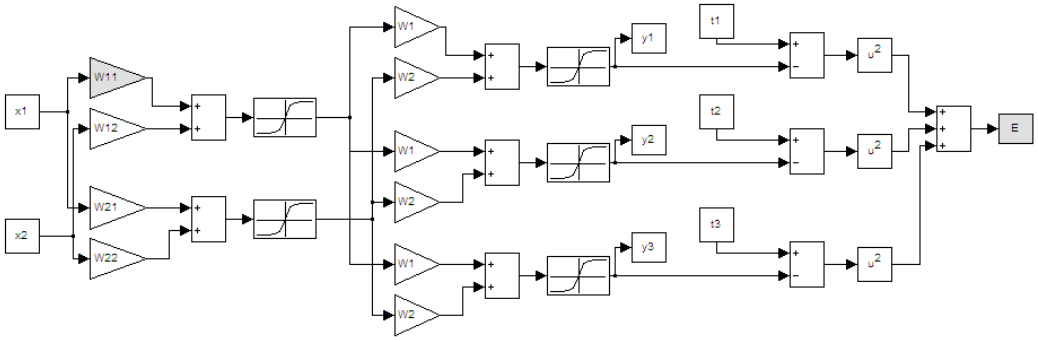


Figura 3.23: Rede MLP de duas camadas com 5 neurônios conectadas ao MSE.

Na Figura 3.2.2, pode-se observar que existem dois blocos em cinza: o peso $w_{11}$ e o erro médio quadrático $E$. Para se atualizar o peso $w_{11}$ conforme a Equação 3.6, deve-se realizar diversas operações matemáticas, utilizando a regra da cadeia, para se obter $\partial E/\partial w_{11}$. Os passos necessários para se chegar a esse resultado são descritos em diversas referências como [1] e [12], e portanto não serão replicados aqui. O importante é saber que um determinado peso é atualizado conforme a Equação 3.7, mostrada a baixo, onde $\eta$ é chamado de taxa de aprendizado.

$$w_{ij}(t+1) = w_{ij}(t) + \eta . \left( \frac{\partial E}{\partial w_{ij}} \right) \tag{3.7}$$

Um dos principais motivos para criação deste *framework* de redes neurais artificiais usando a estrutura do **FASEE** foi justamente o seu mecanismo de cálculo de derivadas parciais, que simplifica de forma significativa o processo de desenvolvimento. A Figura 3.2.2 ilustra o diagrama de blocos usando para treinamento de redes MLP.

No diagrama apresentado na Figura 3.2.2 pode-se notar que um bloco *SQRERR* é conectado a saída de um *MLP*. Também são adicionados algumas instâncias da classe *PARM* usadas tanto para representar as entradas da rede($x_1$ e $x_2$) quanto para representar as saídas desejadas($t_1$, $t_2$ e $t_3$). Para o cálculo das derivadas parciais dos pesos em relação ao erro $E$ basta criar uma instância da classe *dP*(derivada parcial) e, a partir da saída do bloco de erro(*SQRERR*), acionar o método *linear* da saída do bloco, instância da classe *VAROUT*, de modo que o mecanismo desenvolvido no **FASEE** para o cálculo de derivadas parciais retorne uma conjunto de instâncias da classe *dFdX*, a qual cada uma representa a derivada parcial do erro $E$ em relação a todos os pesos $w_{ij}$.

O código abaixo mostra um exemplo de como se usa o algoritmo de treinamento desen-

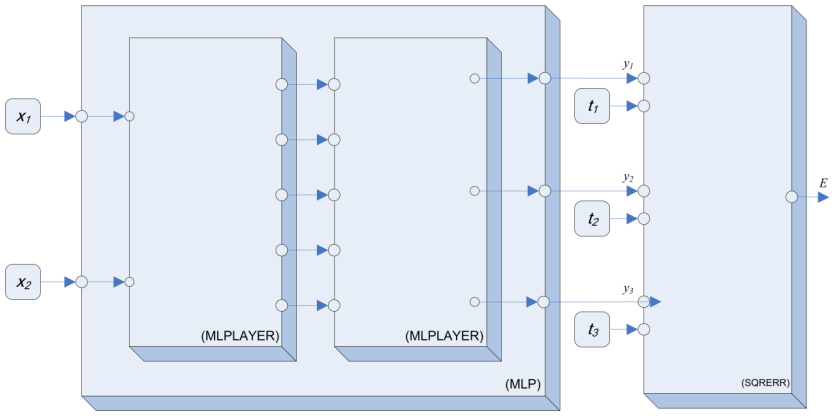


Figura 3.24: Diagrama de blocos do BP.

volvido, em uma rede MLP.

```
1  #include "mlp_train"
2
3  void main()
4  {
5      // cria instancia da rede...
6      MLP* mlp = new ...
7
8      // dados de treinamento...
9      DATAMAT inputs(...
10     DATAMAT targets(...
11
12
13     // cria instancia do algoritmo de treinamento
14     BP* bp = new BP( mlp /*instancia da rede*/ );
15
16     // define numero maximo de epocas
17     bp->maxepoch( 100 );
18
19     // define taxa de aprendizado
20     bp->learnrate( 0.5 );
21
22     // define tolerancia maximo do erro
23     bp->maxerror( 0.001 );
24
25     // treina a rede
26     int ret = bp->train( inputs /*entradas*/, targets /*saidas desejadas*/, NULL /*
           callback*/ );
27
28     // apaga instancia do treinamento
29     delete bp;
30
31
32     // usa a rede treinada ...
33
34
35     // apaga instancia da rede
```

```
36     delete mlp;
37 }
```

A seguir é feita uma descrição detalhada dos pontos mais importantes do que foi feito no código acima.

- Cria instância da rede...

  ```
  MLP* mlp = new ...
  ```

  Primeiramente, foi criada uma instância da rede classe *MLP*, não entrando em detalhes sobre sua configuração.

- Dados de treinamento...

  ```
  DATAMAT inputs(...
  DATAMAT targets(...
  ```

  Criação de duas instâncias da classe *DATAMAT* chamadas *inputs* e *targets*, as quais representando os pares de entradas e saídas desejadas da rede. Estas informações podem vir de qualquer fonte, sendo a classe *DATAMAT* usada como uma forma de padronizar a manipulação dos mesmos.

- Cria instância do algoritmo de treinamento.

  ```
  BP* bp = new BP( mlp /*instancia da rede*/ );
  ```

  Cria uma instância da *BP* passando como parâmetro para o construtor parametrizado a rede *mlp*. Neste momento são criadas as instâncias da classe *PARMs*, usadas para representar os dados de entrada e saída desejados, e a instância da classe *SQRERR*, as quais são conectadas em seqüencia de forma adequada conforme ilustra o diagrama da Figura 3.2.2.

- Definição de alguns parâmetros de treinamento.

  ```
  bp->maxepoch( 100 );
  bp->learnrate( 0.5 );
  bp->maxerror( 0.001 );
  ```

  São definidos alguns parâmetros como máximo número de épocas, taxa de aprendizado e tolerância do erro.

- Treina a rede.

  ```
  int ret = bp->train( inputs /*entradas*/, targets /*saidas desejadas*/, NULL
      /*callback*/ );
  ```

  Para efetivamente realizar o treinamento da rede o método *train* deve ser chamado, passando-se por parâmetro os dados de entradas e saídas desejados. Este método pode

também receber um terceiro parâmetro o qual é uma referência para uma função que será chamada a cada iteração do algoritmo. Esta é uma funcionalidade interessante para visualização do andamento do treinamento, o qual tem em sua lista de parâmetros dados importantes como época atual e o erro médio quadrático atual da rede.

O retorno dessa função, que é definido pelo usuário, indica se o algoritmo de treinamento deve ou não proceder, mesmo que nenhum critério de parada tenha sido atingido.

No algoritmo padrão de treinamento *back-propagation* os pesos de todos os neurônios são atualizados no momento que um determinado padrão de entradas e saídas desejadas é exposto a rede, sendo este método chamado de treinamento **por padrão**. Existe também um método chamado **por ciclo**, também conhecido como treinamento *batch*, no qual todos os padrões são expostos a rede, e só então os pesos são atualizados através de uma média de cada $\partial E/\partial w_{ij}$ de todos as amostras do conjunto de treinamento. A principal diferença entre o treinamento por padrão para o treinamento por ciclo é que neste segundo a convergência se dá de maneira mais estável, porém mais lenta. O treinamento por ciclo também foi desenvolvido neste trabalho, podendo este ser acessado apenas alterando-se o método chamado da classe *BP* de *train* para *train_batch*.

Uma variação do algoritmo *back-propagation* chamando *Resilient Back-propagration*, freqüentemente encontrada em implementações de RNAs, também foi desenvolvido. Detalhes sobre esse método são descritos a seguir.

*Resilient Back-propagation*

O *Resilient Back-propagation*(RPROP), desenvolvido por [25], é um esquema de aprendizado adaptativo local, aplicado em aprendizados por ciclos(*batch learning)* em algoritmos de treinamento de redes MLP. O princípio básico deste método é eliminar a má influência do tamanho da derivada parcial do erro em relação a cada peso. Em conseqüência disso, somente o sinal da derivada é levado em consideração, indicando a direção na qual cada peso deve ser atualizado. A Equação 3.8 mostra como determinado peso de uma rede, em função do tempo, é atualizado.

$$\Delta w_{ij}^{(t)} = \begin{cases} -\Delta_{ij}^{(t)} & , \text{ se } \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ +\Delta_{ij}^{(t)} & , \text{ se } \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\ 0 & , \text{ senão} \end{cases} \tag{3.8}$$

O termo $\frac{\partial E}{\partial w_{ij}}^{(t)}$ é na verdade a média de cada derivada parcial para todas as amostras. A Equação 3.9 mostra como o termo $\Delta_{ij}^{(t)}$ é calculado.

$$\Delta_{ij}^{(t)} = \begin{cases} \eta^+ * \Delta_{ij}^{(t-1)} & , \text{ se } \frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ \eta^- * \Delta_{ij}^{(t-1)} & , \text{ se } \frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\ \Delta_{ij}^{(t-1)} & , \text{ senão} \end{cases} \tag{3.9}$$

$$\text{onde } 0 < \eta^- < 1 < \eta^+$$

Este algoritmo faz com que, a cada passo do treinamento, os pesos sejam atualizados conforme o sinal de sua derivada em relação ao erro, ou seja, no momento que a derivada parcial do erro em relação a um peso não alterar de sinal em relação ao momento anterior($\frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} > 0$), o tamanho do seu passo aumenta em $\eta^+$, e quando o sinal alterar($\frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} < 0$) o passo é reduzido na proporção de $\eta^-$. Assim, quando o erro atual estiver localizado em uma superfície onde há pouco variação, o passo se torna maior, fazendo com que a convergência se torne mais rápida. Por outro lado, quando o erro atual estiver em um ponto da superfície do erro onde há grandes variações, o tamanho do passo é reduzido de forma que o algoritmo não ultrapasse um mínimo local.

Geralmente a escolha dos valor para $\eta^-$ e $\eta^+$ são, respectivamente, 0.5 e 1.2 conforme é demonstrado em [25]. Os limites máximos e mínimos de $\Delta_{ij}$ também são definidos como 50.0 e $1e^{-6}$.

Além de promover um significativo aumento na velocidade de convergência, o RPROP também traz a facilidade de não haver a necessidade de escolha do parâmetro $\eta$ do algoritmo BP padrão, onde na maioria das vezes é um dado altamente correlacionado com o domínio do problema.

Para se treinar uma rede MLP com RPROP usando a estrutura criada neste trabalho, basta utilizar o método *train_rprop* da classe *BP*.

### 3.2.3 Exemplo: Porta XOR

Um exemplo clássico de redes MLP é a aproximação de uma porta *ou-exclusivo*, mais conhecida como porta **XOR** (do inglês *eXclusive OR*). O problema é que esta porta apresenta valores que são não-linearmente separáveis, onde uma rede contendo apenas uma camada não é capaz de aproximar essa função dado que seus pontos não podem ser divididos por somente uma linha reta. Abaixo encontra-se a listagem completa do programa, escrito em **Lua**, para resolver esse problema.

```
1 -- carrega framework de RNA (arquivo 'luafann.dll')
2 require 'luafann'
3
4 -- cria matriz de dados de entradas
5 inp = datamat(4,2)
6
7 -- define dados entrada
8 inp:set(0,0, 0.0); inp:set(0,1, 0.0)
```

```
9  inp:set(1,0, 0.0); inp:set(1,1, 1.0)
10 inp:set(2,0, 1.0); inp:set(2,1, 0.0)
11 inp:set(3,0, 1.0); inp:set(3,1, 1.0)
12
13 -- cria matriz de dados de saída desejados
14 target = datamat(4,1)
15
16 -- define dados de saída desejados
17 target:set(0,0, 0.0)
18 target:set(1,0, 1.0)
19 target:set(2,0, 1.0)
20 target:set(3,0, 0.0)
21
22 -- cria rede mlp
23 m = mlp()
24
25 -- define estrutura da rede: 2 entradas, 2 neurônios na camada escondida, 1 saída
26 m:setup( 2, 2, 1 )
27
28 -- define funcao de callback
29 function callback( msg )
30   print( string.format("Epoch: %d \t MSE: %.4f", msg.epoch, msg.mse ) )
31   return 0
32 end
33
34 -- realiza o treinamento da rede
35 m:train( inp, target, "callback" )
36
37 -- cria matrix para guardar a saída da rede
38 out = datamat(4,1)
39
40 -- testa a rede com a mesma entrada do treinamento
41 m:solve( inp, out )
42
43 -- imprime saída da rede
44 for i=0,3 do
45   print( 'Entrada (' .. inp:get(i,0) .. ',' .. inp:get(i,1) .. ') - Saída (' .. out:
       get(i,0) .. ')' )
46 end
```

Ao executar este programa ele imprime na tela o seguinte:

```
Epoch: 0          MSE: 0.2488
Epoch: 1          MSE: 0.2467
Epoch: 2          MSE: 0.2395
Epoch: 3          MSE: 0.2111
Epoch: 4          MSE: 0.1270
Epoch: 5          MSE: 0.0779
Epoch: 6          MSE: 0.0448
Epoch: 7          MSE: 0.0221
Epoch: 8          MSE: 0.0153
Epoch: 9          MSE: 0.0068
Entrada (0,0) - Saída (0.095145986821804)
Entrada (0,1) - Saída (0.86300361434239)
Entrada (1,0) - Saída (0.86587811497652)
```

```
Entrada (1,1) - Saída (0.094095352175532)
```

Deve-se salientar que em muitos casos o algoritmo de treinamento não foi capaz de achar uma solução para o problema, ficando sobre um mínimo local.

### 3.2.4 Considerações

A seguir são feitas algumas considerações práticas no que diz respeito ao projeto e a implementação de redes MLP.

- Redes com duas ou mais camada intermediárias podem facilitar o treinamento, mas a utilização de um grande número de camadas intermediárias não é recomendado, visto que cada vez que o erro medido, durante o treinamento, é propagado para a camada anterior, onde o mesmo se torna menos útil e preciso. A última camada da rede tem uma noção precisa do erro cometido. Já que a última camada intermediária é uma estimativa do erro, a penúltima camada intermediária tem somente um estimativa da estimativa, e assim por diante.

