Kipu: Um Simulador para Redes Neurais Construtivas ¹

Ernesto Cuadros Vargas

Orientação:

Prof. Dr. André Carlos Ponce de Leon Ferreira de Carvalho

*Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação
- USP, como parte dos requisitos para a obtenção do título de Mestre em Ciências
- Área de Ciências de Computação e Matemática Computacional.*

USP - São Carlos
Agosto de 1998

¹Trabalho realizado com auxílio da CAPES e do CNPq.

Aos meus pais Yolanda e Ernesto
que estão me acompanhando neste
momento importante da minha vida.

E. C. V.

Agradecimentos

Em primeiro lugar, agradeço a Deus pela oportunidade que me deu de poder estudar na USP, se não fosse por ele, eu não teria tido a coragem suficiente para sair de Arequipa sem saber com certeza onde tinha que chegar.

Aos meus pais Yolanda e Ernesto e aos meus irmãos Chepita e Alexito pelo amor, apoio e a compreensão que me deram desde a primeira vez que pensei em sair do meu país, Peru, para fazer o mestrado.

Ao meu amor Govy, pelo seu amor e especialmente pelo apoio e compreensão que nunca me faltou ao longo destes dois anos.

Ao meu orientador e especialmente amigo, André. Obrigado pela ajuda, conselhos e especialmente correções que me fez desde que comecei no programa de mestrado, até o último *slide* da minha apresentação.

Ao meu amigo Humberto que aceito o desafio de formar parte deste projeto, sem a sua ajuda não poderíamos ter alcançado os objetivos traçados.

Aos meus amigos e colegas Chandler, Raul, Jaque, Fabio, Jorge Félix, Luciana, Thelma, Tatiana, Valéria, Andrea, Augusto, Luis Carlos, Rudinei, Mirla, Marcia, Ana Elisa, Luciano, Sandra, Cris, Fabiano, Luis Augusto, Nilda, Paulo e Marco pela amizade e por todas as correções de português, além das festas e churrascos que compartilhamos.

Aos meus professores Edson dos Santos Moreira, Freddy João Valente, Maria da Graça Pimentel, Sandra Maria Aluísio, Maria Carolina Monard, Solange Oliveira Rezende, Marcos José Santana, e Caetano Traina Junior pelos ensinamentos e todas as lições dadas.

Ao pessoal da seção de Pós-Graduação: Marília, Laura e Beth, pela ajuda e apoio oferecido ao longo destes dois anos.

Ao CNPq e a CAPES pelo auxílio financeiro fornecido ao longo destes dois anos.

Agradecimientos

En primer lugar, agradezco a Dios por la oportunidad que me dio de estudiar en la Universidad de São Paulo, si no fuese por él, yo nunca hubiera sido capaz de enfrentar el primer viaje con la inseguridad de no saber exactamente lo que me esperaba, ni a donde llegaría.

A mis padres Yolanda, Ernesto y a mis hermanos Chepita y Alexito por el amor, apoyo y comprensión que recibí desde la primera vez que pense en hacer la maestría fuera del Perú.

A mi amor Govy, por su amor y especialmente por su apoyo y comprensión que hicieron que estos dos años pasen mucho más rápido de lo que esperábamos.

A mi asesor y especialmente amigo, André. Gracias por la ayuda incondicional ofrecida en todo momento, por los consejos y especialmente por todas las correcciones que recibí desde el primer de clases, hasta el último *Slide* de esta trabajo.

A mi amigo y colega Humberto que aceptó el reto de formar parte de este proyecto, sin su ayuda no hubieramos alcanzado los objetivos trazados al inicio del proyecto.

A mis amigos y colegas Chandler, Raul, Jaque, Fabio, Jorge Félix, Luciana, Thelma, Tatiana, Valéria, Andrea, Augusto, Luis Carlos, Rudinei, Mirla, Marcia, Ana Elisa, Luciano, Sandra, Cris, Fabiano, Luis Augusto, Nilda, Paulo y Marco, por su amistad y por todas la veces que me corrigieron el portugués, además de todas aquellas veces que pudimos compartir momentos de alegría.

A mis profesores Edson dos Santos Moreira, Freddy João Valente, Maria da Graça Pimentel, Sandra Maria Aluísio, María Carolina Monard, Solange Oliveira Rezende, Marcos José Santana y Caetano Traina Junior por la enseñanza y por todas las clases dadas.

A la gente de la secretaría de la Pos-Graduación: Marília, Laura y Beth, por la ayuda y apoyo ofrecido a lo largo de estos dos años.

Al CNPq y al CAPES por el apoyo financiero ofrecido a lo largo de estos 2 años.

Resumo

Este trabalho descreve o projeto, desenvolvimento e aplicação de um simulador de Redes Neurais Construtivas para o Reconhecimento de Padrões. Este trabalho faz parte do projeto SAPRI (Sistema para Aquisição, Processamento e Reconhecimento de Imagens) que está sendo desenvolvido para a Marinha do Brasil.

A definição da topologia de uma Rede Neural é um dos principais aspectos a serem considerados quando da sua utilização. A alternativa mais comum envolve a utilização de Redes Neurais com topologia fixa. O problema apresentado por essa técnica é que um grande número de arquiteturas precisam ser testadas antes de encontrar a melhor arquitetura. Os algoritmos construtivos tentam encontrar a melhor topologia para resolver um determinado problema sem a intervenção do usuário.

Para verificar o correto funcionamento dos algoritmos construtivos implementados junto ao simulador, foram realizados diversos experimentos utilizando três conjuntos de dados. O primeiro conjunto é formado por Imagens de Navios capturadas de um radar pela Marinha do Brasil. O segundo conjunto utilizado é formado por vetores de características de Imagens de Navios obtidos da Marinha Americana. Finalmente, o terceiro conjunto de dados utilizado é formado por vetores de características de Silhuetas de Veículos.

Abstract

This work describes the design, development and analysis of a simulator of constructive neural networks to be used for pattern recognition. This work is part of the SAPRI (system for patterns acquisition, processing and recognition) which is being developed for the Brazilian Navy.

The definition of the network topology is one of the main aspects to be considered for its use. The most common approach is the use of fixed topology neural networks. The problem of this approach is the high number of architectures to be tested before the best one is chosen. The constructive algorithms try to find the best topology to solve a given problem without the user intervention.

In order to verify the correct performance of the constructive algorithms implemented by the simulator, a few experiments were carried out using three different datasets. The first dataset is composed by radar images of ships collected by the Brazilian Navy. The second dataset is made of feature vectors of ships obtained from the American Navy. Finally, the third dataset used is formed by feature vectors of vehicles silhouettes.

Sumário

1	Introdução	1
1.1	Reconhecimento de Padrões	3
1.2	Objetivos Referentes à Classificação de Padrões	5
1.3	Organização do Trabalho	6
2	Redes Neurais Construtivas	7
2.1	Introdução	7
2.2	Taxonomia das Redes Neurais Construtivas	8
2.3	Algoritmo Cascade Correlation	9
2.4	Algoritmo UpStart	13
2.5	Algoritmo Tiling e MTiling	15
2.6	Algoritmo Tower	18
2.7	Algoritmo Pyramid	21
2.8	Algoritmo RBF-DDA	23
2.9	Conclusões	25
3	Simuladores de Redes Neurais	26
3.1	Introdução	26
3.2	Processo de Simulação de Redes Neurais	26
3.2.1	Fases no Processo de Simulação de Redes Neurais	26
3.2.2	Recursos Adicionais para complementar uma Simulação de Redes Neurais	27
3.3	Stuttgart Neural Network Simulator	29
3.4	Open Simulator	30