- Um dos principais aspectos relacionados ao projeto de redes MLP diz respeito à função de ativação utilizada. Embora a estrutura proposta neste trabalho seja bastante flexível a ponto de se poder escolher ou criar qualquer tipo de função de ativação possível, a função mais utilizada é a *sigmoidal* visto que a mesma é não-linear e diferenciável. Caso a função de ativação seja linear, esta se equivale a uma rede de uma só camada. A função precisa ser diferenciável para que o gradiente possa ser calculado, direcionando os pesos. Estas duas características são muito importantes visto que os diversos métodos de treinamento fazem uso disso.

- Um problema que se pode encontrar no treinamento de redes MLP é chamado de *underfitting*, onde a rede não converge para um mínimo global pela fato do número de conexões, camadas escondidas e número de neurônios, ser inferior ao que o problema requer. Outro problema, freqüentemente encontrado no processo de treinamento, é chamado de *overfitting*, no qual o número de conexões da rede excede a quantidade apropriada para o problema em questão, no que diz respeito ao número de entradas, saídas e amostras, fazendo com que a rede perca seu poder de generalização memorizando os padrões de treinamento em vez de extrair as suas características gerais.

## 3.3 SOM

Mapas auto-organizados, conhecidos também por *Self-Organizing Maps*(SOM), são redes neurais artificiais que possuem a capacidade de auto-organização. Desenvolvida por Teuvo

Kohonen na década de 80, sua principal característica é que ela aprende através de exemplos, sem a necessidade de um agente externo, ou seja, aprendizado não-supervisionado [15].

Redes SOM são usadas para problemas de classificação, extração de características, compressão de dados e formação de *clusters*. Em classificação de padrões, estes devem compartilhar características comuns para poderem ser agrupados, com cada grupo representando uma classe. Para realizar esse agrupamento, o algoritmo encontra características semelhantes nos dados de entrada, sem um supervisor externo, a partir da redundância dos mesmos. A ausência de redundância torna impossível encontrar padrões ou características nos padrões, ou seja, a redundância dos padrões de entrada fornece o conhecimento incorporado a rede.

Uma rede SOM funciona da seguinte forma: quando determinado padrão é exposto a entrada, a rede procura a unidade mais parecida com este padrão. Durante a fase de treinamento, como será visto a seguir, a rede aumenta a semelhança da unidade escolhida e de seus vizinhos próximos. Desta forma, a rede constrói um mapa topológico onde os neurônios que estão topologicamente próximos respondem de forma semelhante a padrões de entrada semelhante.

Quanto a arquitetura, as redes SOM se organizam em grades ou reticulados, geralmente bidimensional ou unidimensional, mas podendo também ser configuradas para ter $n$ dimensões. A Figura 3.3 mostra uma rede SOM típica, com um retículo de dimensões 3x3.

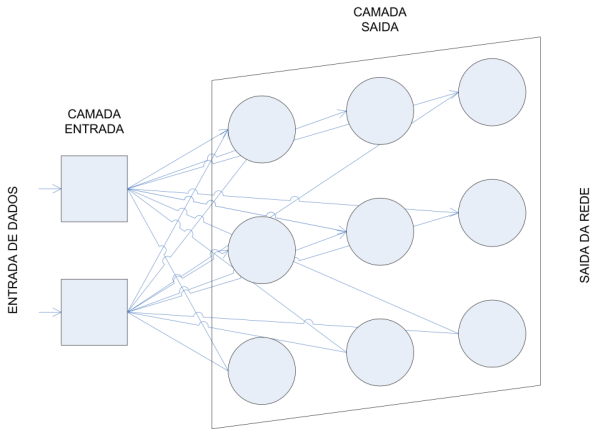


Figura 3.25: Rede SOM típica.

Como pode ser observado na Figura 3.3, todos os neurônios da rede recebem todas as entradas. A função de ativação de cada um é baseada no quadrado da distância euclidiana, expressa pela Equação 3.10.

$$y_j = \sum_{i=1}^{n} (x_i - w_{ij})^2 \tag{3.10}$$

onde $y_j$ é a saída do neurônio de índice $j$, $x_i$ é a $i$-ésima entrada da rede, e $w_{ij}$ é o peso que conecta a entrada $i$ ou neurônio $j$.

### 3.3.1 Projeto *Framework* de Redes SOM

Esta seção apresenta o que foi desenvolvido sobre redes auto-organizadas. A Figura 3.3.1 mostra o diagrama de classes criado para redes SOM.

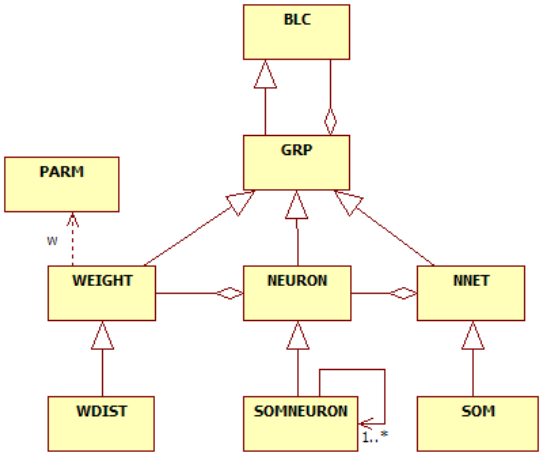


Figura 3.26: Diagrama de classes para construção de redes SOM.

A seguir é feita uma descrição detalhada sobre o que cada classe representa.

#### Classe WDIST

Classe derivada da classe *WEIGHT* na qual é usada para compor uma estrutura usada nas entradas de cada neurônio. Conta com uma entrada e uma saída. A Equação 3.11 mostra a operação matemática realizada por este bloco, onde a saída é o quadrado da diferença entre determinada entrada da rede e o peso associado.

$$y = (u - w)^2 \tag{3.11}$$

onde $y$ é saída do bloco, $w$ o próprio valor do peso, e $u$ a entrada do bloco.

A Figura 3.27 mostra o diagrama de blocos usando a estrutura **FASEE**.

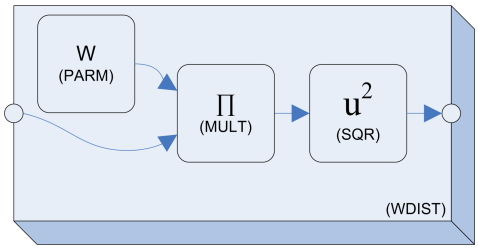


Figura 3.27: Diagrama de blocos da classe *WDIST*.

### Classe SOMNEURON

Classe derivada da classe *NEURON* na qual representa um neurônio em uma rede SOM. A Figura 3.28 mostra o diagrama de blocos de um neurônio *SOMNEURON* usando a estrutura de blocos do **FASEE**.

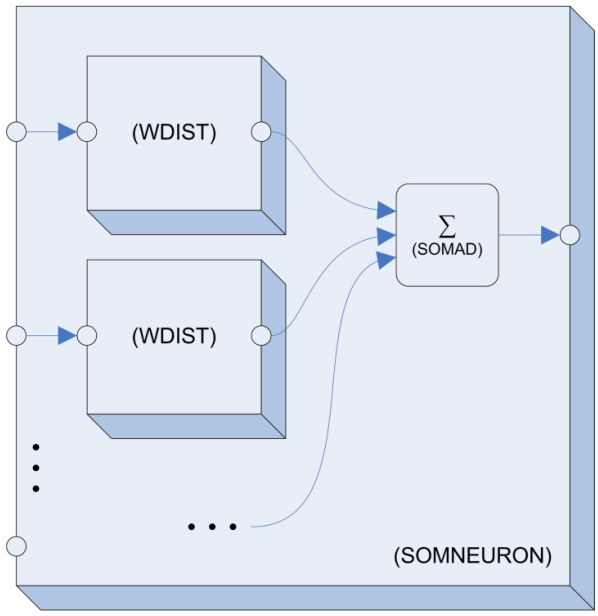


Figura 3.28: Diagrama de blocos da classe SOMNEURON usando a estrutura do **FASEE**.

A função que descreve seu comportamento é a mesma mostrada na Equação 3.10, aqui transcrita na Equação 3.12.

$$y_j = \sum_{i=1}^{n} (x_i - w_{ij})^2 \tag{3.12}$$

Diferentemente de um neurônio da rede MLP, classe *MLPNEURON*, o neurônio SOMNEURON não apresenta uma função de ativação.

### Classe SOM

Esta é a classe principal de redes SOM. Derivada de *NNET*, ela contém métodos usados para o gerenciamento dos neurônios da rede. A Figura 3.29 mostra o diagrama de blocos de uma rede SOM exemplo com quatro neurônios.

Deve-se salientar que os neurônios da Figura 3.29 não estão organizados em forma alguma, apenas todas as entradas estão conectadas a todos os neurônios. A arquitetura da rede é definida na etapa de treinamento, ou seja, é nesta fase que se define se os neurônios da rede formarão uma estrutura unidimensional, bidimensional, etc.

Abaixo é apresentado um trecho de código que exemplifica a criação de uma rede SOM.

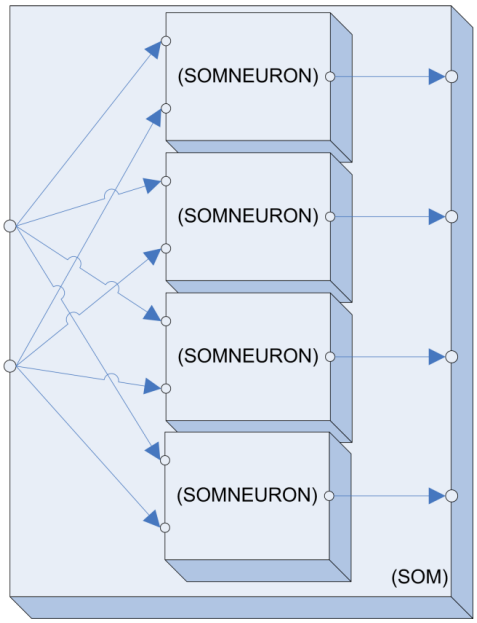


Figura 3.29: Diagrama de blocos da classe SOM usando a estrutura de blocos do **FASEE**.

```
1  #include "som.h"
2
3  void main()
4  {
5      // cria instancia da rede
6      SOM* som = new SOM( 2 /*entradas*/, 4 /*neuronios*/ );
7
8
9      // usa a rede...
10
11
12     // apaga instancia
13     delete som;
14 }
```

### 3.3.2 Treinamento de Redes SOM

O treinamento de redes tipo SOM é não-supervisionado e por competição. Existem essencialmente três processos que são chamados em seqüencia durante a etapa de treinamento [9]. São eles:

1. **Competição**. Dado um padrão de entrada da rede, os neurônios competem entre si para ver quem gera a maior saída. O neurônio que gerar a maior saída é o ganhador da competição.

2. **Cooperação**. O neurônio ganhador define a topologia espacial da vizinhança, ou seja, nesta etapa é que são definidos quais os neurônios próximos ao neurônio ganhador terão

seus pesos atualizados.

3. **Adaptação Sináptica**. Etapa que faz com que os neurônios ativos, ganhador e vizinhança, aumentem ainda mais a sua vantagem aos demais, atualizando seus pesos em relação a um determinado padrão de entrada.

A seguir é feita uma descrição mais detalhada de cada um destes processos.

### Processo de Competição

Para facilitar a compreensão primeiramente deve ser definido um vetor de entradas $\mathbf{x}$ de dimensão $m$ que denota uma padrão de entrada, definido pela expressão

$$\mathbf{x} = [x_1, x_2, ..., x_m] \tag{3.13}$$

Os pesos $\mathbf{w}$ de uma determinado neurônio $j$, com mesma dimensão $m$ da entrada, é definido pela expressão

$$\mathbf{w_j} = [w_{j1}, w_{j2}, ..., w_{jm}], \text{ onde } j = 1, 2, ...n \tag{3.14}$$

onde $n$ é o número total de neurônios na rede.

Para se achar o vetor de pesos sináptico $\mathbf{w_j}$ que seja mais próximo do vetor de entrada $\mathbf{x}$ basta comparar os produtos internos $\mathbf{w_j^T}.\mathbf{x}$ de cada neurônio, e selecionar o que tiver maior valor. Determinando-se o neurônio de maior valor de $\mathbf{w_j^T}.\mathbf{x}$ conseqüentemente determina-se o centro da vizinhança.

Para a etapa de treinamento, faz-se necessário maximizar o produto interno de $\mathbf{w_j^T}.\mathbf{x}$, o qual é matematicamente equivalente a minimizar a distância Euclidiana entre os vetores $\mathbf{x}$ e $\mathbf{w_j}$. Assim, para de definir o neurônio ganhador, adota-se a seguinte expressão:

$$i(\mathbf{x}) = \arg\min_j \|\mathbf{x} - \mathbf{w_j}\|, \text{ onde } j = 1, 2, ...n \tag{3.15}$$

sendo $i$ o índice do neurônio ganhador.

Dependendo da aplicação, a resposta de uma rede SOM pode ser tanto o índice do neurônio ganhador quanto o vetor de pesos do neurônio mais próximo(distância Euclidiana) do vetor de entrada.

### Processo de Cooperação

A idéia principal do processo de cooperação entre os neurônios é de determinar um conjunto de neurônios, próximos ao ganhador, que terão seus pesos sinápticos atualizados no processo seguinte. Os pesos destes "vizinhos" serão atualizados de forma proporcional a suas

distâncias do ganhador, ou seja, quanto mais distantes do ganhador, menor será sua atualização de pesos. Neurobiologicamente, existe interação lateral entre todos os neurônios com seus adjacentes.

A definição do neurônio vencedor determina o centro topológico da vizinhança. Existem diversos tipos de topologias, as quais definem as conexões de cada neurônio em relação aos seus adjacentes. As Figuras 3.30 e 3.31 ilustram o formato de vizinhanças mais freqüentemente encontradas: a quadrada(*grid*) e a hexagonal.

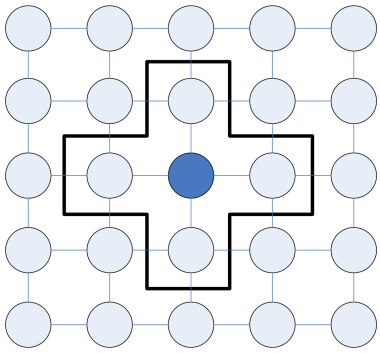

Figura 3.30: Vizinhança quadrada.

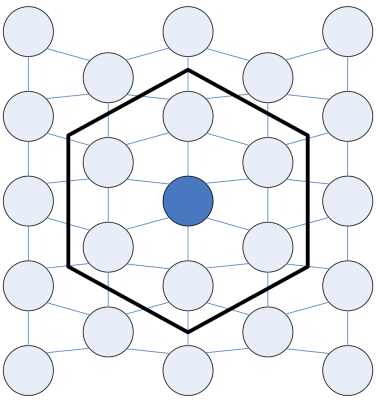

Figura 3.31: Vizinhança hexagonal.

Vale salientar que a topologia da vizinhança somente é definida durante a etapa de treinamento da rede.

A taxa na qual cada peso da vizinhança será atualizado segue a forma de uma Gaussiana. A forma Gaussiana é mais adequada que uma vizinhança retangular, visto que os neurônios adjacentes ao ganhador são atualizados numa proporção significativamente maior que os não adjacentes, levando a um aumento da velocidade de convergência do algoritmo de treinamento.

A Equação 3.16 define a distância lateral entre os neurônios, onde $i$ é o índice do neurônio ganhador, $j$ o índice de um neurônio vizinho, e $d_{ij}$ é a distância lateral entre o neurônio vencedor $i$ em relação a um determinado neurônio $j$, definida pela distância Euclidiana entre eles. Os termos $\underline{r}_i$ e $\underline{r}_j$ representam respectivamente a posição do neurônio ganhador e de uma determinado vizinho no retículo.

$$d_{ij}^2 = \|\underline{r}_i - \underline{r}_j\|^2 \tag{3.16}$$

A Equação 3.17 mostra a função da vizinhança, usada para determinar a proporção na qual os neurônios vizinhos ao ganhador serão atualizados.

$$h_{ij}(n) = \exp\left(-\frac{-d_{ij}^2}{2.\sigma(n)^2}\right) \tag{3.17}$$

O parâmetro $\sigma$ determina a largura efetiva da vizinhança. Como este parâmetro varia com

o tempo ele diminui a largura da vizinhança a medida que o número de épocas aumenta. A razão na qual o desvio padrão da Gaussiana da vizinhança diminui com o tempo é exponencial, expresso na Equação 3.18. A Figura 3.3.2 mostra a forma de três gráficos da função Gaussiana com diferentes $\sigma$.

$$\sigma(n) = \sigma_0 . \exp\left(-\frac{n}{\tau_1}\right) \tag{3.18}$$

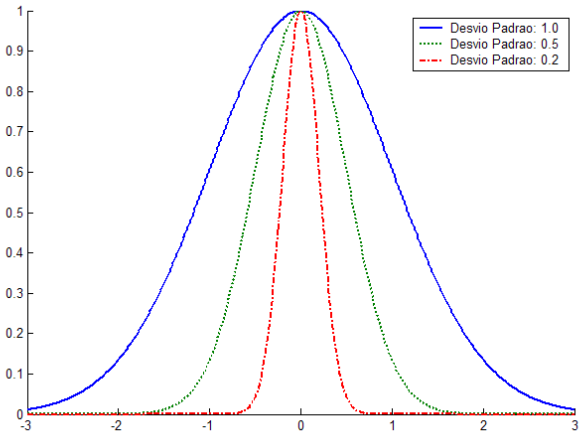


Figura 3.32: Funções Gaussianas com $\sigma = 1$, $\sigma = 0.5$ e $\sigma = 0.2$.

A constante de tempo $\tau_1$ da Equação 3.18 indica a velocidade na qual o desvio padrão decai com o tempo.

Neste trabalho, além de se poder usar as topologias pré-definidas(quadrada, hexagonal e circular), foi desenvolvida uma forma de se definir a topologia de uma rede SOM a partir de um arquivo onde, para cada linha, é descrita a posição de cada neurônio em um espaço de $n$ dimensões, definido pelo usuário. O único critério que deve ser obedecido ao se definir uma topologia é que neurônios adjacentes devem ter uma distância euclidiana igual ou muito próxima de 1.

### Processo de Adaptação Sináptica

O último processo envolvido na formação de um mapa auto-organizado de características é o processo de adaptação dos pesos sinápticos. Para que a rede possa se auto-organizar, o vetor de pesos sinápticos $\underline{w}_i$, do neurônio ganhador, deve variar de forma que seus valores "caminhem" em direção do vetor de entradas $\underline{x}$. A expressão que descreve a atualização dos pesos na forma discreta é mostrada na Equação 3.19 [9].

$$\underline{w}_j(n+1) = \underline{w}_j(n) + \eta(n).h_{ji}(n).(\underline{x} - \underline{w}_j(n)) \tag{3.19}$$

Na Equação 3.19, o termo $h_{ij}(n)$ refere-se a função vizinhança, a qual determina a taxa na qual determinado neurônio $j$ será atualizado conforme sua distância do neurônio ganhador $i$, descrito no processo anterior.

O termo $\eta(n)$ define a razão de aprendizado, na qual varia em função do número de épocas. Ela deve iniciar com um valor de $\eta_0$ e decrescer gradualmente com o aumento do número de épocas. A Equação 3.20 define a expressão da taxa de aprendizado [9].

$$\eta(n) = \eta_0 . \exp(-\frac{n}{\tau_2}) \tag{3.20}$$

A constante de tempo $\tau_2$ define a velocidade de decaimento da taxa de aprendizado em função do número de épocas.

Ainda dentro do processo de adaptação sináptica, pode-se dividir este em dois sub-processos: ordenação e convergência. Neste primeiro ocorre a ordenação topológica dos vetores sinápticos. Ela foi empiricamente definida como sendo 10% do número total de épocas. Já no segundo sub-processo, a convergência ocorre um refinamento do mapa de características, onde a taxa de aprendizado é pequena e o tamanho da vizinhança que será significantemente atualizada não passa de dois neurônios adjacentes ao ganhador.

Para o treinamento de redes SOM foi necessário criar uma classe chamada *SOMTRAIN*, conforme mostra o diagrama da Figura 3.33.

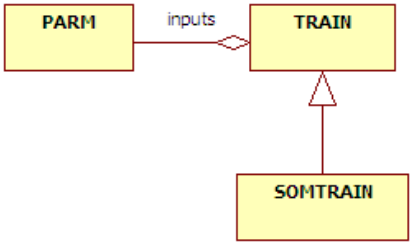


Figura 3.33: Diagrama de classes para treinamento de redes SOM.

A classe *SOMTRAIN* engloba todos os conceitos expostos nesta seção. Esta classe deve ser utilizada para treinar a rede cuja a organização dos neurônios se dê em uma ou duas dimensões, com vizinhança quadrada ou hexagonal.

O código abaixo mostra um exemplo de treinamento de uma rede SOM com duas entradas em um retículo de 10x10 com vizinhança hexagonal.

```
1 #include "som_train.h"
2
3 void main()
4 {
5     // cria instancia da rede
```

```
6     SOM* som = new SOM( 2 /*entradas*/, 100 /*neuronios*/ );
7
8     // dados de treinamento...
9     DATAMAT inputs(...
10
11
12    // cria instancia da classe de treinamento da rede
13    SOMTRAIN st = new SOMTRAIN(som);
14
15    // define o numero de epocas
16    st->maxepoch( 10000 );
17
18    // define a estrutura da rede: dimensoes e topologia
19    st->setup( 10 /*num. linhas*/, 10 /*num. colunas*/, HEXTOP /*topologia*/ );
20
21    // treina a rede
22    int ret = st->train( inputs /*entradas*/, NULL /*callback*/ );
23
24    // apaga instancia do treinamento
25    delete st;
26
27
28    // usa a rede treinada ...
29
30
31    // apaga instancia da rede
32    delete som;
33 }
```

A seguir é feita uma descrição detalhada dos pontos mais importantes do que foi feito no código acima.

- Cria instância da rede...

  ```
  SOM* som = new SOM( 2 /*entradas*/, 100 /*neuronios*/ );
  ```

  Primeiramente, foi criada uma instância da rede classe *SOM*, passando por parâmetro para o construtor parametrizado o número de entradas da rede, seguido do número de neurônios.

- Dados de treinamento...

  ```
  DATAMAT inputs(...
  ```

  Cria uma instâncias da classe *DATAMAT* chamadas *inputs* que representando as entradas (amostras) da rede. Estas informações podem vir de qualquer fonte, sendo a classe *DATAMAT* usada como uma forma de padronização. Vale salientar que como o treinamento de redes SOM é não-supervisionado, não há a necessidade de dados de saídas desejada da rede.

- Cria instância do algoritmo de treinamento.

```
SOMTRAIN st = new SOMTRAIN( som /*instancia da rede*/ );
```

Cria uma instância da *SOMTRAIN*, passando como parâmetro para o construtor parametrizado a rede *som*. Neste momento são criadas as instâncias da classe *PARMs*, usadas para representar os dados de entrada, assim como inicializar os parâmetros de treinamento com valores padrão.

- Definição do numero de épocas.

```
st->maxepoch( 10000 );
```

Neste ponto é definido o número máximo de épocas que o processo de treinamento irá realizar. Este valor deve ser cuidadosamente escolhido para cada problema em questão, visto que ele irá definir o número de épocas em cada sub-processo do processo de adaptação sináptica.

- Definição da estrutura da rede: dimensões e topologia.

```
st->setup( 10 /*num. linhas*/, 10 /*num. colunas*/, HEXTOP /*topologia*/ );
```

É neste ponto é definida a estrutura da rede, no que diz respeito a suas dimensões e ligações entre os neurônios. O primeiro e segundo parâmetros, referem-se, respectivamente, ao número de linhas e colunas do retículo formado pelos neurônios da rede. Já o terceiro parâmetro determina a topologia da rede, que pode ser quadrada(*GRIDTOP*) ou hexagonal(*HEXTOP*), como demonstrado nas Figuras 3.30 e 3.31.

- Treina a rede.

```
int ret = st->train( inputs /*entradas*/, NULL /*callback*/ );
```

Para o treinamento da rede ser efetivamente realizado o método *train* deve ser chamado, passando-se por parâmetro os dados de entradas da rede. Assim como no treinamento de redes MLP, este método pode também receber um segundo parâmetro, o qual é uma referência para uma função que será chamada a cada iteração do algoritmo.

### 3.3.3 Exemplo Completo

Nesta seção será mostrado um exemplo completo do uso de uma rede SOM para resolver o problema clássico de divisão de um determinado grupo de dados em classes distintas, conhecido do termo em inglês *clustering*. Uma rede SOM será criada com duas entradas que irão receber as coordenadas $(x, y)$ de cada amostra durante o treinamento. Neste exemplo, sabe-se *a priori* que o número de agrupamentos existentes nos dados de treinamento é quatro, portanto a rede será configurada para ter uma topologia que contém quatro neurônios conectados em um *grid* de $2x2$. Os dados de treinamento contém 40 amostras. A listagem completa do programa, escrito em **Lua**, é mostrada abaixo.

```
-- carrega framework de RNA (arquivo 'luafann.dll')
require 'luafann'

-- cria matriz de dados de entradas
inp = datamat()
inp:load('clusters4_samples40.txt')

-- cria rede SOM
s = som()

-- define estrutura da rede: 2 entrada, 4 neurônios, topologia GRID(2x2)
s:setup( 2, 4, TOPOLOGY.GRID2D, 2, 2 )

s:param('maxepoch',100)

-- define taxa de aprendizado inicial(epoch=0)
s:param('etamax',1.0)

-- define taxa de aprendizado final(epoch=maxepoch)
s:param('etamin',0.01)

-- define desvio padrão inicial da gaussiana da vizinhança(epoch=0)
s:param('stdmax',0.1)

-- define desvio padrão final da gaussiana da vizinhança(epoch=0)
s:param('stdmin',0.01)

-- define funcao de callback
function callback( msg )
  print( string.format("Epoch: %d \t Error: %.7f", msg.epoch, msg.error ) )
  return 0
end

-- realiza o treinamento da rede
s:train( inp, "callback" )

ns = s:neurons()

for _,n in ipairs(ns) do
  print( string.format('Neuronio:%s(w1,w2)=(x,y)=(%.3f,%.3f)',n.name,n[1],n[2]) )
end
```

Ao executar o programa a seguinte saída é exibida na tela:

```
Epoch: 1          Error: 211.3199942
Epoch: 2          Error: 2.5455119
Epoch: 3          Error: 2.2802266
Epoch: 4          Error: 2.1125927
Epoch: 5          Error: 1.9283644
Epoch: 6          Error: 1.7681299
...
Epoch: 96         Error: 0.0000496
Epoch: 97         Error: 0.0000407
Epoch: 98         Error: 0.0000333
Epoch: 99         Error: 0.0000273
```

```
Neuronio: N1_NNET (w1,w2)=(x,y)=(0.454,0.998)
Neuronio: N2_NNET (w1,w2)=(x,y)=(0.621,0.238)
Neuronio: N3_NNET (w1,w2)=(x,y)=(0.449,0.449)
Neuronio: N4_NNET (w1,w2)=(x,y)=(0.162,0.388)
```

Deve-se perceber que, para este exemplo, a saída da rede é o valor dos pesos sinápticos de cada neurônio conectado as entradas da rede, ou seja, cada neurônio apresenta duas ligações com a camada de entrada, cada uma com um peso associado, formando assim um par ordenado $(x, y)$ que indica, neste caso, a posição do neurônio em um espaço de duas dimensões. A Figura 3.34 mostra tanto os dados de treinamento quanto a saída da rede.

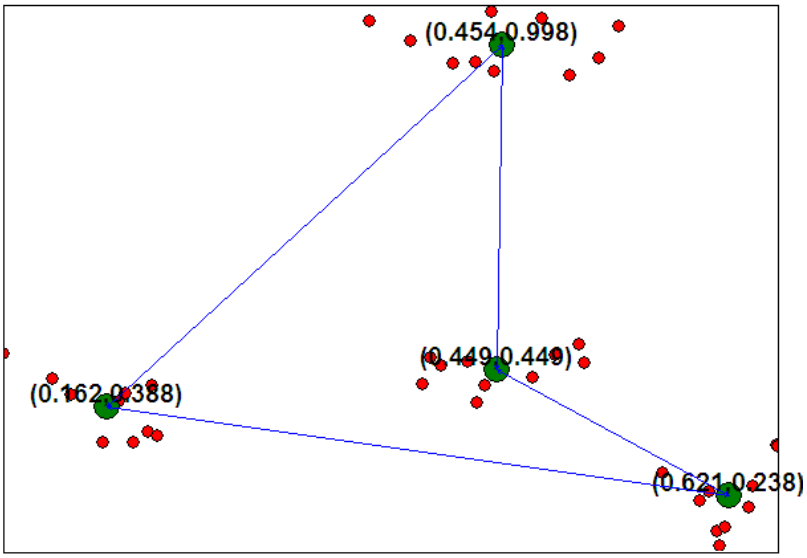


Figura 3.34: Saída do treinamento de rede SOM: neurônios posicionados sobre os agrupamentos.

Outras duas simulações foram realizadas com um conjunto de 1000 amostras distribuídas aleatoriamente, com o intuito de mostrar a interação entre os neurônios conforme sua topologia. Neste caso, criou-se uma rede com 100 neurônios distribuídos tanto em um *grid* quanto numa topologia hexagonal de $10x10$neurônios.

Visualizando-se a Figura 3.35 se pode concluir que as redes se comportaram de forma esperada, onde os retículos se estenderam por praticamente toda a área onde há amostras, e que cada neurônio respeitou de forma adequada as ligações entre seus adjacentes, os vizinhos.