3.5	Sesame	32
3.6	Conclusões	33
4	Kipu, Um Simulador de Redes Neurais Construtivas	34
4.1	Introdução	34
4.2	Interface Gráfica do Usuário	35
4.2.1	Criação de Redes	36
4.2.2	Controle dos Parâmetros da Simulação	39
4.2.3	Monitoramento da Simulação	40
4.3	Implementação do Simulador Kipu	42
4.3.1	Estrutura de Dados	43
4.3.2	Geração e Adição de Novos Algoritmos	45
4.3.3	Algoritmos Implementados no Simulador Kipu	50
4.4	Utilitários	58
4.4.1	Módulo de Geração de Arquivos de Padrões	58
4.4.2	Módulo para a Visualização de Padrões	59
4.5	Conclusões	59
5	Experimentos Utilizando o Simulador Kipu	61
5.1	Introdução	61
5.2	Experimentos com Dados da Marinha Brasileira	62
5.2.1	Treinamento com os Dados de Navios da Marinha Brasileira	64
5.2.2	Validação com os Dados da Marinha Brasileira	65
5.2.3	Teste com os Dados da Marinha Brasileira	66
5.3	Experimentos com os Dados da Marinha Americana	67
5.3.1	Treinamento com os Dados da Marinha Americana	69
5.3.2	Validação com os Dados da Marinha Americana	70
5.3.3	Teste com os Dados da Marinha Americana	71
5.4	Experimentos com Dados de Silhuetas de Veículos	72
5.4.1	Treinamento com os Dados de Silhuetas de Veículos	74

5.4.2	Validação com os Dados de Silhuetas de Veículos	75
5.4.3	Teste com os Dados de Silhuetas de Veículos	76
5.5	Conclusões	77
6	Conclusões	79
6.1	Contribuições	80
6.2	Dificuldades Encontradas	81
6.3	Trabalhos Futuros	82

Lista de Figuras

1.1	Instituições envolvidas no projeto SAPRI.	2
1.2	Sistema de Reconhecimento de Padrões.	4
1.3	Reconhecimento de uma letra utilizando uma RNA.	5
2.1	Estado inicial da Rede <i>Cascade Correlation</i>	11
2.2	Rede <i>Cascade Correlation</i> depois de adicionar uma unidade.	11
2.3	Rede <i>Cascade Correlation</i> depois da segunda adição.	12
2.4	Rede <i>Cascade Correlation</i> após a inserção de três novos neurônios.	12
2.5	Rede <i>UpStart</i> antes de começar o processo de treinamento.	14
2.6	Rede <i>UpStart</i> depois da inserção da primeira e da segunda unidade adicional.	14
2.7	Rede <i>UpStart</i> depois da inserção da terceira e quarta unidade adicional.	15
2.8	Rede MTiling antes de inserir novas unidades.	17
2.9	Rede MTiling depois de inserir novas unidades na primeira camada intermediária	17
2.10	Rede MTiling depois de inserir novas unidades na primeira e na segunda camada intermediária	18
2.11	Rede <i>Tower</i> antes de começar o treinamento.	19
2.12	Rede <i>Tower</i> depois de inserir uma camada.	19
2.13	Rede <i>Tower</i> após a inserção de uma segunda camada.	20
2.14	Rede <i>Pyramid</i> antes de começar o treinamento.	21
2.15	Rede <i>Pyramid</i> após a inserção de uma camada.	21
2.16	Rede <i>Pyramid</i> após a inserção de uma segunda camada.	22
2.17	Estrutura de uma rede RBF.	24
2.18	Função de ativação para as unidades RBF.	24

4.1	Desenho Visual de uma Rede Neural utilizando o simulador Kipu.	36
4.2	Diálogo de propriedades de uma Rede Neural.	37
4.3	Diálogo de propriedades de uma camada.	38
4.4	Diálogo de propriedades de um neurônio.	39
4.5	Entrada de dados utilizando um diálogo contido em uma DLL.	40
4.6	Curva de erro durante o treinamento de uma RNA.	41
4.7	Resultados obtidos durante o treinamento de uma RNA e no teste.	42
4.8	Estrutura de dados do simulador Kipu.	45
4.9	Membros importáveis e exportáveis declarados dentro de uma DLL.	46
4.10	Hierarquia de classes formada por diversos algoritmos.	47
4.11	Modelo de um arquivo <i>Template</i> utilizado para geração de código.	48
4.12	Diálogo utilizado para a criação de um novo algoritmo.	49
4.13	Diálogo utilizado para a criação de um novo algoritmo.	49
4.14	Utilitário para a geração de arquivos de padrões.	58
4.15	Utilitário para a visualização de padrões.	59
5.1	Imagens de quatro tipos de navios enviados pela Marinha.	62
5.2	Taxa de acerto durante a fase de treinamento utilizando os Dados de Navios da Marinha Brasileira.	65
5.3	Taxa de acerto durante a fase de Validação utilizando os Dados de Navios da Marinha Brasileira.	66
5.4	Taxa de acerto durante a fase de Teste utilizando os Dados de Navios da Marinha Brasileira.	67
5.5	Taxa de acerto durante a fase de Treinamento utilizando os Dados da Marinha Americana.	70
5.6	Taxa de acerto durante a fase de Validação utilizando os Dados da Marinha Americana.	71
5.7	Taxa de acerto durante a fase de Teste utilizando os Dados da Marinha Americana.	72
5.8	Taxa de acerto durante a fase de Treinamento com os Dados de Silhuetas de Veículos.	75
5.9	Taxa de acerto durante a fase de Validação com os Dados de Silhuetas de Veículos.	76

5.10 Taxa de acerto durante a fase de Teste com os Dados de Silhuetas de Veículos. 77

Lista de Tabelas

5.1	Dados da Marinha Brasileira processados com o algoritmo <i>BackPropagation</i> .	63
5.2	Dados da Marinha Brasileira processados com o algoritmo <i>Cascade Correlation</i> .	63
5.3	Dados da Marinha Brasileira processados com o algoritmo <i>Tower</i>	63
5.4	Dados da Marinha Brasileira processados com o algoritmo <i>Pyramid</i>	64
5.5	Taxa de acerto durante o treinamento utilizando os Dados de Navios da Marinha Brasileira.	64
5.6	Taxa de acerto durante a Validação utilizando os Dados de Navios da Marinha Brasileira.	65
5.7	Taxa de acerto durante o Teste utilizando os Dados de Navios da Marinha Brasileira.	66
5.8	Dados da Marinha Americana processados com o algoritmo <i>BackPropagation</i> .	68
5.9	Dados da Marinha Americana processados com o algoritmo <i>Cascade Correlation</i> .	68
5.10	Dados da Marinha Americana processados com o algoritmo <i>Tower</i>	68
5.11	Dados da Marinha Americana processados com o algoritmo <i>Pyramid</i>	69
5.12	Taxa de acerto durante a fase Treinamento utilizando os Dados da Marinha Americana.	69
5.13	Taxa de acerto durante a fase Validação utilizando os Dados da Marinha Americana.	70
5.14	Taxa de acerto durante a fase de Teste utilizando os Dados da Marinha Americana.	71
5.15	Dados de Silhuetas de Veículos processados com o algoritmo <i>BackPropagation</i> .	73
5.16	Dados de Silhuetas de Veículos processados com o algoritmo <i>Cascade Correlation</i>	73
5.17	Dados de Silhuetas de Veículos processados com o algoritmo <i>Tower</i>	73
5.18	Dados de Silhuetas de Veículos processados com o algoritmo <i>Pyramid</i>	74

5.19	Taxa de acerto durante o Treinamento utilizando os Dados de Silhuetas de Veículos.	74
5.20	Taxa de acerto durante o Validação utilizando os Dados de Silhuetas de Veículos.	75
5.21	Taxa de acerto durante o Teste utilizando os Dados de Silhuetas de Veículos.	76

Capítulo 1

Introdução

Este trabalho diz respeito ao projeto e desenvolvimento de um simulador de Redes Neurais Construtivas para o Reconhecimento de Padrões. Este trabalho faz parte do projeto SAPRI (Sistema para Aquisição, Processamento e Reconhecimento de Imagens) que está sendo desenvolvido para a Marinha do Brasil. O projeto SAPRI é eminentemente estratégico e por isso requer especial cuidado e atenção. Os objetivos deste projeto serão explicados a seguir. Mais detalhes podem ser encontrados em [SAPRI 97].