### 3.3.4 Considerações

Em diversas implementações de redes tipo SOM, a exemplo onde ocorre em [26], observa-se que a função vizinhança é simplificada de forma a facilitar a sua codificação, onde ao invés de se utilizar uma função Gaussiana para descrever a taxa na qual cada neurônio vizinho do

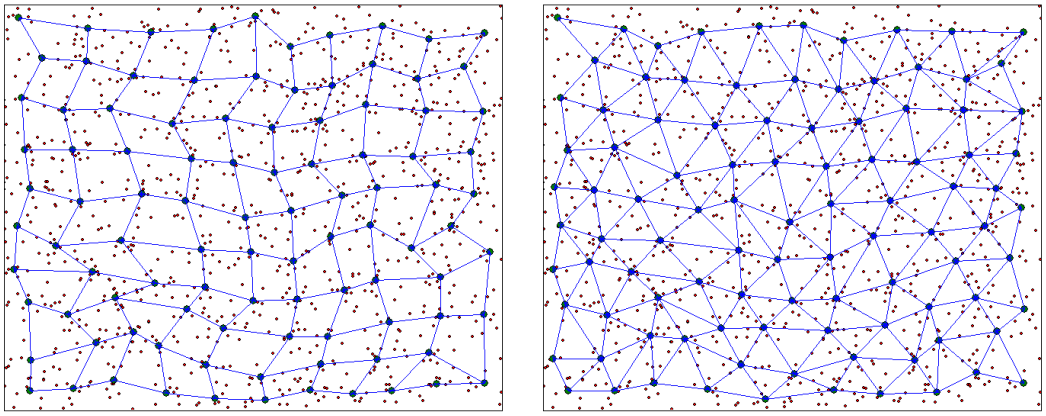

Figura 3.35: 1000 amostras aleatórias e 100 neurônios: topologias GRID e HEX.

ganhador deverá ser atualizada, usa-se uma função *passo* para se determinar um conjuntos de neurônios que serão atualizados, indiferente da suas distâncias relativas ao ganhador. Embora em ambas as versões a função vizinhança decaia com o tempo, a primeira torna o sub-processo de convergência mais rápido, visto que retrata de forma mais realista a iteração entre os neurônios.

Uma desvantagem do treinamento de rede SOM é que ocorrem distorções nas bordas do retículo, pelo fato de uma rede com topologia quadrada(*GRIDTOP*) os neurônios internos tem mais vizinhos que os das bordas. Outro problema associado a utilização de redes SOM é o seu grande número de parâmetros a serem definidos até que se consiga um treinamento satisfatório. Entre eles, pode-se citar: a maneira na qual a vizinhança muda de tamanho, razão de treinamento, e número de iterações total.

# Capítulo 4

# Lógica Difusa

Neste capítulo será inicialmente realizada uma pequena descrição sobre lógica *fuzzy*. A seguir é feita a descrição do *framework* de lógica *fuzzy* que foi desenvolvido.

## 4.1 Introdução

A lógica difusa ou lógica *fuzzy* é uma generalização da lógica booleana que admite valores lógicos intermediários entre verdadeiro ou falso. Como existem várias formas de se implementar um modelo *fuzzy*, a lógica *fuzzy* deve ser vista mais como uma área de pesquisa sobre tratamento da incerteza, ou uma família de modelos matemáticos dedicados ao tratamento da incerteza, do que uma lógica propriamente dita [4]. A lógica difusa normalmente está associada ao uso de uma teoria de conjuntos *fuzzy*. As implementações da lógica difusa permitem que estados indeterminados possam ser tratados por dispositivos de controle, como por exemplo avaliar conceitos como "morno", "médio", etc.

Normalmente, o uso da lógica difusa está associado ao uso de conjuntos nebulosos. Um conjunto nebuloso estende o conceito de conjunto, permitindo que um elemento passa a ter um grau de pertinência variando entre 0 e 1, ao invés de pertencer ou não ao conjunto como na teoria de conjuntos tradicional. Para cada conjunto, então, é criada uma função de pertinência, que indica o grau de pertinência de seus elementos. Normalmente, essa função é criada de forma a representar algum conceito impreciso, como "ser alto".

### 4.1.1 Tempo de Projeto

Para a criação de um sistema *fuzzy* devem ser realizados os seguintes passos:

**Configuração das entradas** Para cada entrada definir as variáveis lingüísticas e suas funções de pertinência.

**Configuração das saídas** Para cada saída definir as variáveis lingüísticas e suas funções de pertinência.

**Definição das regras** Para cada regra definir as suas premissas e implicações.

### 4.1.2 Tempo de Execução

Uma vez criado o sistema *fuzzy*, pode-se executá-lo através de um motor de inferência *fuzzy*. O raciocínio *fuzzy*, também é conhecido como raciocínio aproximado, e pode ser dividido em 5 etapas.

1. Transformação das variáveis do problema em valores *fuzzy*. Processo chamado de *fuzzification*, onde, para cada entrada, determina-se o grau de pertinência de cada conjunto. Este valor é limitado entre 0 e 1.

2. Aplicação dos operadores *fuzzy*. Os operadores usados na lógica *fuzzy* são AND e OR, conhecidos como operadores de relação. Na lógica *fuzzy* são utilizados para definir o grau máximo e mínimo de pertinência de um conjunto.

3. Aplicação da implicação. O terceiro passo é aplicar o operador de implicação, usado para definir o peso no resultado e remodelar a função, ou seja, consiste em criar a hipótese de implicação.

4. Combinação de todas as saídas *fuzzy* possíveis. Neste passo ocorre a combinação de todas as saídas em um único conjunto *fuzzy*, chamado de *fuzzyset*.

5. Transformação do resultado *fuzzy* em um resultado nítido. Processo chamado de *defuzzificação*, ou *defuzzification*, que consiste em encontrar valores escalares, dentro da faixa estipulada pela lógica *fuzzy* para uma saída, a partir de um critério definido durante a criação do sistema.

## 4.2 Projeto *Framework* Lógica *Fuzzy*

Neste trabalho, inicialmente, o desenvolvimento do *framework* de lógica difusa era baseado no *framework* **FASEE**. Porém, devido a algumas limitações deste, o foco mudou para o desenvolvimento voltado a linguagem de programação **Lua**. Nesta seção será mostrado primeiramente até que ponto foi desenvolvido o *framework* de lógica difusa sobre o *framework* **FASEE**, identificando as limitações que levaram a mudança de foco. Na seção seguinte é descrito em detalhes o *framework* de lógica difusa desenvolvido em **Lua**, juntamente com um código exemplo que demonstra as suas funcionalidades.

### 4.2.1 FASEE

O desenvolvimento do *framework* de lógica difusa no **FASEE** em digrama de blocos foi realizado de forma conceitual, utilizando digramas de classe da linguagem UML para modelagem

das classes e suas ligações e dependências. A idéia era criar um ambiente que contemplasse todos os conceitos envolvidos em uma máquina de inferência *fuzzy*, flexível e genérica o suficiente, possibilitando a criação de operações *fuzzy* (ex.: funções de pertinência, métodos de agregação, implicação, etc.) definidas pelo usuário, através da modelagem destas em diagramas de blocos no **FASEE**. Após uma quantidade significativa de tentativas de adaptação, o diagrama de classes apresentado na Figura 4.1 foi obtido.

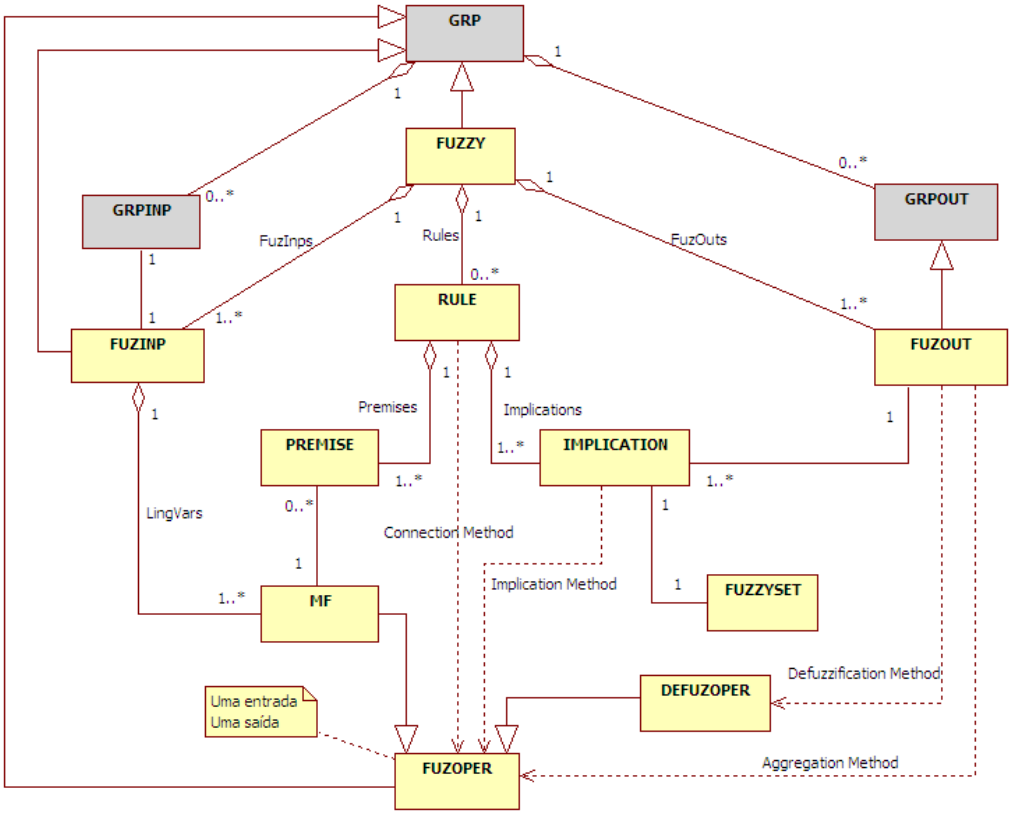


Figura 4.1: Diagrama de classes do framework de lógica *fuzzy*.

A seguir é feita uma descrição de cada classe encontrada no diagrama da Figura 4.1.

### Classe FUZOPER

Classe base para qualquer operação dentro da lógica difusa. Dela são especializadas classes como *FUZTMIN* e *FUZTSUM*, que representam respectivamente os operadores de norma mínima ($T_{min}$) e soma ($T_{sum}$).

**Classe MF**

Classe que representa um função de pertinência(*membership function*) dentro da lógica difusa.

**Classe DEFUZOPER**

Classe base para operações de *defuzzificação*.

**Classe FUZZYSET**

Classe que representa um conjunto *fuzzy*, usado como saída das operações de implicações e agregação.

**Classe FUZINP**

Classe derivada da classe *GRPINP* que representa entradas de um sistema *fuzzy*. Essa classe contém uma lista de instâncias da classe *MF*, as quais representam as variáveis lingüísticas de uma determinada entrada.

**Classe FUZOUT**

Classe derivada da classe *GRPOUT* que representa saídas de um sistema *fuzzy*. Essa classe contém uma lista de instâncias da classe *IMPLICATION*.

**Classe IMPLICATION**

Classe que representa uma implicação de uma determinada regra. Ela contém uma instância da classe *FUZZYSET* que é o resultado da aplicação da operação de implicação.

**Classe PREMISE**

Classe que representa uma premissa de uma determinada regra dentro de um sistema *fuzzy*.

**Classe RULE**

Classe que representa uma regra dentro de um sistema *fuzzy*, englobando um conjunto de premissas(*PREMISE*) e implicações(*IMPLICATION*).

**Classe FUZZY**

Classe, derivada da *GRP*, que define um sistema *fuzzy*, com um lista de regras(*RULE*), entradas(*FUZINP*) e saídas(*FUZOUT*).

Embora a estrutura de blocos do **FASEE** tenha servido de forma plena para desenvolvimento do *framework* de redes neurais artificiais, não serviu de forma satisfatório para o de lógica difusa. O principal motivo foi que, entre as etapas de implicação das regras e a de *defuzzyficação* das saídas, a estrutura de blocos do **FASEE** teria que trafegar entre suas conexões dados *fuzzy*(*fuzzysets*). A operação de trafegar vetores de dados poderia ser conseguida da multiplexação/demultiplexação de entradas e saídas de blocos, porém esta funcionalidade não encontra-se implementada na estrutura de blocos do **FASEE**. Em [7] isso pode ser observado, onde é desenvolvido uma biblioteca para criação de sistemas *fuzzy* usando como base o Simulink, e que a solução encontrada para trafegar *fuzzysets* entre os blocos foi justamente o uso da multiplexação/demultiplexação de dados.

### 4.2.2 Lua Fuzzy

A Figura 4.2 mostra como o *framework* de lógica *fuzzy* foi estruturado usando os recursos disponíveis do ambiente **Lua**.

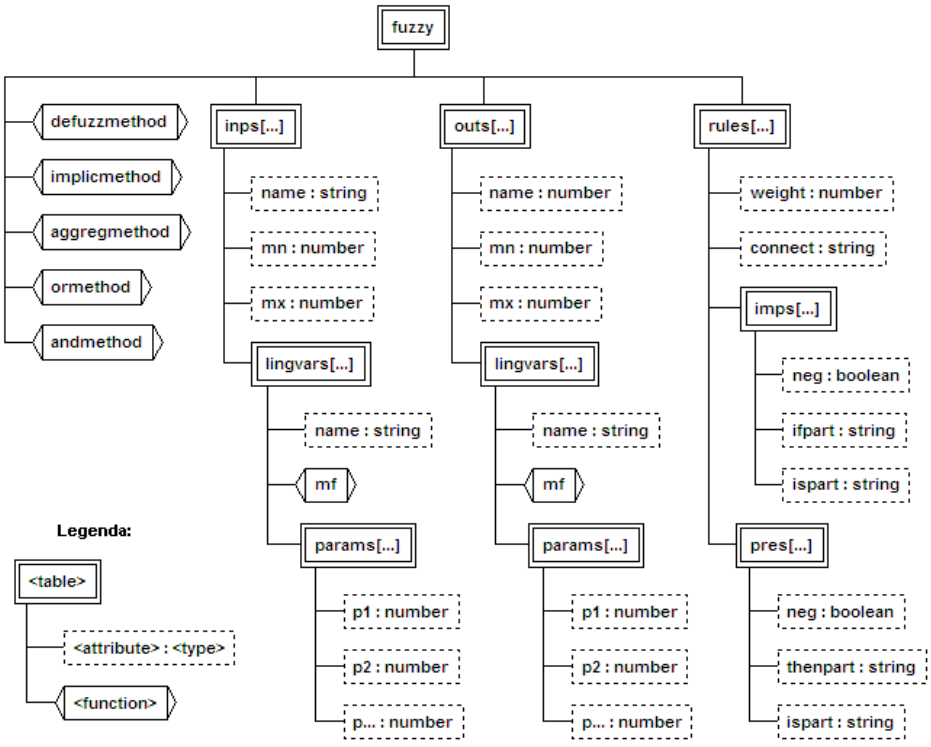


Figura 4.2: Diagrama da estrutura de lógica *fuzzy*.

Abaixo será apresentada uma tabela para cada *table* **Lua** apresenta na Figura 4.2, onde é descrito o significado e o propósito de campo.

| *table fuzzy* | | |
|---|---|---|
| Campo | Descrição | Tipo |
| andmethod | Função que representa o modo como as operações $E(AND)$ são realizadas. Valores comuns para este campo são funções que implementam as normas $T_{min}$(função *tmin*) e $T_{prod}$(função *tprod*). | function |
| ormethod | Função que representa o modo como as operações $OU(OR)$ são realizadas. Valores comuns para este campo são funções que implementam as normas $T_{max}$(função *tmax*) e $T_{sum}$(função *tsum*). | function |
| implicmethod | Função que representa o modo como as operações implicação das premissas das regras são realizadas. Valores comuns para este campo são funções que implementam as normas $T_{min}$(função *tmin*) e $T_{prod}$(função *tprod*). | function |
| aggregmethod | Função que representa o modo como as operações de agregação de implicações para um determinada saída são realizadas. Valores comuns para este campo são funções que implementam as normas $T_{max}$(função *tmax*) e $T_{sum}$(função *tsum*). | function |
| defuzzmethod | Função que representa o modo como os *fuzzysets* das saídas são defuzzificados. Um valor comum para este campo é a função para o cálculo do centro de massa de *fuzzysets*(função *centroid*). | function |
| inps | Campo que é uma *table* que representa as entradas na máquina de inferência *fuzzy*. | table |
| outs | Campo *table* que representa as saídas na máquina de inferência *fuzzy*. | table |
| rules | Campo *table* que representa as regras do sistema *fuzzy*. | table |

Abaixo é apresentada uma tabela com a descrição dos campos das *tables* de entradas(*inps*).

| *table fuzzy.inps[...]* | | |
|---|---|---|
| Campo | Descrição | Tipo |
| name | Nome da entrada. | string |
| mn | Limite inferior dos dados de entrada. | number |
| mx | Limite superior dos dados de entrada. | number |
| lingvars | Lista de variáveis lingüística. | table |

Abaixo é apresentada uma tabela com a descrição dos campos das *table* de saídas(*outs*).

| *table fuzzy.outs[...]* | | |
|---|---|---|
| Campo | Descrição | Tipo |
| name | Nome da saída. | string |
| mn | Limite inferior dos dados de saída. | number |
| mx | Limite superior dos dados de saída. | number |
| lingvars | Lista de variáveis lingüística. | table |

Abaixo é apresentada uma tabela com a descrição dos campos das *tables* de variáveis lingüística(*lingvars* associadas a entradas e saídas.

| *table fuzzy.inps[...].lingvars e fuzzy.outs[...].lingvars* | | |
|---|---|---|
| Campo | Descrição | Tipo |
| mf | Funções de pertinência associada a variável lingüística. | function |
| params | Vetor de parâmetros das funções de pertinência associadas as variáveis lingüísticas.(ex.: params = {0.5, 1.0, 1.5}) | table |

Abaixo é apresentada uma tabela com a descrição dos campos das *table* de regras(*rules*) de um sistema *fuzzy*.

| *table fuzzy.rules[...]* | | |
|---|---|---|
| Campo | Descrição | Tipo |
| weight | Usado na operação de implicação para ponderar cada regra conforme o valor do peso. | number |
| connect | Função usada para a conexão entre as premissas de uma regra. Pode assumir somente os valores 'andmethod' ou 'ormethod'. | string |
| pres | Lista de premissas. | table |
| imps | Lista de implicações. | table |

Abaixo é apresentada uma tabela com a descrição dos campos das *tables* de premissas(*pres*) de uma regra.

| *table fuzzy.rules[...].pres[...]* | | |
|---|---|---|
| Campo | Descrição | Tipo |
| neg | Usado para indicar a máquina de inferência *fuzzy* para negar o resultado da premissa, ou seja, usar o complemento do valor de saída. | number |
| ifpart | Nome da variável de entrada. | string |
| ispart | Nome da variável lingüística associada a variável de entrada. | string |

Abaixo é apresentada uma tabela com a descrição dos campos das *tables* de implicações(*imps*) de uma regra.

| *table fuzzy.rules[...].imps[...]* | | |
|---|---|---|
| Campo | Descrição | Tipo |
| neg | Usado para indicar a máquina de inferência *fuzzy* para negar o resultado da implicação, ou seja, usar o complemento do valor de saída. | number |
| thenpart | Nome da variável de saída. | string |
| ispart | Nome da variável lingüística associada a variável de saída. | string |

Embora a linguagem **Lua** não seja orientada a objetos, ela apresenta alguns mecanismos que possibilitam adicionar algumas características de POO, como definição classes, que são as próprias *tables*, e de métodos. Assim, a Figura 4.3, mostra um diagrama de classes que representa, de forma conceitual, o *framework* de lógica *fuzzy*. Todas as funcionalidades do *framework* de lógica *fuzzy* feito em **Lua** poderão também ser utilizadas no *framework* de modelos do **FASEE**, um vez que um bloco chamado *LUABLC* foi criado para realizar a comunicação entre os dois *frameworks*. Detalhes sobre o bloco *LUABLC* serão discutidas no final deste capítulo.

A idéia mostrada no diagrama de Figura 4.3 é salientar os pontos flexíveis da estrutura, que são as interfaces de *defuzzificação*(*defuzz*), de operadores *fuzzy*(*fuzzoper*), e de funções de pertinência(*mf*).

O código abaixo mostra um exemplo clássico de uso do *framework* de lógica *fuzzy*. O objetivo é determinar a gorjeta a ser dada para um garçom em um restaurante, calculada sobre um percentual da conta, a partir de dois critérios de entrada: a qualidade do serviço e da comida, com ambos variando entre 0 e 10.

```
1  -- carrega o framework de logica fuzzy
2  require 'luafuzzy'
3
4  -- cria um novo sistema
5  fuzzy = luafuzzy()
6
7  -- configura a entrada 'service'
8  serv = fuzzy:addinp( 'service', 0, 10 )
9  serv:addlingvar( 'poor', gaussmf, { 1.5, 0. } )
10 serv:addlingvar( 'good', gaussmf, { 1.5, 5. } )
11 serv:addlingvar( 'excellent', gaussmf, { 1.5, 10. } )
12
13 -- configura a entrada 'food'
14 food = fuzzy:addinp( 'food', 0, 10 )
15 food:addlingvar( 'rancid', trapmf, { 0, 0, 1, 3 } )
16 food:addlingvar( 'delicious', trapmf, { 7, 9, 10, 10 } )
17
18 -- configura a saída 'tip'
```

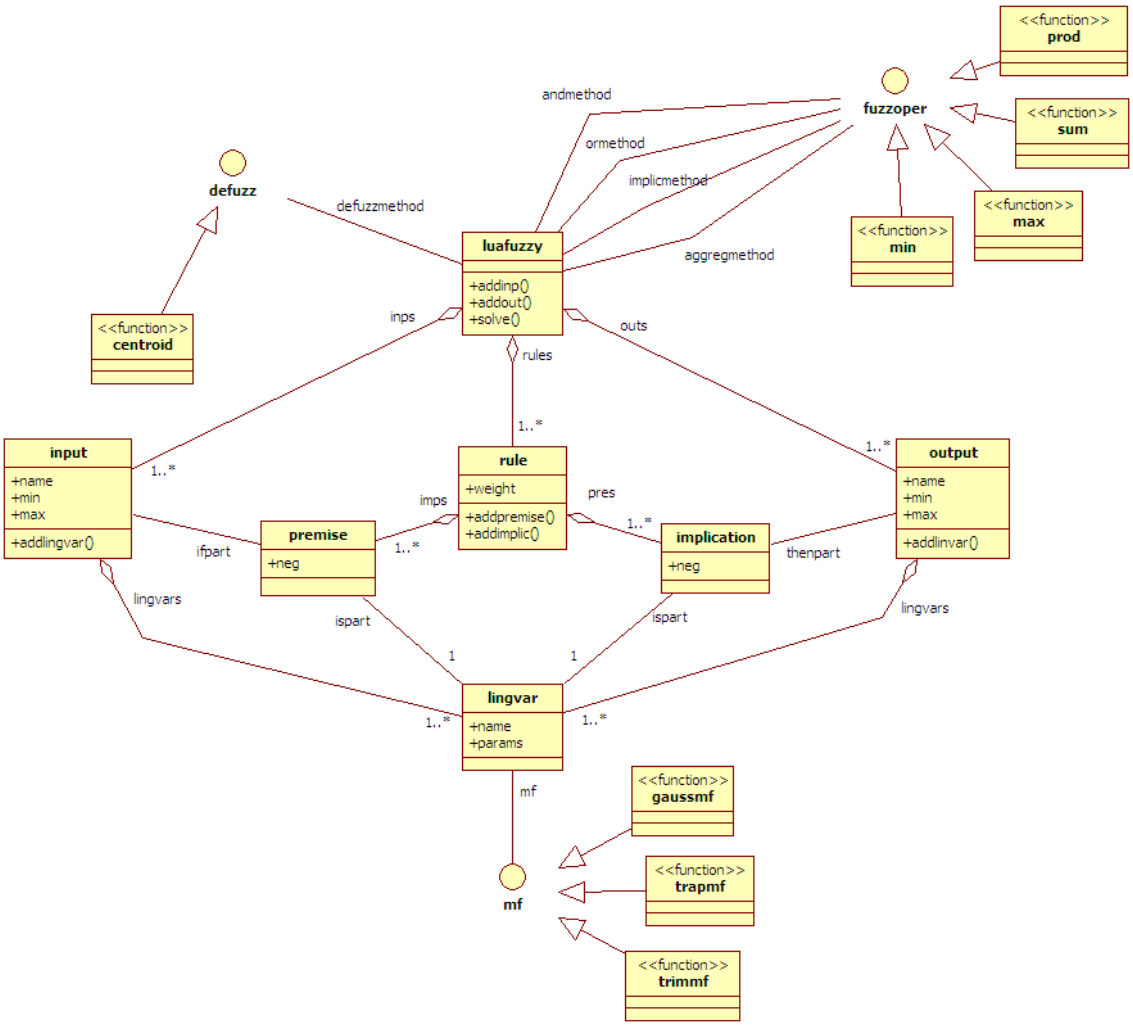


Figura 4.3: Diagrama de classes do framework de lógica *fuzzy* desenvolvido em **Lua**.

```
19 tip = fuzzy:addout( 'tip', 0, 30 )
20 tip:addlingvar( 'cheap', trimf, { 0, 5, 10 } )
21 tip:addlingvar( 'average', trimf, { 10, 15, 20 } )
22 tip:addlingvar( 'generous', trimf, { 20, 25, 30 } )
23
24
25 -- define a regra 1
26 r1 = fuzzy:addrule( 1, 'ormethod' )
27 r1:addpremise( false, 'service', 'poor' )
28 r1:addpremise( false, 'food', 'rancid' )
29 r1:addimplic( false, 'tip', 'cheap' )
30
31 -- define a regra 2
32 r2 = fuzzy:addrule( 1, 'andmethod' )
33 r2:addpremise( false, 'service', 'good' )
34 r2:addimplic( false, 'tip', 'average' )
```

```
35
36 -- define a regra 3
37 r3 = fuzzy:addrule( 1, 'ormethod' )
38 r3:addpremise( false, 'service', 'excellent' )
39 r3:addpremise( false, 'food', 'delicious' )
40 r3:addimplic( false, 'tip', 'generous' )
41
42 -- testa o sistema
43 outs = fuzzy:solve( 8, 6.5 )
44 print( outs )
```

A seguir é feita uma análise detalhada dos pontos mais importantes do que foi feito no código acima.

- Carga do framework de lógica *fuzzy*.

```
require 'luafuzzy'
```

  Essa operação faz com que sejam carregadas as funções do framework de lógica *fuzzy* contidas no arquivo *script* **luafuzzy.lua**.

- Cria novo sistema *fuzzy*.

```
fuzzy = luafuzzy()
```

  Essa operação atribuí a variável *fuzzy* uma *table* com as configurações iniciais de um sistema *fuzzy*.

- Configura as entradas 'service' e 'food'.

```
serv = fuzzy:addinp( 'service', 0, 10 )
serv:addlingvar( 'poor', gaussmf, { 1.5, 0. } )
serv:addlingvar( 'good', gaussmf, { 1.5, 5. } )
serv:addlingvar( 'excellent', gaussmf, { 1.5, 10. } )

-- configura a entrada 'food'
food = fuzzy:addinp( 'food', 0, 10 )
food:addlingvar( 'rancid', trapmf, { 0, 0, 1, 3 } )
food:addlingvar( 'delicious', trapmf, { 7, 9, 10, 10 } )
```

  As variáveis *serv* e *food* são atribuídas à uma *table* contendo as suas configurações como nome e faixa de valores de entrada. Cada *table* possuí um método chamado *addlingvar*, que por sua vez configura as variáveis lingüísticas associadas a cada entrada, como, por exemplo, a variável linguistica 'poor' associada a entrada 'service', que apresenta uma função de pertinência gaussiana com desvio padrão igual a 1.5 e média 0.

- Configura a variável de saída.

```
-- configura a saída 'tip'
tip = fuzzy:addout( 'tip', 0, 30 )
tip:addlingvar( 'cheap', trimf, { 0, 5, 10 } )
tip:addlingvar( 'average', trimf, { 10, 15, 20 } )
tip:addlingvar( 'generous', trimf, { 20, 25, 30 } )
```

A variável *tip* é atribuída uma *table* com as configurações da variável como nome e faixa de valores de saída. A seguir é chamado o método para configurações das variáveis lingüísticas, que inicialmente adiciona a variável 'cheap' com função de pertinência triangular, onde o início do triângulo começa em 0, sobe até o valor de pertinência 1 e desce ao grau de pertinência 0 novamente em 10.

- Configuração das regras.

```
-- define a regra 1
r1 = fuzzy:addrule( 1, 'ormethod' )
r1:addpremise( false, 'service', 'poor' )
r1:addpremise( false, 'food', 'rancid' )
r1:addimplic( false, 'tip', 'cheap' )

-- define a regra 2
r2 = fuzzy:addrule( 1, 'andmethod' )
r2:addpremise( false, 'service', 'good' )
r2:addimplic( false, 'tip', 'average' )

-- define a regra 3
r3 = fuzzy:addrule( 1, 'ormethod' )
r3:addpremise( false, 'service', 'excellent' )
r3:addpremise( false, 'food', 'delicious' )
r3:addimplic( false, 'tip', 'generous' )
```

As variáveis *r1*, *r2* e *r3* são atribuídas *tables* com as configurações de cada regra, incluindo as premissas(*table pres*) e implicações (*table imps*), que são definidas pelos métodos *addpremise* e *addimplic* respectivamente.

- Executa uma simulação.

```
-- cria vetor de entrada, com 'service'=8.0 e 'food'=6.5
inps = { 8.0, 6.5 }

-- executa a simulação
outs = fuzzy:solve( inps )

-- imprime os valores as saídas
print( outs )
```

Neste trecho inicialmente é criado um vetor contendo os dados de entrada com 8.0 e 6.5, que representam o serviço prestado pelo restaurante(variável *'service'*) e a qualidade da comida(variável *'food'*), respectivamente. A seguir é realizada a simulação, passando como parâmetro para o método *solve* a variável *'inps'*. Esta função retorna um vetor com os valores de saída que finalmente são impressos na tela.

Para este exemplo o programa imprimiu para a saída *'tip'* o valor de 22.21, indicando que o garçom deve receber de gorjeta cerca de 22% do valor da conta.

## 4.3 Bloco LUABLC

Para que houvesse a possibilidade de se criar soluções que utilizassem tanto as técnicas de RNAs e lógica *fuzzy* desenvolvidas, em conjunto, foi necessário criar um "canal" de comunicação entre o *framework* de modelos do **FASEE**, desenvolvido em **C++**, e o *framework* de lógica *fuzzy*, desenvolvido em **Lua**. Este "canal" se traduz na criação de um bloco, derivado da classe *BLC*, chamado *LUABLC*. Esta classe que além de ser derevada da classe *BLC* também contém uma instância de um interpretador **Lua**, fazendo com que qualquer programa desenvolvido em **Lua** possa ser utilizado dentro do *framework* de modelos do **FASEE**, uma vez que todas as entradas e saída do bloco *LUABLC* são configuradas no ambiente **Lua**. A Figura 4.4 mostra o diagrama de classes do *LUABLC* e suas relações.