O projeto SAPRI tem por objetivo o tratamento de imagens de radar, fornecendo a especificação de um ambiente voltado para a segurança da navegação e para o controle de tráfego aeronaval. O sistema será empregado, particularmente, em tarefas de controle tático e no auxílio à verificação de regras de segurança de navegação. No tocante ao aspecto tecnológico, o projeto visa a elaboração de um ambiente computacional eficiente, que integre todas as fases envolvidas no processamento automatizado de imagens de radar, desde a fase inicial de aquisição de imagens até o processo final de tomada de decisões. O sistema beneficiará em grande parte a Marinha do Brasil, que utilizará o sistema no aparelhamento de suas embarcações. Um dos benefícios desse sistema consiste em tornar o processo de monitoramento do ambiente menos dependente da experiência, sensibilidade e do nível de atenção dos operadores humanos, os quais são suscetíveis a flutuações decorrentes dos estados de cansaço e *stress*.

As pesquisas desenvolvidas para a execução do projeto envolvem estudos principalmente nas áreas de Redes Neurais Artificiais (RNAs), Processamento de Imagens, Reconhecimento e Classificação de Padrões, sendo as tarefas divididas entre o Instituto de Pesquisas da Marinha do Brasil, Instituto de Computação da UNICAMP, Laboratório de Computação Inteligente da UFPE, Laboratório de Inteligência Computacional do ICMC/USP - São Carlos, Escola de Engenharia Elétrica da UFG e Instituto de Informática da UFRGS. Cada instituição tem uma tarefa específica como pode ser observado na Figura 1.1

<p><u>Instituto de Pesquisas</u> <u>Marinha do Brasil</u></p> <p>Captura das Imagens através de Radar.</p>
<p><u>Instituto de Computação</u> <u>Universidade de Campinas</u></p> <p>Pré-processamento das Imagens, eliminação de ruído.</p>
<p><u>Departamento de Informática</u> <u>Laboratório de Computação Inteligente</u> <u>Universidade Federal de Pernambuco</u></p> <p>Extração de características das imagens; Reconhecimento de Padrões.</p>
<p><u>Instituto de Ciências Matemáticas de São Carlos</u> <u>Universidade de São Paulo - São Carlos</u></p> <p>Reconhecimento de Padrões.</p> <ul style="list-style-type: none"> • Combinação de estimadores (Paulo Prampero); • Projeto Evolucionário de Redes Neurais (Elson Mendes); • Reconhecimento de Alvos utilizando Redes Neurais Construtivas (Ernesto Cuadros Vargas).
<p><u>Escola de Engenharia Elétrica - Grupo Pirineus</u> <u>Universidade Federal de Goiás</u></p> <p>Reconhecimento de Padrões.</p>
<p><u>Instituto de Informática</u> <u>Universidade Federal do Rio Grande do Sul</u></p> <p>Análise tática.</p>

Figura 1.1: Instituições envolvidas no projeto SAPRI.

Como pode-se observar na Figura 1.1, o ICMC encontra-se entre as instituições encarregadas de desenvolver o módulo de Reconhecimento de Padrões. Para poder alcançar esse objetivo, diversos aspectos estão sendo investigados, tais como: Combinação de Estimadores, Otimização de Redes Neurais através de Algoritmos Genéticos e este trabalho, que se refere à utilização de Redes Neurais Construtivas (RNC) ¹ para o Reconhecimento de Alvos.

A definição da topologia de uma Rede Neural é um dos principais aspectos a serem considerados quando da sua utilização. A alternativa mais comum envolve a utilização de Redes Neurais com topologia fixa. Entretanto, no caso de Reconhecimento de Alvos, é interessante que a rede possa continuar aprendendo novos padrões, mesmo depois de finalizado o processo de treinamento. As RNC oferecem a possibilidade de adaptar automaticamente a sua arquitetura à inclusão de novos padrões. Esse foi o principal motivo que deu início a este trabalho de pesquisa. Existem diversos modelos de RNC que serão apresentados com mais detalhes no Capítulo 2.

O passo inicial deste trabalho envolveu a procura de simuladores de Redes Neurais que permitissem o desenvolvimento e uso de Redes Construtivas. Atualmente existem diversos simuladores de Redes Neurais, e alguns dos mais conhecidos serão apresentados no Capítulo 3. Mesmo sendo bastante úteis, só alguns deles (SNNS, Seção 3.3) têm implementados algoritmos construtivos (*Cascade Correlation* e *RBF-DDA*). A pequena quantidade de algoritmos construtivos implementados não é suficiente para obter conclusões em relação ao comportamento das RNC, desta limitação nasceu a idéia de criar um simulador que fosse adequado para o trabalho com esse tipo de rede. O Capítulo 4 apresenta o simulador Kipu, que foi projetado e implementado visando o desenvolvimento de algoritmos construtivos. O simulador Kipu constitui a principal contribuição deste trabalho de pesquisa. É dentro desse contexto que esta dissertação foi realizada.

Uma vez que o principal objetivo deste trabalho é o Reconhecimento de Alvos através de RNC, a seguir, na Seção 1.1, será apresentado em maiores detalhes o processo de Reconhecimento de Padrões. Nessa seção, também serão apresentadas as operações que compõem um Sistema típico de Reconhecimento de Padrões.

1.1 Reconhecimento de Padrões

Quando o ser humano "Re-conhece" algum objeto, significa que ele está relacionando-o com alguma idéia que previamente foi armazenada na memória, o que significa que está "Re-Identificando" aquele objeto.

Em termos gerais, Classificação de Padrões é a área que compreende a identificação ou classificação de medidas de informação em categorias. Uma categoria tem a característica de representar entidades ou padrões de informação que apresentam similaridades.

O ato de reconhecer padrões é uma atividade tão natural no dia a dia do ser humano, que

¹Uma Rede Neural Construtiva é uma Rede Neural Artificial que é treinada com um Algoritmo Construtivo.

muitas vezes nem é percebido. O simples fato de caminhar implica que as pessoas devem reconhecer os diversos obstáculos que vão aparecendo para poder evitá-los. Os olhos e os ouvidos são os principais órgãos que fornecem ao cérebro informações a respeito do meio ambiente. Mesmo sendo uma tarefa muito fácil para as pessoas, é muito complicado obter de um computador um desempenho que seja semelhante àquele obtido por seres humanos. Embora a tarefa de reconhecer padrões seja mais difícil de ser realizada usando computadores, espera-se que algum dia os computadores possam reconhecer rostos, imagens, vozes e sons tão bem quanto os seres humanos.

Um sistema genérico de Reconhecimento de Padrões apresenta um conjunto de operações que devem ser realizadas como pode ser observado na Figura 1.2.

Figura 1.2: Sistema de Reconhecimento de Padrões.

Em primeiro lugar, deve existir algum mecanismo para capturar do mundo externo os padrões a serem processados. Isto pode ser realizado utilizando algum aparelho apropriado, por exemplo, um radar para a captura de imagens. Uma vez que as imagens foram capturadas, elas devem ser pré-processadas, o que envolve a eliminação de problemas, tais como o ruído contido nas imagens. O próximo passo consiste na extração das características mais importantes dos padrões. Esse último ponto é muito importante, pois simplifica muito o processamento do classificador, o que implica em menos tempo para atingir um nível de desempenho adequado. Finalmente, podem ser classificados os padrões. Os três últimos módulos: Pré-processamento, Extração de Características e Classificação de Padrões, podem ser implementados por Redes Neurais. Este trabalho está relacionado ao problema de Classificação de Padrões.

Existem diversas abordagens relacionadas às operações de reconhecer padrões, que podem ser usadas de forma integrada na solução de diversos problemas práticos. Entre essas abordagens pode-se destacar o Reconhecimento de Padrões Estatístico, o Reconhecimento de Padrões Sintático e as RNAs que é a abordagem utilizada neste trabalho.

Para que uma RNA possa ser utilizada como classificador, ela deve ser treinada de forma adequada utilizando um algoritmo apropriado. Depois do processo de treinamento, a rede pode ser utilizada como um classificador dentro de um sistema maior de Reconhecimento de Padrões.

Um exemplo clássico de Reconhecimento de Padrões com RNAs que pode ajudar a compreender esse processo é o reconhecimento de imagens de letras formadas por *pixels*. Nesse caso, a entrada da rede é composta por cada um dos pontos da letra a ser reconhecida, e

a saída é a letra com a qual o padrão está sendo associado. O aprendizado ocorre com a apresentação de um conjunto de letras pertencentes a cada uma das categorias existentes. Quando uma nova letra é apresentada durante a fase de teste, a rede a classifica na categoria com a qual tenha mais semelhança. Na Figura 1.3 é apresentado um exemplo de reconhecimento da letra I considerando algum ruído na imagem.

Figura 1.3: Reconhecimento de uma letra utilizando uma RNA.

1.2 Objetivos Referentes à Classificação de Padrões

O objetivo deste projeto é o desenvolvimento de um simulador para RNC para poder realizar o Reconhecimento de Alvos. Um outro objetivo é melhorar o desempenho geral do módulo de reconhecimento de imagens do projeto SAPRI com a utilização de Redes Neurais que sejam mais flexíveis e dinâmicas na definição da estrutura utilizada.

É importante salientar que embora a motivação inicial para este trabalho tenha sido gerada pela necessidade de um sistema para a Marinha do Brasil, a tecnologia e os resultados científicos gerados com o mesmo também apresentam um grande valor potencial para outras áreas de aplicação que utilizam Sistemas de Processamento de Imagens e Classificação de Padrões.

Concluído o trabalho, a tecnologia desenvolvida será integralmente aproveitada pela Marinha do Brasil, refletindo-se em um sistema de controle tático a ser implantado em embarcações que sofram atualizações técnicas. A longo prazo, através do crescimento da interação, por decorrência do esquema de compartilhamento de responsabilidades entre os participantes do projeto, um dos mais valiosos resultados a ser colhido será a integração de instituições acadêmicas brasileiras, em projetos de pesquisa aplicados à área de defesa, atividade geralmente restrita às instituições militares.

1.3 Organização do Trabalho

Este trabalho está organizado da seguinte maneira: no Capítulo 2 serão apresentadas as RNC, as quais têm a propriedade de inserir elementos na rede durante o seu processo de treinamento. Serão vistos nesse capítulo, alguns Algoritmos Construtivos, tais como *UpStart*, *Tiling*, *Tower*, *Pyramid*, *Cascade Correlation* e *RBF-DDA*.

No Capítulo 3 será apresentado um resumo de alguns simuladores de Redes Neurais existentes. Para cada simulador, serão apresentadas as características mais importantes junto com suas vantagens e desvantagens.

O Capítulo 4 refere-se à modelagem e implementação do simulador Kipu. Esse simulador foi criado pensando na flexibilidade necessária para o desenvolvimento, simulação e análise de RNC. Nesse mesmo capítulo serão apresentados e analisados os pontos chaves para a construção de um completo simulador de Redes Neurais. Finalmente, será apresentada a estrutura de dados utilizada pelo simulador proposto, a qual tenta ser a mais flexível possível para facilitar o trabalho com RNC.

No Capítulo 5 serão apresentados alguns experimentos realizados utilizando o simulador *Kipu*. Nesse mesmo capítulo também serão analisados os resultados obtidos utilizando Redes Neurais Cconvencionais frente aos obtidos utilizando RNC.

Finalmente, no Capítulo 6, serão apresentadas as conclusões do trabalho.

Capítulo 2

Redes Neurais Construtivas

2.1 Introdução

Um dos fatores que mais influenciam o desempenho de uma Rede Neural é sua Arquitetura. Várias técnicas têm sido desenvolvidas para a otimização dessa arquitetura, sendo que essas técnicas podem ser agrupadas em quatro abordagens: Empírica, Genética, Construtiva e *Pruning*.

A primeira abordagem, a Empírica, é a mais utilizada, e está baseada no método conhecido como Tentativa e Erro. Dito de outra forma, quando o usuário deseja resolver um determinado problema, ele escolhe uma arquitetura para sua RNA segundo o seu próprio critério, especialmente segundo sua experiência com outros problemas similares. Se a RNA escolhida não alcança o nível de desempenho esperado pelo usuário, ele muda alguns de seus parâmetros e refaz os experimentos. Essa operação se repete até que o usuário encontre a arquitetura mais adequada para o seu problema. Obviamente, o principal inconveniente dessa abordagem é a necessidade de treinar várias RNAs antes de encontrar aquela mais adequada para um problema específico. A busca da melhor arquitetura pode levar à obtenção de uma rede com um bom desempenho, às custas de um aumento considerável no tempo de processamento. Quanto mais aspectos são considerados na otimização, mais rapidamente pode-se chegar a uma insuficiência de recursos computacionais para uma busca completa. Apesar desse problema, esse é o procedimento mais utilizado atualmente.

A segunda abordagem é a Genética, na qual inicialmente existe um conjunto de RNAs candidatas para resolver um determinado problema. Se nenhuma delas alcança um desempenho que seja satisfatório para o usuário, um novo conjunto de RNAs é criado combinando as características das RNAs que obtiveram os melhores desempenhos do grupo ou geração anterior, sendo essa combinação realizada por operadores genéticos. Esse processo é repetido até encontrar uma rede que alcance os níveis de acerto que o usuário deseja.

A terceira abordagem é conhecida como *Pruning*, que procura melhorar o desempenho das RNA, através da eliminação de elementos redundantes ou pouco significativos. Existem várias técnicas de *Pruning*, que permitem determinar quais são as unidades que devem ser

removidas da rede. Entre os algoritmos utilizados para determinar se uma unidade deve ser removida destacam-se os seguintes: *Skeletonization* proposto por Mozer e Smolensky em [Mozer 89], o método de *Karnin* [Karnin 90], e o Algoritmo *lprune* proposto por Prechelt [Prechelt 95]. Um método de *Pruning* geralmente procura otimizar o funcionamento de uma RNA já treinada. Para alcançar esse objetivo, os elementos que não forem relevantes para o funcionamento da RNA podem ser eliminados.

A quarta e última abordagem é a Construtiva, que será mais detalhada por ser o enfoque deste trabalho. A abordagem Construtiva é uma alternativa para a construção incremental da arquitetura de RNAs. Geralmente, este tipo de RNAs são inicializadas com poucas unidades. Durante o treinamento novas unidades são adicionadas até a obtenção de um adequado nível de generalização. Um exemplo prático de algoritmo construtivo são as redes *Cascade Correlation* [Fahlman 90].

A seguir, na Seção 2.2, podem ser observados alguns critérios utilizados para classificar as Redes Neurais Construtivas. Logo após serão apresentados também alguns modelos de RNC.

2.2 Taxonomia das Redes Neurais Construtivas

As Redes Neurais Artificiais (RNAs) de forma geral, podem ser classificadas segundo vários critérios. As RNA podem ser constituídas por uma única camada ou por muitas camadas de neurônios. Essas últimas são também conhecidas como **Redes Multi-camadas**.

Um outro critério que pode ajudar na classificação de modelos de RNA diz respeito ao paradigma utilizado para o seu treinamento. Dois paradigmas tem sido utilizados para o treinamento construtivo: supervisionado e não supervisionado. O aprendizado supervisionado consiste em apresentar à rede os padrões de entrada junto com a resposta desejada. Exemplos de algoritmos supervisionados construtivos são os algoritmos *Cascade Correlation*, *Tower* e *Pyramid*. No aprendizado não supervisionado não são fornecidas as respostas desejadas para cada padrão. A rede agrupa os padrões utilizando alguma medida de similaridade. Como exemplos de tais algoritmos podem ser citados os algoritmos Kohonen construtivo [Halgamuge 97], SOFT [Carvalho 97] e SONN [Tenorio 89]. Neste trabalho serão tratados apenas algoritmos construtivos supervisionados.