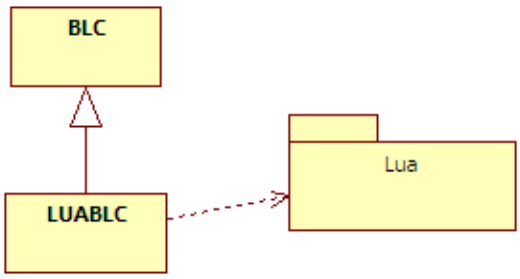


Figura 4.4: Diagrama de classes do bloco LUABLC

# Capítulo 5

# Resultados

Neste capítulo serão apresentados exemplos que foram criados usando o que foi desenvolvido ao longo do trabalho, salientado pontos que vão ao encontro dos objetivos do trabalho proposto.

## 5.1 Aplicações em Redes Neurais Artificiais: Redes MLP

### 5.1.1 Aproximador de Funções

A fim de exemplificar o uso das redes MLP foi criado um programa que a partir de dados de entrada e saídas desejada realiza um treinamento de uma rede MLP com diferentes quantidades de neurônios para verificar o comportamento da rede. Os dados de entrada são definidos em um vetor que varia de $[-\pi...\pi]$, divididos linearmente em 30 pontos. Foram realizados dois treinamentos onde, os dados de saídas desejados são o resultado de uma função seno que varia conforme a entrada. Por sua vez, no segundo os dados de treinamento são também o resultado de uma função seno que varia conforme a entrada, acrescidos de um erro aleatório, de forma a tornar mais visível o comportamento da rede através da comparação das saídas desejadas e das resultantes dos treinamentos.

Abaixo é mostrado a listagem do código **Lua** utilizado neste exemplo, com todas as linhas comentadas de forma a indicarem os seus propósitos.

```
1  -- carrega framework de RNA (arquivo 'luafann.dll')
2  require 'luafann'
3
4  -- cria matriz de dados de entrada
5  inp = datamat()
6
7  -- lê dados de entrada do arquivo 'input.mlp'
8  inp:load('input.mlp')
9
10 -- numero de entradas da rede
11 ninp = inp:maxcols()
12
```

```
13 -- cria matriz de dados de saída desejada
14 target = datamat()
15
16 -- lê dados de saída desejada do arquivo 'target.mlp'
17 target:load('target.mlp')
18
19 -- numero de saídas da rede
20 nout = target:maxcols()
21
22 -- cria rede MLP
23 m = mlp()
24
25 -- define estrutura da rede: 1 entrada, 4 neurônios camada escondida, 1 saída
26 m:setup(ninp,6,nout)
27
28 -- define parametro 'maxepoch'
29 m:param("maxepoch", 1000)
30
31 -- define parametro 'maxerror'
32 m:param("maxerror", 0.001)
33
34 -- define funcao de callback, chamada a cada época do processo de treinamento
35 function callback(msg)
36   print(string.format("Epoch: %d \t MSE: %.4f", msg.epoch, msg.mse))
37   return 0
38 end
39
40 -- treina a rede. (tipos de treinamento: 'normal', 'batch', 'rprop')
41 m:train(inp, target, "rprop", "callback")
42
43 -- cria matriz de dados de entrada para validação da rede
44 inp2 = datamat()
45
46 -- lê dados de entrada do arquivo 'input2.mlp'
47 inp2:load('input2.mlp')
48
49 -- numero de amostras dos dados de entrada
50 nsamples = inp2:maxrows()
51
52 -- cria matrix para guardar a saída da rede
53 out = datamat(nsamples,nout)
54
55 -- testa a rede
56 m:solve(inp, out)
57
58 -- salva saída da rede em arquivo
59 out:save('output.mlp')
```

No código acima a função *callback* é chamada a cada passo do processo iterativo de treinamento, imprimindo na tela a época atual do treinamento e seu respectivo erro médio quadrático (MSE). Abaixo é mostrado um trecho desta saída.

```
Epoch: 0          MSE: 0.1875
Epoch: 1          MSE: 0.1901
```

```
Epoch: 2          MSE: 0.1869
...
Epoch: 47         MSE: 0.0010
```

A Figura 5.1.1 mostra três gráficos, onde cada um contém duas curvas indicando a saída desejada da rede, e a saída final da rede após seu treinamento. Os gráficos mostram o resultado do treinamento da função seno com, respectivamente, zero, dois e quatro neurônios da camadas escondida. Já a Figura 5.1.1 mostra o resultado do treinamento da função seno com erro aleatório dado pela equação 5.1.

$$f(x) = sin(x) + rand() * 0.15 \tag{5.1}$$

onde $x$ também varia de $[-\pi...\pi]$, e a função $rand()$ retorna um número aleatório uniformemente distribuído entre [0...1]. Caso $f(x)$ ultrapasse os limites de [0...1] o valor do retorno da função $rand()$ é considerado negativo.

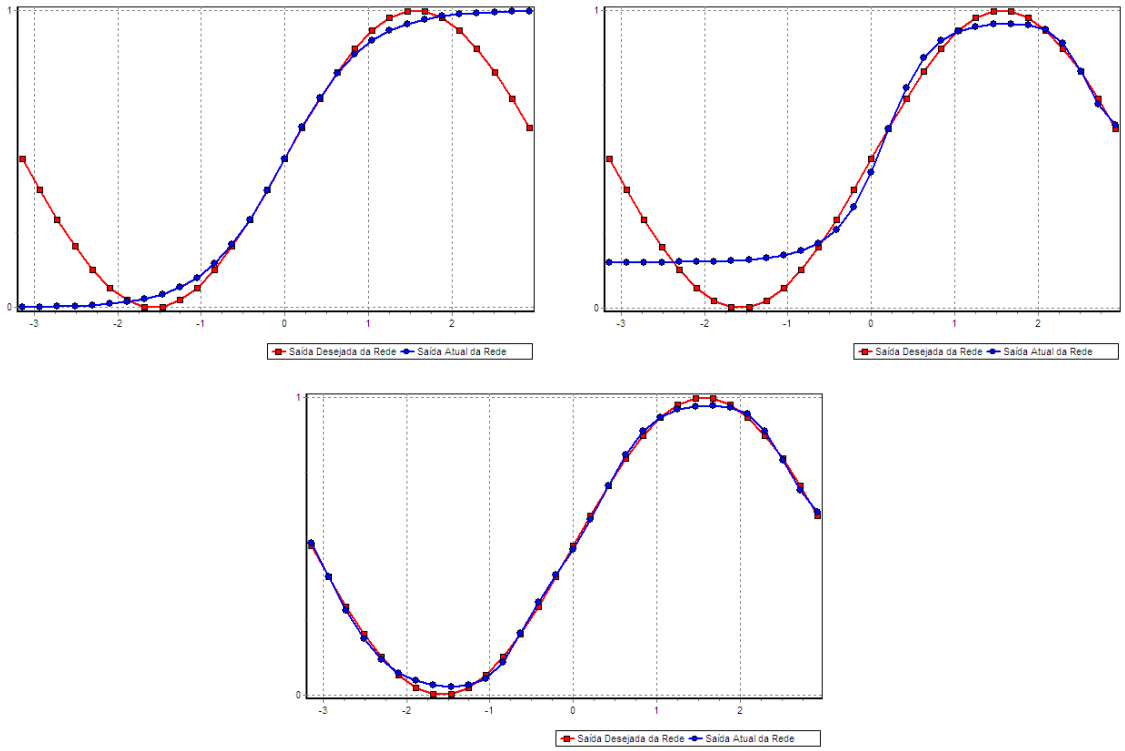


Figura 5.1: Aproximação da função seno com 0, 2 e 4 neurônios na camada escondida.

Analisando-se os gráficos das Figuras 5.1.1 e 5.1.1 pode-se, perceber que nos treinamentos com zero e dois neurônios na camada escondida a rede neural não foi capaz de extrair as características principais da função seno, pelo fato dela apresentar um número de não linearidades superior que a rede possa aprender. Este problema é conhecido na literatura como *underfitting* [1]. A solução é acrescentar mais neurônios a camada escondida, ou até mesmo

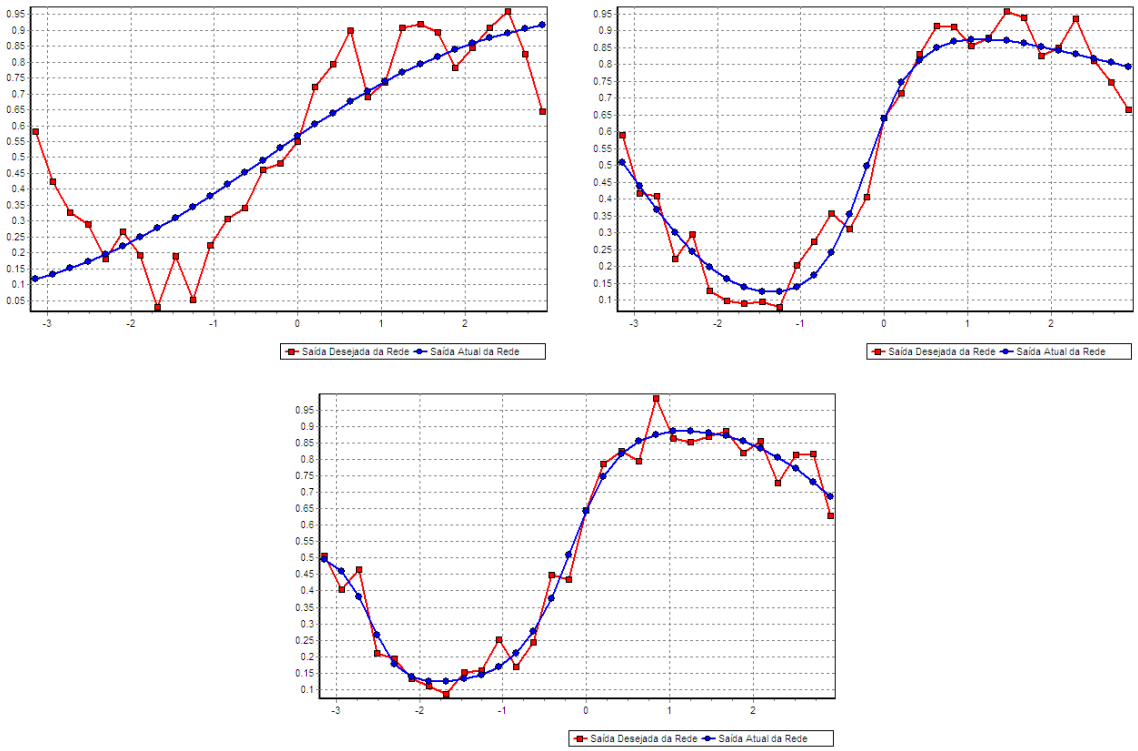


Figura 5.2: Aproximação da função seno (com erro aleatório) com 0, 2 e 4 neurônios na camada escondida.

acrescentar mais camadas escondidas.

## 5.2 Aplicações em Redes Neurais Artificiais: Redes SOM

### 5.2.1 Segmentação de Imagens

Segmentação se refere ao processo de dividir uma imagem em regiões ou objetos com o objetivo de simplificar e/ou mudar sua representação para facilitar a sua análise. A segmentação de imagens é tipicamente usada para localizar objetos e formas (linhas, curvas, etc) em imagens [8].

O resultado da segmentação de imagens é um conjunto de regiões/objetos, ou até mesmo um conjunto de contornos, extraídos da imagem. Como resultado, cada um dos pixels em uma mesma região é similar com referência a alguma característica ou propriedade computacional, tais como cor, intensidade, textura ou continuidade.

Na área de segmentação de imagens existem métodos baseados em formatos, como o método de detecção de linhas, bordas e pontos, e também os métodos baseados nas características dos *pixels*, como a detecção de cores e intensidades de tons de cinza. A idéia deste exemplo é unir os dois métodos: detectar linhas em uma imagem com auxilio da intensidade

dos tons de cinza de cada pixel.

A Figura 5.3 mostra uma imagem cujo objeto de interesse é um circulo, que neste caso está desfocado e parcialmente incompleto. O que se sabe é que o objeto apresenta uma forma fechada, podendo ser aproximada por uma rede SOM com topologia circular(anular), mesmo contendo ruído e faltando informações. A entrada do algoritmo de treinamento é uma matriz de três colunas que representa a posição $x$ e $y$ de cada pixel não nulo(com tom de cinza maior que 0) e o valor do seu tom de cinza. O número de linhas representa o número de pixels válidos na imagem, na qual também é o número de amostras expostas para o treinamento da rede. A idéia deste exemplo foi retirada do trabalhado [24].

Figura 5.3: Imagem com ruído original.

As Figuras 5.4, 5.5 e 5.6 mostram como o sistema foi capaz de recuperar as informações contidas na imagem original, com 15, 30 e 50 neurônios, respectivamente.

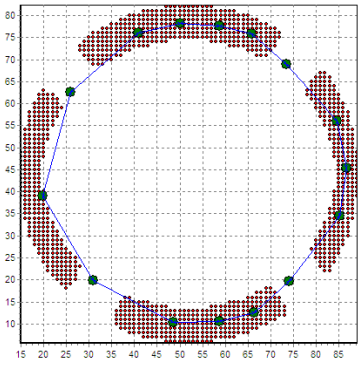

Figura 5.4: 15 neurônios.

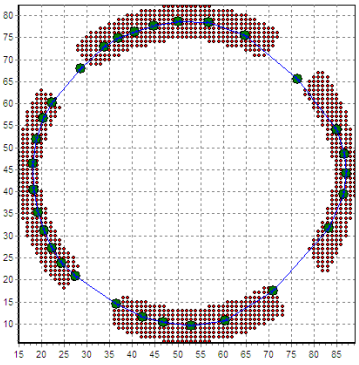

Figura 5.5: 30 neurônios.

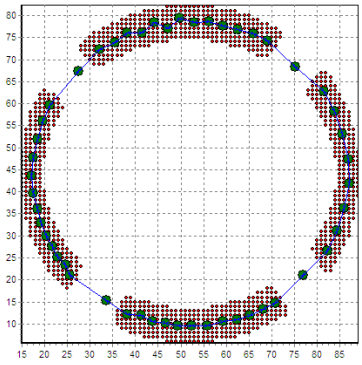

Figura 5.6: 50 neurônios.

Para detectar o centro dos segmentos que formam o círculo foi necessário adicionar uma nova funcionalidade as redes SOM: uma função de *callback*, chamada para calcular um coeficiente de ponderação, na qual é usada para ajustar os pesos de um determinado neurônio ganhador conforme um critério externo definido pelo usuário. Para o caso do exemplo, o critério externo é o valor do tom de cinza associado a cada pixel da imagem. Quanto mais "escuro" for a cor, maior o coeficiente de ponderação do ajuste dos pesos. Neste caso, o coeficiente pode ser calculado pela expressão da Eq. 5.2.

$$coef = \frac{winner_{value} - min_{gray}}{max_{gray} - min_{gray}} \tag{5.2}$$

Na Eq. 5.2 $winner_{value}$ é o valor do tom de cinza do neurônio ganhador, e $min_{gray}$ e $max_{gray}$ são os valores mínimos e máximos de tons de cinza encontrados na imagem. A funções de *callback* implementada é mostrada abaixo.

```
1
2 double wc( SOMNEURON* _winner, const int& _sample_idx, const DATAVEC& _input )
3 {
4   double coef = (_input[2] - min_gray)/(max_gray - min_gray);
5
6   return coef;
7 }
```

Neste exemplo, a função de *callback* funciona como uma espécie de ímã, onde os pontos na imagem com tons de cinza mais intenso tem maior influência durante o processo de atualização dos pesos. A eficiência deste método pode ser visualizada realizando o mesmo exemplo, porém retirando o coeficiente de ponderação. As Figuras 5.7, 5.8 e 5.9 mostram como o sistema reagiu sem este artifício.