As Redes Neurais Construtivas (RNC) são Redes Multi-camadas. Elas podem ser classificadas segundo o critério utilizado para a inserção de novas unidades. Algumas RNC procuram melhorar seu desempenho inserindo um neurônio por vez, isto ocorre com as redes *Cascade Correlation* (Seção 2.3), *UpStart* [Frean 89] (Seção 2.4), e *RBF-DDA* [Berthold 95] (Seção 2.8). As redes *Cascade Correlation* criam uma nova camada com um único neurônio por vez. Esta é a RNC mais conhecida e a mais utilizada dentro do grupo dos algoritmos construtivos. As redes *UpStart* e as *RBF-DDA* também inserem um neurônio por vez, mas todos os novos neurônios são inseridos em uma mesma camada.

Outros tipos de RNC permitem a inserção de mais de um neurônio por vez. Esse é o caso das redes *MTiling* (Seção 2.5) que inserem mais de um neurônio por vez na mesma camada. O número de neurônios inseridos é definido de acordo com o número de padrões que geraram

a mesma saída, embora pertençam a classes diferentes.

Existem também aqueles algoritmos que criam uma camada por vez. Se enquadram nesse caso o algoritmo *Tower* [Parekh 95] (Seção 2.6) e o algoritmo *Pyramid* (Seção 2.7). Nessas redes, a nova camada passa a ser a saída da rede. A diferença entre elas está na forma que a nova camada é conectada. No algoritmo *Tower*, a nova camada é conectada à camada de entrada e à camada anterior, se houver. O algoritmo *Pyramid* além de conectar a nova camada à camada de entrada e à camada anterior, também a conecta a todas as outras camadas anteriores. Outros trabalhos referentes a RNC podem ser encontrados em [Berthold 95, Fritzke 95a, Fritzke 95b].

A seguir serão descritos os algoritmos anteriormente mencionados, enfatizando suas características particulares, assim como as vantagens e desvantagens que apresentam.

2.3 Algoritmo Cascade Correlation

O *Cascade Correlation* é um algoritmo de aprendizado supervisionado [Fahlman 90]. O *Cascade Correlation* combina duas idéias chaves: a primeira diz respeito a utilização de uma arquitetura em cascata, na qual novas unidades são inseridas à rede, uma por vez. A segunda idéia é o emprego de um algoritmo que cria e insere as novas unidades na rede. Esse algoritmo foi proposto para superar limitações apresentadas pelo algoritmo *BackPropagation*, especialmente a sua lentidão.

Inicialmente, uma rede *Cascade Correlation* é composta por duas camadas: uma camada de entrada e uma camada de saída. O número de unidades em ambas as camadas será determinado por cada problema. Cada unidade da camada de entrada deve ser conectada a cada unidade da camada de saída.

Como já foi dito, a rede *Cascade Correlation* inicial não possui camadas intermediárias. As conexões entre a camada de entrada e a camada de saída são treinadas com todos os padrões do conjunto de treinamento. Esse ajuste dos pesos é realizado até que o desempenho da rede não melhore mais. Finalizada essa etapa, é necessária mais uma apresentação de todo o conjunto de treinamento para poder calcular o erro do ciclo¹. Se o erro for aceitável o treinamento deve parar. Senão, existe um erro residual que deve ser reduzido ainda mais.

O processo de criação de uma nova unidade na rede começa pela criação de uma Camada de Candidatos. A camada recebe conexões da camada de entrada que podem ser treinadas. As saídas dos neurônios da camada de candidatos ainda não são conectadas a nenhum outro elemento da rede. A rede é treinada ajustando em cada ciclo os pesos das conexões de entrada dos neurônios da camada de candidatos. O objetivo desses ajustes é maximizar S (Ver Equação 2.1, que é a soma das correlações entre V (valor de saída de um neurônio candidato) e E_o (erro residual do o -ésimo neurônio da camada de saída). O valor de S é

¹Sem a existência de camadas intermediárias, os pesos das conexões podem ser ajustados utilizando a Regra Delta, o algoritmo *Perceptron*, *QuickProp* ou qualquer outro algoritmo para RNAs. Sem camadas intermediárias, o algoritmo *QuickProp* funciona como a Regra Delta, com a única diferença de que converge com mais rapidez.

calculado de acordo com a seguinte formula:

$$S = \sum_0 \left| \sum_p (V_p - \bar{V})(E_{p,o} - \bar{E}_o) \right| \quad (2.1)$$

Na formula anterior, o representa o número do neurônio da camada de saída, e p é o número de padrão que é apresentado. Os valores $\bar{V}$ e $\bar{E}_o$ são as médias de V e E_o para todos os padrões. Para maximizar S , é necessário calcular $\delta S/\delta w_i$, que é a derivada parcial de S com relação aos pesos de cada conexão da camada de candidatos w_i , como pode ser observado a seguir na fórmula 2.2.

$$\delta S/\delta w_i = \sum_{p,o} \text{Sgn}(E_{p,o} - \bar{E}_o) f'_p I_{i,p} \quad (2.2)$$

na fórmula anterior, Sgn representa o valor da correlação entre o valor de ativação do candidato e a o -ésima saída da rede. f'_p é a derivada (para o padrão p), da função de ativação do candidato, em relação à soma de suas entradas. Finalmente, $I_{i,p}$ é a entrada que o candidato recebe da unidade de entrada i , para o padrão p .

Depois de ter calculado o valor de $\delta S/\delta w_i$ para cada conexão de entrada, S pode ser maximizada utilizando um gradiente ascendente. Mais uma vez é realizado o ajuste de pesos unicamente para as conexões que vêm da camada de entrada. Quando o valor de S não mudar mais, uma cópia da unidade candidata com a maior correlação é inserida na rede. As conexões de entrada de essa nova unidade já foram ajustadas, portanto são congeladas a partir desse momento. A nova unidade é conectada à camada de saída com os pesos das conexões entre ela e a camada de saída, inicializados de forma aleatória. Unicamente as conexões que chegam à camada de saída continuam sendo treinadas. Cada uma das unidades inseridas forma então uma nova camada intermediária com uma única unidade. A inserção de novas unidades intermediárias vai dando à rede uma aparência de cascata e é essa a razão do seu nome.

Entre as vantagens das redes *Cascade Correlation* está a rapidez do seu treinamento. Essa rapidez se deve ao fato de que as conexões são congeladas depois de serem treinadas, e os únicos pesos que são ajustados posteriormente são aqueles que chegam à camada de saída. Na Figura 2.1 é apresentada uma rede *Cascade Correlation* no seu estado inicial. As Figuras 2.1, 2.2 e 2.3 ilustram o crescimento da rede devido à inserção de novas unidades intermediárias.

Figura 2.1: Estado inicial da Rede *Cascade Correlation*.

Figura 2.2: Rede *Cascade Correlation* depois de adicionar uma unidade.

Figura 2.3: Rede *Cascade Correlation* depois da segunda adição.

Figura 2.4: Rede *Cascade Correlation* após a inserção de três novos neurônios.

Figura 2.3: Rede *Cascade Correlation* depois da segunda adição.

Figura 2.4: Rede *Cascade Correlation* após a inserção de três novos neurônios.

Segundo experimentos realizados pelos próprios criadores do algoritmo, o mesmo aprende rapidamente a diferenciar os padrões. Os passos realizados pelo algoritmo *Cascade Correlation* são:

1. Inicializar a rede com o número de entradas e saídas requeridas;
2. Treinar só com a camada de entrada conectada à camada de saída até que o erro não melhore mais;
3. Se o erro não for aceitável, então:
 - 3.1. Criar uma camada de candidatos que recebe conexões da camada de entrada e treinar os pesos das conexões de entrada até que o erro não melhore mais;
 - 3.2 Se o erro não for aceitável, repetir:
 - 3.2.1. Calcular a matriz de correlação entre a saída gerada pelos neurônios da camada de candidatos e os erros dos neurônios da camada de saída;
 - 3.2.2. Inserir na rede uma cópia do neurônio que tiver a melhor correlação. Os pesos de entrada do novo neurônio são congelados, e as suas conexões para a camada de saída são inicializadas com valores aleatórios;
 - 3.2.3. Treinar as conexões que chegam à camada de saída da rede até que o erro não varie mais, e voltar ao passo 3.2.

Deve ser observado que a cada inserção de uma nova unidade a rede fica mais lenta. Considerando que cada nova unidade possui conexões que se originam na camada de entrada e em todas as unidades inseridas antes dela, a quantidade de operações por ciclo tem um crescimento do tipo exponencial. Por causa disso, não é conveniente que a rede insira um grande número de novas unidades.

2.4 Algoritmo UpStart

Um outro algoritmo construtivo interessante é o algoritmo *UpStart* desenvolvido por Marcus Frean [Frean 89]. Esse algoritmo se caracteriza pelas unidades de entrada assumirem valores discretos $\{-1, +1\}$.

Inicialmente, a rede *UpStart* é formada por uma camada de entrada e uma camada de saída. Todos os neurônios da rede utilizam a função de saída *Threshold*. As conexões entre essas camadas podem ser treinadas com o algoritmo *Perceptron* [Rumelhart 86] até que o desempenho da rede não melhore mais. Se os padrões apresentados são linearmente separáveis, a rede conseguirá convergir sem problemas. Caso contrário, os pesos das conexões nunca irão se estabilizar. No entanto, uma simples extensão deste algoritmo denominada *Pocket* [Gallant 93] pode solucionar esse problema. Esse algoritmo consiste em utilizar o algoritmo *Perceptron* e apresentar o conjunto de padrões de forma aleatória à rede. Ao mesmo tempo deve ser mantida uma cópia do maior conjunto de pesos que não tenha registrado mudanças. Esse conjunto de pesos ajuda a minimizar o erro da saída da rede.

Uma unidade na camada de saída da rede pode gerar erros de dois tipos: *Wrongly ON* e *Wrongly OFF*. O primeiro tipo de erro, o *Wrongly ON*, ocorre quando uma unidade gera a saída com o valor 1, sendo que deveria ter gerado o valor 0. O segundo tipo de erro, o *Wrongly OFF*, é o contrário ao primeiro caso. Neste caso a unidade gera o valor 0 quando deveria gerar o valor 1.

Seja uma unidade pertencente à camada de saída que gera um erro do tipo *Wrongly ON*, para um subconjunto de padrões C . Um novo peso altamente negativo poderia ser criado para corrigir o erro dessa unidade. Para criar esse novo peso, uma nova unidade é criada na camada intermediária. A camada de entrada é conectada à nova unidade, que por sua vez é conectada à camada de saída da rede. Os pesos das novas conexões são inicializados de forma aleatória, e devem ser treinados com o subconjunto C antes mencionado. Desta forma os erros gerados pelas unidades da camada de saída são corrigidos. O oposto acontece quando uma unidade da camada de saída gera um erro do tipo *Wrongly OFF*. Na Figura 2.5 é apresentada uma pequena rede *UpStart* antes da inserção de novas unidades, e nas Figuras 2.6 e 2.7 pode ser observado o crescimento da rede devido à inserção de novas unidades.

Figura 2.5: Rede *UpStart* antes de começar o processo de treinamento.

Figura 2.6: Rede *UpStart* depois da inserção da primeira e da segunda unidade adicional.

Figura 2.7: Rede *UpStart* depois da inserção da terceira e quarta unidade adicional.

Como pode ser observado nas Figuras 2.6 e 2.7, as redes *UpStart* possuem apenas uma camada intermediária onde as novas unidades são inseridas.

O algoritmo *UpStart* funciona da seguinte forma:

1. Treinar a rede inicial (camada de entrada conectada à camada de saída) com o algoritmo *Perceptron* até obter um bom desempenho;
2. Para cada unidade O_i camada de saída, repetir:
 - 2.1. Detectar o subconjunto de padrões S , que fazem que O_i gere um erro do tipo *Wrongly ON*.
 - 2.2. Se o conjunto S for não vazio, então:
 - 2.2.1. Criar uma unidade na camada intermediária;
 - 2.2.2. Conectar todos os neurônios da camada de entrada à nova unidade;
 - 2.2.3. Conectar a nova unidade à camada de saída;
 - 2.2.4. Treinar os novos pesos com o conjunto de dados S , até que o desempenho da rede não melhore mais;
 - 2.3. Repetir 2.1, e 2.2 para as unidades que gerem erro do tipo *Wrongly OFF*.
3. Se o erro produzido pela rede ainda não for aceitável, voltar ao passo 2.

2.5 Algoritmo Tiling e MTiling

Mezard e Nadal propuseram um algoritmo construtivo chamado *Tiling* [Mézard 89]. Os criadores do algoritmo conseguiram provar que ele converge para nenhum erro de classificação

quando o conjunto de padrões utilizado pode ser separado em duas categorias. Entretanto em muitos problemas práticos, os padrões podem pertencer a mais de duas classes ($M > 2$). Para lidar com essas situações, surgiu uma variação do algoritmo *Tiling* para mais de duas classes. Essa variação é chamada de *MTiling* [Parekh 95]. O algoritmo *MTiling* pode sofrer algumas modificações para melhorar o seu desempenho. Algumas dessas variações são: o algoritmo *Pocket* [Gallant 93], *Thermal perceptron* [Frean 90] e *Barycentric Correction Procedure* [Poulard 95].

A camada de saída possui inicialmente M neurônios (M é o número de classes) chamados de Neurônios Mestres. Se com esses M neurônios a rede for incapaz de classificar corretamente os padrões de entrada, Neurônios Auxiliares são inseridos na camada de saída. Dessa forma pode-se garantir uma representação fiel dos padrões do conjunto de treinamento. Depois de inserir os Neurônios Auxiliares na camada de saída atual, ela deixa de ser a saída da rede, e se torna a última camada intermediária da rede. Uma nova camada de saída é criada, com M Neurônios Mestres, e a antiga camada de saída é conectada a ela. O critério de fidelidade diz que dois padrões pertencentes a classes diferentes não podem produzir saídas idênticas em uma mesma camada. Segundo os criadores do algoritmo, a fidelidade é uma condição necessária para a convergência da rede para uma taxa de erro de classificação igual a zero.

Para o treinamento de uma rede segundo o algoritmo *MTiling*, devem ser considerados os seguintes passos:

1. Treinar uma rede com M Neurônios Mestres na camada da saída. Cada um desses neurônios deve estar conectado às N entradas;
2. Enquanto a camada de saída não cumprir com o critério de fidelidade, repetir:
 - 2.1. Entre todas as saídas não fiéis na camada de saída, identificar aquela que possua o maior número de padrões associado a ela;
 - 2.2. Adicionar um grupo de k ($1 \leq k \leq M$) Neurônios Auxiliares à camada de saída (k é o número de classes representadas no conjunto de padrões identificado em 2.1);
 - 2.3. Criar uma nova camada de saída com M Neurônios Mestres;
 - 2.4. Conectar a antiga camada de saída à nova camada criada no passo anterior;
 - 2.4. Treinar os novos pesos com os padrões identificados em 2.1.

A convergência do Algoritmo *MTiling* para uma taxa de erro igual a zero pode ser provada em duas partes. Primeiro, é provado que uma representação fiel pode ser atingida por qualquer conjunto de dados finito e não contraditório utilizando um número finito de Neurônios Auxiliares. Logo é mostrado que com cada camada adicional, o número de erros de classificação é reduzido pelo menos em uma unidade por vez, mais detalhes podem ser encontrados em [Parekh 95].

Para ilustrar melhor o funcionamento do algoritmo *MTiling*, pode-se observar na Figura 2.8 uma rede *MTiling* no seu estado inicial. Neste exemplo existem sete entradas que devem ser

classificadas em três classes. Nas Figuras 2.9 e 2.10 pode se observar a mesma rede após a inserção de novas unidades nas camadas intermediárias, os **Neurônios Auxiliares**. Esses neurônios estão destacados na cor amarela. A inserção é realizada para garantir o critério de fidelidade em cada uma das camadas, e para eliminar os erros de classificação.

Figura 2.8: Rede MTiling antes de inserir novas unidades.

Figura 2.9: Rede MTiling depois de inserir novas unidades na primeira camada intermediária

Figura 2.10: Rede MTiling depois de inserir novas unidades na primeira e na segunda camada intermediária .