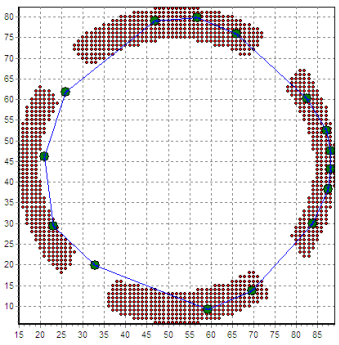

Figura 5.7: 15 neurônios.

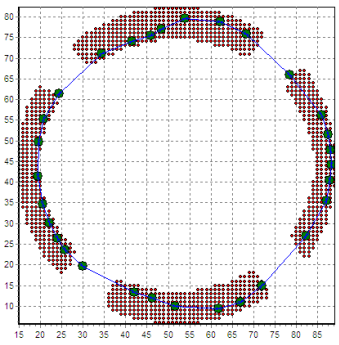

Figura 5.8: 30 neurônios.

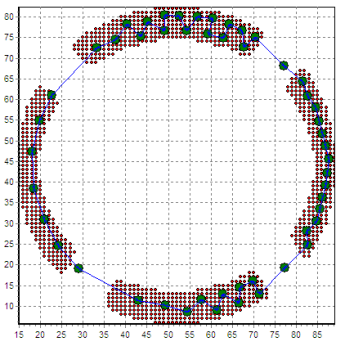

Figura 5.9: 50 neurônios.

Embora nas Figuras 5.7, 5.8 e 5.9 o sistema conseguiu detectar a forma fechada de um círculo, o resultado é menos preciso que o anterior, visto que o critério para o posicionamento dos neurônios não leva em consideração a intensidade dos tons de cinza de cada *pixel* da imagem, e sim, somente a posição dos *pixels* com intensidade superior a zero.

## 5.3 Aplicações de Lógica Difusa

### 5.3.1 Controle de Tensão em Redes de Distribuição

Nesta seção é descrito a solução que foi criada em [3] para o problema de controle de tensão em sistemas de distribuição, que por sua vez é uma extensão dos métodos desenvolvidos em [30]

e [6]. Este trabalho usa a teoria de controle *fuzzy* combinada com ferramentas de configuração topológica e fluxo de potência para redes de distribuição.

Para modelar os controladores *fuzzy* foi utilizado o *framework* de lógica *fuzzy* proposto neste trabalho, integrado ao programa de gestão de redes de distribuição **PSL ©DMS**, da empresa **PowerSysLab**.

Atualmente, em grande parte das empresas de distribuição de energia, o controle é realizado de forma não coordenada através dos operadores que se baseiam em estudos que utilizam carga prevista e condições topológicas prováveis de operação. As tomadas de decisões são, ainda, baseadas em características e "sentimentos" que não correspondem, hoje, a complexidade do sistema, e tais decisões podem causar desgastes dos equipamentos, violação dos níveis de segurança, bem como não serem eficazes e nem consistentes.

A idéia central é corrigir as violações de tensão de uma rede de distribuição de maneira a assegurar que todos os nós da rede estejam com seus valores de tensão dentro de uma faixa segura de operação.

Uma vez identificada a violação de tensão, é construída uma matriz que contém a efetividade de controle que cada dispositivo possuí em cada nó do sistema referente ao outro dispositivo, chamada de Matriz Sensibilidade. Baseada nessa matriz, a metodologia identifica qual dispositivo de controle é mais efetivo para o nó de pior violação de tensão. Dessa forma o programa deve corrigir ou melhorar a tensão no nó de pior violação e conseqüentemente em outros nós que também são, de maneira menos eficaz, atingidos pelas mudanças efetuadas em determinado dispositivo de controle. Isto significa que, para uma dada ação de controle, não apenas o nó diretamente em questão será afetado, mas também mais nós do sistema. Também é possível acontecer que a ação efetuada em determinado nó tenha como conseqüência uma violação de tensão em outro nó que em princípio não estava violado, mas provavelmente a beira da violação. Caso isso ocorra, esse nó irá entrar para o conjunto de nós com violação de tensão na próxima vez que o algoritmo verificar se todos os nós se encontram no intervalo desejado atribuído à faixa de regulação. Dessa forma, o algoritmo só pára quando todas as tensões estiverem dentro do intervalo adequado e pré-estabelecido de operação, ou porque não há mais dispositivos de controle para efetuar tais ações.

O método proposto foca então em regular o nível de tensão do sistema de distribuição, coordenadamente, utilizando como dispositivos de controle o transformadores com tap sobre carga, banco de capacitores da subestação, os reguladores de tensão e os bancos de capacitores alocados ao longo dos alimentadores. Uma descrição mais detalhada tanto do problema de coordenação de tensão em redes de distribuição pode ser vista em [3].

As variáveis de entrada da solução são:

- A Matriz Sensibilidade;
- A posição dos bancos de capacitores e suas respectivas potências;

- A posição do tap dos transformadores e/ou reguladores de tensão;
- A violação de tensão.

O objetivo é definir qual ação de controle deve ser realizada, quando existir um problema de sub ou sobretensão. Para isto, é definida uma banda morta variando de $V_{min}$ a $V_{max}$, na qual devem ser colocadas as tensões de todos os nós do SD. Se em algum nó houver violação de tensão, o algoritmo identifica a pior violação, e através da matriz sensibilidade, identifica o dispositivo de controle que tem mais efetividade de controle sobre o nó com violação de tensão.

As entradas normalizadas do controlador C1 são eficiência e posição, e resultam em um sinal de poder de controle que junto com a variável violação de tensão, igualmente normalizada, constituem as variáveis de entrada do controlador C2 e que tem como sinal de saída a ação de controle. Este conjunto de controladores representa uma cascata de controladores de Mamdani, conectados conforme a Figura 5.10.

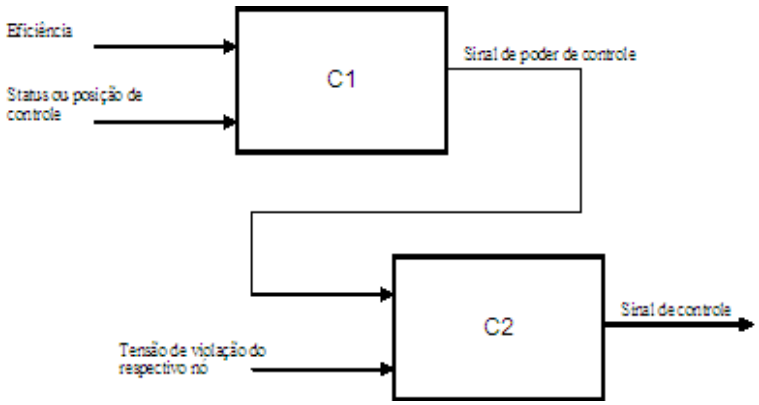


Figura 5.10: Controladores *fuzzy*.

Nesta etapa do programa o mapa de regras dos controladores tem extrema importância. Nelas é armazenado o conhecimento do especialista. Neste trabalho, foram implementados os mapas de regras apresentados em [6], apresentados nas Figuras 5.11 e 5.12.

| | **NB** | **NS** | **ZE** | **PS** | **PB** |
|---|---|---|---|---|---|
| NB | PB | PB | PB | PS | ZE |
| NS | PB | PB | PS | ZE | NS |
| ZE | PB | PS | ZE | NS | NB. |
| PS | PS | ZE | NS | NB | NB |
| PB | ZE | NS | NB | NB | NB |

Figura 5.11: Mapa de regras do controlador C1 [30].

| | NB | NS | ZE | PS | PB |
|---|---|---|---|---|---|
| **NB** | PB | PB | ZE | NB | NB |
| **NS** | PS | PS | ZE | NS | NS |
| **ZE** | ZE | ZE | ZE | ZE | ZE. |
| **PS** | NS | NS | ZE | PS | PS |
| **PB** | NB | NB | ZE | PB | PB |

Figura 5.12: Mapa de regras do controlador C2 [30].

As cinco siglas representam as cinco funções de pertinência e significam: número negativo grande (NB), negativo pequeno (NS), zero (ZE), positivo pequeno (PS) e positivo grande (PB).

No mapa do controlador C1 as linhas representam a entrada eficiência e as colunas o *status*. Em seguida, no mapa do controlador C2, as linhas representam as violações de tensão e as colunas o poder de controle que é o sinal de saída do controlador C1.

São cinco funções de pertinência, para duas variáveis de entrada, o que gera 25 regras a serem interpretadas conforme o valor da variável de entrada. Neste exemplo, a entrada de eficiência é um número grande negativo e a entrada de posição também. Como resultado do mapa de regras do controlador C1, deve-se obter um número grande positivo para o sinal poder de controle. Todas as entradas e saídas estão normalizadas dentro do intervalo $[-1...1]$.

Para validação do sistema usando o *framework* de lógica *fuzzy* integrado ao **PSL ©DMS** o mesmo sistema de distribuição usado em [3] foi modelado no **PSL ©DMS**. Para um mesmo nível de carregamento do sistema, comparou-se as saídas dos dois trabalhos. O resultado foi que as saídas foram iguais, incluindo resultados parciais do processo iterativo de identificação e correção das tensões dos nós da rede. Assim podemos concluir que o *framework* de lógica *fuzzy* se comportou de forma adequado em comparação a um *software* de lógica *fuzzy* consagrado, que é o *Toolbox* de *Fuzzy Logic* do Matlab.

## 5.4 Aplicação Neuro-*Fuzzy*

### 5.4.1 Exemplo Neuro-*Fuzzy* Cooperativo

Este exemplo ilustrará o uso do *framework* de redes neurais artificiais e o de lógica *fuzzy* integrados. O objetivo é criar um sistema neuro-*fuzzy* cooperativo, no qual funções de pertinência do sistema *fuzzy* sejam aproximadas por redes neurais artificiais, a partir de dados de treinamento definidos pelo usuário [29].

O sistema *fuzzy* exemplo será o mesmo utilizado como exemplo no Capítulo 4, onde o objetivo é determinar a gorjeta de um garçom, considerando-se a qualidade do serviço e da comida da casa.

A modificação no exemplo original foi a troca da função de pertinência da variável linguística *poor* da entrada *service* de uma função gaussiana, de média 0.0 e desvio padrão 1.5, por uma rede neural que, não por acaso, aproxima, durante a etapa de treinamento, uma função gaussiana de mesmos parâmetros. A escolha destes dados de treinamento é justificado pelo fato que posteriormente há uma comparação da saída do exemplo original com a do modificado.

A listagem completa do exemplo modificado é mostrado abaixo.

```
1  -- carrega framework de logica fuzzy
2  require 'luafuzzy'
3
4  -- carrega framework de redes neurais artificiais
5  require 'luafann'
6
7  ------------------------------------------------------------
8  -- RNA
9  ------------------------------------------------------------
10
11 -- cria rede MLP
12 m = mlp()
13
14 -- configura a rede
15 m:setup( 1, 6, 1 )
16
17 -- cria matriz de dados de entradas
18 inp = datamat()
19 -- lê dados do arquivo...
20 inp:load('gauss_input.txt')
21
22 -- cria matriz de dados de saída desejados
23 target = datamat()
24 -- lê dados do arquivo...
25 target:load('gauss_target.txt')
26
27 -- define funcao de callback
28 function callback( msg )
29   --if math.mod( msg.epoch, 10 )== 0 then
30     print( string.format('Epoch: %d \t MSE: %.4f', msg.epoch, msg.mse ) )
31   --end
32   return 0
33 end
34
35 -- define tolerância do erro da rede
36 m:param('maxerror',0.0001)
37
38 -- realiza o treinamento da rede
39 m:train( inp, target, 'callback' )
40
41 -- cria matriz de entrada da funcao de pertinencia
42 inp_mlp = datamat(1,1)
43
44 -- cria matriz de saída da funcao de pertinencia
45 out_mlp = datamat(1,1)
```

```
46
47  ------------------------------------------------------------
48  -- Cria funcao de pertinencia que usa a rede MLP
49  ------------------------------------------------------------
50
51  function gaussmf_mlp( x, params )
52
53    -- define valor de entrada da rede
54    inp_mlp:set(0,0, x )
55
56    -- resolve a rede
57    m:solve( inp_mlp, out_mlp )
58
59    -- recupera e retorna valor de saída da rede
60    return out_mlp:get(0,0)
61  end
62
63  ------------------------------------------------------------
64  -- Lógica Fuzzy
65  ------------------------------------------------------------
66
67  -- cria sistema fuzzy
68  local fuzzy = luafuzzy()
69
70  -- configura entrada 'service'
71  local serv = fuzzy:addinp( 'service', 0, 10 )
72  --serv:addlingvar( 'poor', gaussmf, { 1.5, 5. } ) --<< FUNCAO ORIGINAL COMENTADA
73  serv:addlingvar( 'poor', gaussmf_mlp ) --<< NOVA FUNCAO DE PERTINENCIA RNA
74  serv:addlingvar( 'good', gaussmf, { 1.5, 5. } )
75  serv:addlingvar( 'excellent', gaussmf, { 1.5, 10. } )
76
77  -- configura entrada 'food'
78  local food = fuzzy:addinp( 'food', 0, 10 )
79  food:addlingvar( 'rancid', trapmf, { 0, 0, 1, 3 } )
80  food:addlingvar( 'delicious', trapmf, { 7, 9, 10, 10 } )
81
82
83  -- configura saída 'tip'
84  local tip = fuzzy:addout( 'tip', 0, 30 )
85  tip:addlingvar( 'cheap', trimf, { 0, 5, 10 } )
86  tip:addlingvar( 'average', trimf, { 10, 15, 20 } )
87  tip:addlingvar( 'generous', trimf, { 20, 25, 30 } )
88
89  -- configura regra 'r1'
90  local r1 = fuzzy:addrule( 1, 'ormethod' )
91  r1:addpremise( false, 'service', 'poor' )
92  r1:addpremise( false, 'food', 'rancid' )
93  r1:addimplic( false, 'tip', 'cheap' )
94
95  -- configura regra 'r2'
96  local r2 = fuzzy:addrule( 1, 'andmethod' )
97  r2:addpremise( false, 'service', 'good' )
98  r2:addimplic( false, 'tip', 'average' )
99
100 -- configura regra 'r3'
```

```
101 local r3 = fuzzy:addrule( 1, 'ormethod' )
102 r3:addpremise( false, 'service', 'excellent' )
103 r3:addpremise( false, 'food', 'delicious' )
104 r3:addimplic( false, 'tip', 'generous' )
105
106 ------------------------------------------------------------
107 -- Testa Sistema Neuro-Fuzzy Cooperativo
108 ------------------------------------------------------------
109
110 -- cria valores para entrada 'service'
111 for s = serv.mn,serv.mx,2.5 do
112
113   -- cria valores para entrada 'food'
114   for f = food.mn,food.mx,2.5 do
115
116     -- resolve a sistema fuzzy: retorna valor de 'tip'
117     local t = fuzzy:solve( s, f )
118
119     print(string.format('Service=%.1f Food=%.1f Tip=%.4f',s,f,t) )
120   end
121 end
```

A linha 73 da listagem acima mostra o ponto onde a nova função de pertinência que usa a rede MLP treinada é definida. Para fins de comparação os dois exemplos, original e o modificado, foram executados. Abaixo é mostrado a comparação entre as duas saídas.

Saída do Exemplo Modificado

```
Service=0.0 Food=0.0 Tip=5.0780
Service=0.0 Food=2.5 Tip=5.0780
Service=0.0 Food=5.0 Tip=5.0780
Service=0.0 Food=7.5 Tip=11.1139
Service=0.0 Food=10.0 Tip=5.0780
Service=2.5 Food=0.0 Tip=8.0406
Service=2.5 Food=2.5 Tip=9.7548
Service=2.5 Food=5.0 Tip=9.7548
Service=2.5 Food=7.5 Tip=14.6719
Service=2.5 Food=10.0 Tip=9.7548
Service=5.0 Food=0.0 Tip=10.0608
Service=5.0 Food=2.5 Tip=12.0286
Service=5.0 Food=5.0 Tip=14.8055
Service=5.0 Food=7.5 Tip=17.7927
Service=5.0 Food=10.0 Tip=14.8055
Service=7.5 Food=0.0 Tip=11.9924
Service=7.5 Food=2.5 Tip=14.9927
Service=7.5 Food=5.0 Tip=19.6783
Service=7.5 Food=7.5 Tip=19.6840
Service=7.5 Food=10.0 Tip=19.6783
Service=10.0 Food=0.0 Tip=15.0000
Service=10.0 Food=2.5 Tip=18.8915
Service=10.0 Food=5.0 Tip=24.5897
Service=10.0 Food=7.5 Tip=24.5897
Service=10.0 Food=10.0 Tip=24.5897
```

Saída do Exemplo Original

```
Service=0.0 Food=0.0 Tip=5.0780
Service=0.0 Food=2.5 Tip=5.0780
Service=0.0 Food=5.0 Tip=5.0780
Service=0.0 Food=7.5 Tip=11.1085
Service=0.0 Food=10.0 Tip=5.0780
Service=2.5 Food=0.0 Tip=8.0406
Service=2.5 Food=2.5 Tip=9.9968
Service=2.5 Food=5.0 Tip=10.0023
Service=2.5 Food=7.5 Tip=15.0073
Service=2.5 Food=10.0 Tip=10.0023
Service=5.0 Food=0.0 Tip=10.0608
Service=5.0 Food=2.5 Tip=12.0286
Service=5.0 Food=5.0 Tip=15.0000
Service=5.0 Food=7.5 Tip=17.9714
Service=5.0 Food=10.0 Tip=15.0000
Service=7.5 Food=0.0 Tip=11.9924
Service=7.5 Food=2.5 Tip=14.9927
Service=7.5 Food=5.0 Tip=19.9977
Service=7.5 Food=7.5 Tip=20.0032
Service=7.5 Food=10.0 Tip=19.9977
Service=10.0 Food=0.0 Tip=15.0000
Service=10.0 Food=2.5 Tip=18.8915
Service=10.0 Food=5.0 Tip=24.9220
Service=10.0 Food=7.5 Tip=24.9220
Service=10.0 Food=10.0 Tip=24.9220
```

Realizando-se uma análise superficial sobre as saídas dos dois exemplos, percebe-se que os valores de gorjeta(*tip*) do exemplo modificado e do exemplo original apresentaram valores semelhantes para um mesmo valor da qualidade do serviço(*service*) e da comida(*food*). Assim podemos concluir que a rede neural artificial aproximou de forma satisfatório a função gaussiana, a ainda seu uso no *framework* de lógica *fuzzy* se deu de forma apropriada.

# Capítulo 6

# Conclusões

O trabalho apresentado propôs a criação de um *framework* de redes neurais artificiais e um *framework* de lógica *fuzzy*. Para tanto, inicialmente foi realizada uma descrição sobre a estrutura básica de desenvolvimento deste trabalho, o **FASEE**, onde foi então detalhado seu *framework* de modelos, ressaltando seu mecanismo de derivadas parciais, funcionalidade na qual simplificou consideravelmente a implementação de alguns algoritmos de treinamento RNAs.

O *framework* de redes neurais artificiais foi todo desenvolvido em **C++** usando o *framework* de modelos do **FASEE**. Dois tipos de redes foram implementados: a criação de um ambiente de manipulação de redes *Perceptron* Multi-Camadas (*Multi-Layer Perceptron*) e um de Mapas Auto-Organizados (*Self Organizing-Maps*).

Para as redes MLP, foi desenvolvido o método de treinamento *back-propagation* com três variantes: treinamento por padrão(*on-line*), treinamento por ciclo(*batch*), e uma variação deste segundo chamada de *RPROP*. O *RPROP* introduz uma heurística no processo de treinamento, aumentando de forma significativa a velocidade de convergência.

Um aspecto importante que, embora levando em consideração durante o desenvolvimento, mas que não se teve sucesso na implementação, foi que, ao se criar uma rede MLP, não é possível definir uma funções de ativação qualquer, em que automaticamente é usada uma função *sigmoidal*. Entretanto o sistema possibilita a troca das funções de ativação pelo usuário após a sua criação, contornando em um primeiro momento este problema.

Para as redes SOM foi desenvolvido um ambiente no qual durante a criação de redes as únicas informações necessárias são o número de entradas e o número de neurônios da rede, onde posteriormente todas as entradas são conectadas a todos os neurônios. As topologias são definidas durante a etapa de treinamento. Estas podem ser as já pré-definidas de duas dimensões (circular, retangular ou hexagonal), ou topologias definidas pelo usuário, na qual uma matriz contendo a posição de cada neurônio num espaço $\Re^n$ é usada, e o critério para o ajuste dos pesos durante a etapa de adaptação sináptica é a distância euclidiana entre os neurônios. Assim, dois neurônios são considerados vizinhos quando a distância entre eles for

igual a 1.

Um requisito proposto na introdução deste trabalho, e que não foi atendido, foi a criação de um ambiente de redes de base radial(RBF). Embora o *framework* de redes contemplasse essa possibilidade, não foi possível a sua criação devido a falta de tempo hábil para levantamento de requisitos e de implementação.

Em ambos os casos, redes MLP e SOM, todas as suas funcionalidade também foram disponibilizadas para serem usadas em **Lua**.

O objetivo inicial do desenvolvimento do *framework* de lógica *fuzzy* era que este fosse desenvolvido na mesma linha do de RNAs, usando como base o *framework* de modelos do **FASEE**. Porém, devido a uma limitação do **FASEE**, o *framework* de lógica *fuzzy* foi inteiramente desenvolvido em **Lua**. A limitação do **FASEE** é a impossibilidade de multiplexar e demultiplexar entradas e saídas, sendo essa funcionalidade necessária para a transação de *fuzzysets* entre blocos. Outro ponto decisivo para escolha da linguagem **Lua** foi sua reconhecida capacidade de poder ser "embutida" em programas escritos em **C++**, permitindo assim trafegar dados e chamar funções/métodos do **Lua** no **C++**, e vice-versa.

No *framework* de lógica *fuzzy* foi criada o motor de inferência Mamdani. Os pontos flexíveis do sistema que podem ser definidos pelo usuário são: a definição de novas funções de pertinência, definição de novos operadores *fuzzy*, e a definição de novos métodos de *defuzzificação*, no qual os únicos requisitos são que, para cada funcionalidade, as novas funções definidas pelo usuário deverão implementar uma interface já definida pelo sistema, ou seja, por exemplo, para se definir uma nova função de *defuzzificação* o usuário deve criar uma função que receba como parâmetro um *fuzzyset* – vetor de pares ordenado $(x, y)$ – e retornar uma valor escalar que identifique o *fuzzyset* a partir do critério escolhido(centro de massa, médias dos máximos, etc.).

Embora as técnicas de redes neurais e lógica *fuzzy* tenham sido desenvolvidas em ambientes diferentes, elas podem ser usadas em conjunto. Para a integração foi necessário a criação de um bloco especial no **FASEE** chamado de *LUABLC*. Este bloco contém um interpretador **Lua**. Assim, é possível usar, por exemplo, uma rede neural para representar uma função de pertinência em um sistema *fuzzy*, ou mesmo usar um sistema *fuzzy* como função de ativação de um determinado neurônio em uma rede neural, caracterizando assim em ambos os casos um sistema híbrido neuro-*fuzzy*.

Para avaliar os *frameworks* desenvolvidos, diversas aplicações foram criadas, e diferentes simulações foram realizadas, mostradas no capítulo de aplicações, a fim de comparar resultados numéricos do sistema proposto com sistemas já existentes e consagrados, como, por exemplo, o *toolbox* de redes neurais artificiais e lógica *fuzzy* do Matlab. Em todos os casos os resultados coincidiram, revelando uma correta implementação das técnicas.

Deve-se salientar que nenhum teste de desempenho foi realizado, ou seja, nenhuma comparação foi realizada, em termos de velocidade e uso de memória, do que foi desenvolvido

neste trabalho com outros sistemas similares.

A maior dificuldade encontrada durante o desenvolvimento deste trabalho foi a integração das técnicas de RNA e lógica *fuzzy*, onde, inicialmente tentou-se unir as duas metodologias em uma mesma plataforma de desenvolvimento, o **FASEE**, que devido a sua limitação não foi possível, e na qual a solução foi criar as metodologias em plataformas(linguagens) diferentes e, então, criar uma "porta" de comunicação entre elas: o bloco *LUABLC*.

Assim, podemos concluir que, tanto o *framework* de RNAs quanto o de lógica *fuzzy*, atenderam de forma satisfatória as expectativas iniciais, e que, juntamente com a criação do bloco *LUABLC*, criou-se um *framework* integrado de redes neurais artificiais e lógica *fuzzy*, meta principal desse trabalho.

Como trabalhos futuros na parte de redes neurais artificiais pode-se citar alguns pontos que venham a contribuir e/ou completar este trabalho, como a implementação de outros tipos de redes, como redes de base radial(RBF), redes recorrentes de Hopfield, e a criação uma interface gráfica para manipulação das redes.

No que diz respeito a aperfeiçoamentos e ampliações do *framework* de lógica *fuzzy* pode-se citar a implementação da máquina de inferência Sugeno, desenvolver a funcionalidade de multiplexação e demultiplexação de dados trocados entre blocos do *framework* **FASEE** para uma possível reimplementação do sistema, e também a criação de uma interface gráfica para manipulação de sistemas *fuzzy*.

Outro trabalho futuro interessante seria a integração dos resultados deste trabalho com o *framework* de Algoritmos Genéticos já existe no **FASEE**, abrindo outras possibilidades de criação de sistemas de inteligência artificial híbridos.

# Referências Bibliográficas


[1] André Carlos Ponce de Leon Ferreira Carvalho Antônio de Pádua Braga, Teresa Bernarda Ludermir. *Redes Neuráis Artificiais - Teoria e Aplicações*. LTC, 2000.

[2] M. Bonner, S. Mayer, A. Raggl, and W. Slany. FLIP++ A fuzzy logic inference processor library. *Fuzzy Logic in Artificial Intelligence: Towards Intelligent Systems: IJCAI'95 Workshop, Montréal, Canada, August 19-21, 1995, Selected Papers*, 1997.

[3] Leonardo Elizeire Bremermann. Ccontrole fuzzy volt/var em sistemas de distribuição. Master's thesis, PUCRS, 2008.

[4] E. Cox. *The fuzzy systems handbook: a practitioner's guide to building, using, and maintaining fuzzy systems*. Academic Press Professional, Inc. San Diego, CA, USA, 1994.

[5] H. Demuth and M. Beale. Neural network toolbox user's guide. *Matlab user's guide*, 2001.

[6] V. MIRANDA et. Alli. An improved fuzzy inference system for voltage/var control. *IEEE Transactions*, 22, 2007.

[7] W. GHARIEB and G. NAGIB. SIMULINK FUZZY LOGIC LIBRARY.

[8] E.L. Hall. *Computer image processing and recognition*. Academic Press New York, 1979.

[9] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1999.

[10] R. Ierusalimschy. *Programming In Lua*. Roberto Ierusalimschy, 2003.

[11] R. Ierusalimschy, L.H. de Figueiredo, and W. Celes. Lua 5.1 Reference Manual. *Lua. org*, 2006.

[12] Euvaldo Cabral Jr. *Redes Neurais Artificiais: modelos em c*. ReNeArt, 2003.

[13] F. S. OSÓRIO J.R. BITTENCOURT. Annef - artificial neural networks framework: Uma solução software livre para o desenvolvimento, ensino e pesquisa de aplicações de

inteligência artificial multiplataforma. *II Workshop sobre Software Livre. Porto Alegre*, pages p.13–16, 2001.

[14] F. S. OSÓRIO J.R. BITTENCOURT. Fuzzyf - fuzzy logic framework: Uma solução software livre para o desenvolvimento, ensino e pesquisa de aplicações de inteligência artificial multiplataforma. *III Workshop sobre Software Livre. Porto Alegre*, 2002.

[15] T. Kohonen. *Self-Organizing Maps*. Springer, 2001.

[16] Zsolt L. Kovács. *Redes Neurais Artificias: Fundamentos e Aplicações*. Collegium Cognitio, 1996.

[17] Alessandro Manzoni. *Desenvolvimento de um sistema computacional orientado a objetos para sistemas elétricos de potência: aplicação a simulação rápida e análise da estabilidade de tensão*. PhD thesis, Universidade Federam do Rio de Janeiro(UFRJ), Março 2005.

[18] Paolo Marrone. *JOONE - Java Object Oriented Neural Engine: The Complete Guide*, 2007. www.joone.org.

[19] K. Mehrotra, C.K. Mohan, and S. Ranka. *Elements of Artificial Neural Networks*. MIT Press Cambridge, MA, USA, 1996.

[20] A.J. Mendez, E.G. Rosello, M.J. Lado, J.G. Dacosta, D.M. Torres, and M.P. Cota. IMO. Net Artificial Neural Networks: an object-oriented reusable software component library to integrate Matlab Neural Networks functionality. *Proceedings of the Seventh Mexican International Conference on Computer Science (ENC'06)-Volume 00*, pages 159–163, 2006.

[21] Steffen Nissen. *Implementation of a Fast Artificial Neural Network Library (fann)*, 2003.

[22] L.D. Paulson. Developers Shift to Dynamic Programming Languages. *Computer*, 40(2):12–15, 2007.

[23] S. Rabin. *AI Game Programming Wisdom*. Charles River Media, 2002.

[24] C.C. Reyes-Aldasoro. Image Segmentation with Kohonen Neural Network Self-Organising Maps. *citeseer. nj. nec. com/460754. html*.

[25] M. Riedmiller. Rprop - description and implementation details. *University of Karlsruhe*, 1994.

[26] Joey Rogers. *Object-Oriented Neural Networks in C++*. Academic Press, 1997.

[27] DE Rumelhart, GE Hinton, and RJ Williams. Learning internal representations by error propagation. *Mit Press Computational Models Of Cognition And Perception Series*, pages 318–362, 1986.

[28] S.J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. 1995.

[29] Lefteri H. Tsoukalas and Robert E. Uhrig. *Fuzzy and neural approaches in engineering*. Wiley New York, 1997.

[30] P. CALISTO V. MIRANDA. A fuzzy inference system to voltage/var control in dms - distribution management system. *14th PSCC*, 2002.

[31] B. Widrow. Generalization and information storage in networks of ADALINE neurons. *Self-Organizing Systems*, pages 435–461, 1962.

[32] A. Zell, N. Mache, R. Hubner, G. Mamier, and M. Vogt. SNNS: Stuttgart Neural Network Simulator. *User Manual, Version*, 4, 1995.