2.6 Algoritmo Tower

O algoritmo construtivo *Tower* foi desenvolvido por Gallant [Gallant 90]. Esse algoritmo constrói uma torre de **Unidades Lógicas Threshold** (ULTs). A camada que está na parte inferior da torre recebe N entradas, uma para cada componente do vetor de entrada. A torre é construída adicionando novas camadas à rede. Os neurônios adicionados são treinados utilizando os pesos das conexões existentes, até que a classificação desejada seja alcançada. Cada um dos neurônios adicionados vai representar uma nova saída e recebe como entrada cada um dos N componentes do padrão de entrada, além das entradas fornecidas pelos neurônios da camada anterior. Na Figura 2.11 é apresentada uma rede *Tower* antes de começar o treinamento. Na Figura 2.12 é mostrada a mesma rede após a inserção de uma nova camada. Na Figura 2.13 pode-se observar a mesma rede após a inserção de uma segunda camada.

Figura 2.11: Rede *Tower* antes de começar o treinamento.

Figura 2.12: Rede *Tower* depois de inserir uma camada.

Figura 2.13: Rede *Tower* após a inserção de uma segunda camada.

Supondo que as camadas estão numeradas a partir da camada número 1 (camada de entrada) até a camada número N (camada de saída), o funcionamento do algoritmo *Tower* pode ser descrito da seguinte maneira:

1. Inicializar L com 2 (camada de saída);
2. Enquanto o erro obtido pela rede não for aceitável, e o número máximo de camadas não for alcançado, repetir:
 - 2.1. Treinar a rede até que o erro não melhore mais;
 - 2.2. Se o erro obtido depois do treinamento ainda não é aceitável, então:
 - 2.2.1. $L = L + 1$;
 - 2.2.2. Adicionar uma camada com M unidades (M é o número de classes) na posição L ;
 - 2.2.3. Conectar a camada de entrada e a camada $L-1$ à camada L . Esta última passa a ser a nova saída da rede;
 - 2.2.4. Treinar os pesos associados a cada unidade da camada adicionada.

2.7 Algoritmo Pyramid

O algoritmo *Pyramid* [Gallant 90] constrói uma rede de maneira similar ao Algoritmo *Tower*, exceto que cada nova unidade adicionada é conectada à camada de entrada e a todas as camadas anteriores. Assim como na rede *Tower*, os neurônios da nova camada constituem a nova saída da rede. Na Figura 2.14 é apresentada uma rede *Pyramid* cuja arquitetura inicial é igual à de uma rede *Tower*. A Figura 2.15 também é similar à Figura 2.12. Contudo mas a diferença entre os dois algoritmos pode ser percebida comparando as Figuras 2.16 e 2.13 que mostram a diferença das conexões criadas pelo algoritmo *Pyramid*.

Figura 2.14: Rede *Pyramid* antes de começar o treinamento.

Figura 2.15: Rede *Pyramid* após a inserção de uma camada.

Figura 2.16: Rede *Pyramid* após a inserção de uma segunda camada.

Na Figura 2.16 observa-se que a camada **Escondida 1** está totalmente conectada à camada de saída, o que não acontece na rede *Tower*. O algoritmo *Pyramid* funciona de acordo com os seguintes passos:

1. Inicializar L com 2 (camada de saída);
2. Enquanto o erro obtido pela rede não for aceitável, e o número máximo de camadas não for alcançado, repetir:
 - 2.1. Treinar a rede até que o erro não melhore mais;
 - 2.2. Se o erro obtido depois do treinamento ainda não é aceitável, então:
 - 2.2.1. $L = L + 1$;
 - 2.2.2. Adicionar uma camada com M unidades (M é o número de classes) na posição L ;
 - 2.2.3. Conectar todas as camadas já existentes à camada L . Esta última passa a ser a nova saída da rede;
 - 2.2.4. Treinar os pesos associados a cada unidade da camada adicionada.

2.8 Algoritmo RBF-DDA

Um outro algoritmo construtivo é o *RBF-DDA* (*Radial Basis Function with Dynamic Decay Adjustment*) [Zell 95b]. Esse algoritmo é uma extensão do algoritmo RCE [Hudak 92, Reilly 82] e fornece uma forma rápida de treinamento para redes do tipo RBF [Moody 89]. Uma rede do tipo RBF treinada com o algoritmo DDA tem um desempenho comparável ao de uma rede do tipo MLP, mas o seu treinamento é muito mais rápido e simples [Berthold 95].

Uma rede RBF treinada com DDA (RBF-DDA) tem uma estrutura muito similar a uma rede RBF comum. A rede possui uma camada intermediária e não existem conexões entre neurônios de camadas não adjacentes.

Uma rede do tipo RBF apresenta as seguintes características:

- A camada intermediária contém unidades RBF. Essas unidades são inseridas durante o processo de treinamento. A camada de entrada é totalmente conectada com a camada intermediária;
- Cada unidade da saída representa uma classe.

Para a classificação é utilizado o critério *Winner Takes All*. Cada unidade da camada escondida é conectada a uma única unidade da camada de saída.

Uma das principais diferenças entre uma rede RBF e uma MLP está na camada intermediária. Ao invés de utilizar uma função do tipo sigmoide ou qualquer outra função não linear, uma rede RBF utiliza uma função *Gaussiana* como função de ativação. Assim ao invés de utilizar o produto escalar entre o vetor de pesos e o vetor da entradas, calcula-se a distância Euclidiana, como pode ser visto na Equação 2.3.

$$R_i(\vec{x}) = e^{-\frac{|\vec{x} - \vec{r}_i|^2}{\sigma_i^2}} \quad (2.3)$$

Se a rede recebe como entrada o vetor $\vec{x}$, R_i representa a ativação de uma unidade com referência ao vetor $\vec{r}_i$ e com um desvio padrão σ_i .

Na Figura 2.17 pode-se observar que o vetor de pesos que conecta todas as unidades de entrada à camada intermediária representa o centro Gaussiano. A distância Euclidiana do vetor de entrada ao vetor de pesos é usada como entrada para a função Gaussiana. Se o vetor de entrada está compreendido dentro do espaço que aquele neurônio abrange, este terá uma ativação alta. Em contraste, o neurônio poderia ficar quase sem ativação porque o padrão de entrada está muito distante do centro Gaussiano. Cada unidade de saída simplesmente calcula a soma de todas as ativações multiplicadas pelo peso para as unidades RBF da classe correspondente.

Figura 2.17: Estrutura de uma rede RBF.

O algoritmo DDA introduz a idéia de distinção entre os **vizinhos aceitáveis** e **vizinhos conflitantes** dentro de uma área de conflito. São utilizados dois *Thresholds* θ^+ e θ^- . Na Figura 2.18, observa-se que os dois *Thresholds* definem uma **Área de conflito** onde não é permitida a existência de nenhum outro padrão de uma classe conflitante. Cada padrão pertencente ao conjunto de treinamento deve estar compreendido no círculo central de pelo menos um neurônio da classe correta.

Figura 2.18: Função de ativação para as unidades RBF.

Geralmente θ^+ é maior do que θ^- , e ambos os valores determinam a área de conflito. Em resumo, as principais propriedades do algoritmo DDA são:

- **treinamento construtivo:** novas unidades RBF são inseridas quando for necessário. O número de unidades intermediárias é determinado pelo treinamento. Os raios de cada neurônio são ajustados dinamicamente durante a fase de treinamento;
- **treinamento rápido:** geralmente são necessários cerca de cinco ciclos para completar o treinamento, isto ocorre devido à natureza construtiva do algoritmo;

- convergência garantida: é possível demonstrar que o algoritmo DDA converge;
- dois parâmetros não críticos: só existem dois parâmetros θ^+ e θ^- que podem ser ajustados manualmente. Ambos os parâmetros não são críticos. Segundo os experimentos realizados pelos criadores do algoritmo $\theta^+ = 0,4$ e $\theta^- = 0,2$ são uma boa escolha;
- propriedades garantidas: já foi demonstrado que depois do treinamento a rede continua classificando corretamente os padrões do conjunto treinamento. Os padrões não classificados corretamente estarão abaixo de θ^- e os que foram bem classificados estarão acima de θ^+ .

2.9 Conclusões

Neste capítulo foram apresentados alguns modelos de Redes Neurais Construtivas (RNC), que têm como característica principal o fato de mudar a sua topologia inicial durante a fase de treinamento. A variação da topologia da rede significa adicionar novos elementos (camadas, neurônios ou conexões). A inserção de novos elementos é provocada pela incapacidade da rede de atingir um nível de desempenho adequado com sua topologia original.

Diversas técnicas são utilizadas para adicionar elementos a uma rede. Alguns algoritmos construtivos adicionam um neurônio por vez, como é o caso das redes *Cascade Correlation* (Ver Seção 2.3), *UpStart* (Ver Seção 2.4) e *RBF-DDA* (Ver Seção 2.8). No caso do algoritmo *Tiling* (Ver Seção 2.5) um grupo de neurônios são inseridos para melhorar o seu desempenho. Finalmente, os algoritmos *Tower* (Ver Seção 2.6) e *Pyramid* (Ver Seção 2.7) inserem camadas completas para melhorar o seu desempenho.

Capítulo 3

Simuladores de Redes Neurais

3.1 Introdução

Quando são utilizadas RNAs para resolver um problema prático, um simulador pode facilitar muito o trabalho do usuário, permitindo uma maior dedicação aos aspectos fundamentais da resolução do problema. O usuário geralmente procura um simulador que lhe forneça o maior número de ferramentas e que, ao mesmo tempo o auxilie a resolver o seu problema o mais rápido possível. Idealmente, um simulador deve auxiliar um usuário de Redes Neurais em todas as fases que conformam o processo de simulação de Redes Neurais. É por isso que antes de apresentar os simuladores mais conhecidos, é necessário conhecer quais são as fases que compõem um Processo de Simulação de RNAs [Murre 93], para depois, poder analisar as vantagens e desvantagens oferecidas por alguns dos simuladores existentes.

3.2 Processo de Simulação de Redes Neurais

Na simulação de Redes Neurais existem fases que precisam ser seguidas em uma dada ordem durante todo o processo. É no auxílio à execução dessas fases que os simuladores devem concentrar seus esforços e servir como uma ferramenta útil ao usuário final. Na prática, é muito difícil encontrar simuladores que consigam ser eficientes como ferramentas de apoio na totalidade das fases. A seguir são descritas essas fases e alguns recursos que podem ser oferecidos por um simulador para que ele possa ser considerado uma eficiente ferramenta de apoio.

3.2.1 Fases no Processo de Simulação de Redes Neurais

Quatro fases precisam ser seguidas para resolver um problema utilizando RNAs, essas fases são:

Fase 1. Translação do problema em uma representação adequada para Redes Neurais: esta tarefa envolve, entre outros, a escolha da estrutura da rede a ser usada e a forma de representação dos dados. Essa representação pode incluir a seleção de um conjunto de dados que represente adequadamente o domínio do problema. Nesta fase pode ser incorporado um editor de padrões ao simulador, ou permitir a importação de formatos de arquivos de softwares que manipulem planilhas de cálculo, tais como o MS-EXCEL.

Fase 2. Teste da validade das decisões tomadas na fase anterior: esta fase verifica se o pré-processamento dos dados e a arquitetura escolhida para a rede estão bem definidos. Para isso, o simulador requer uma interface gráfica que permita ao usuário final acompanhar e manipular minuciosamente a arquitetura e os parâmetros da simulação.

Fase 3. Execução da simulação: dependendo do tamanho da rede, da complexidade dos algoritmos e da quantidade de dados, esta fase pode levar alguns dias. Os resultados obtidos nesta fase podem levar o usuário a voltar à fase anterior. Nesta fase a velocidade é um ponto crucial para o desempenho da simulação. Em redes muito grandes o uso de *hardware* específico para processamento paralelo é de grande utilidade.

Fase 4. Análise dos resultados: durante esta fase verifica-se como a rede alcança os resultados, quão bem ela generaliza e se é tolerante a falhas. Aqui é necessário fazer uma análise estatística dos valores produzidos como saída. Deve-se investigar a natureza do processamento das camadas intermediárias em uma rede MLP treinada com o algoritmo *BackPropagation*, por exemplo. Interfaces gráficas e o uso de gráficos comparativos são de grande utilidade.

Um simulador eficiente deve fornecer ferramentas de apoio à execução dessas fases. A provisão de recursos adicionais pode contribuir muito para facilitar o trabalho do usuário [Kock 96].

3.2.2 Recursos Adicionais para complementar uma Simulação de Redes Neurais

Além dos recursos oferecidos para o processo de simulação, existem outros recursos que geralmente são do interesse de usuários de Redes Neurais, tais recursos compreendem: Extensão dos modelos usados, Incorporação do modelo completo em aplicações, e Facilidade de Integração com outras técnicas computacionais. Cada um desses recursos é explicado a seguir com mais detalhes:

- Extensão dos modelos usados - a extensibilidade é um dos principais problemas a ser atacado no projeto de um simulador, ou seja, o simulador deve facilitar a adição de novos módulos e modelos de Redes Neurais, assim como permitir a alteração dos modelos existentes;
- Incorporação do modelo completo em aplicações - a utilidade do simulador está diretamente ligada à facilidade de sua incorporação em um sistema que utilize Redes Neurais, seja como um programa *stand-alone* (para demonstrações) ou como parte de

um sistema maior (por exemplo, um módulo de reconhecimento de padrões dentro de um sistema de produção). Uma alternativa para esta incorporação pode ser a geração do código fonte (rede treinada) que pode ser compilado ou link-editado com outros aplicativos, ou a permissão para a comunicação direta em tempo de execução com outros softwares;

- Facilidade de Integração com outras técnicas computacionais - em aplicações complexas, pode ser interessante combinar as redes com outros métodos computacionais. Uma tendência recente tem sido a utilização de Redes Neurais em Sistemas Híbridos Inteligentes, onde mais de uma técnica inteligente é combinada visando a obtenção de um melhor desempenho. Um exemplo seria a utilização de um módulo para a extração de conhecimento da rede que integraria este conhecimento a um sistema simbólico.

O principal objetivo de um Simulador de Redes Neurais deve ser o de minimizar o esforço realizado pelo usuário em cada uma das fases mencionadas anteriormente. A qualidade de um simulador pode ser avaliada, tanto pelas características da implementação, quanto pelos recursos oferecidos ao usuário. A seguir são apresentados alguns critérios considerados para avaliar um simulador de Redes Neurais.

Características da implementação:

1. paradigma de programação utilizado (por exemplo, orientação a objetos, funcional, procedural ou declarativa);
2. nível de definição das classes (redes, camadas e neurônios);
3. implementação dos algoritmos claramente separada da implementação da interface com o usuário;
4. capacidade de aproveitar a existência de mais de um processador, ou de ser executado de forma distribuída em várias máquinas.

Recursos oferecidos ao usuário:

1. facilidade de aprendizado e operação da Interface Gráfica do Usuário;
2. qualidade na documentação (ajuda *On-Line*, tutorial);
3. facilidade para mudar as propriedades das redes e os parâmetros da simulação;
4. disponibilidade de ferramentas visuais tanto para o desenho da rede quanto para a realização da simulação;
5. disponibilidade de módulos para a análise estatística dos resultados;
6. possibilidade de geração de código em alguma linguagem (por exemplo, C ou C++);
7. disponibilidade de ferramentas para a criação de novos algoritmos;

8. quantidade de algoritmos disponíveis;
9. bibliotecas para o trabalho com matrizes dos tipos: *short*, *integer*, *float*, *double* e tipos de dados complexos;
10. gerenciamento de arquivos comuns e de padrões;
11. modificação simples dos algoritmos existentes;
12. execução tanto no modo interativo quanto em *Batch*;
13. possibilidade de modificação ou extensão do simulador pelo usuário.

A seguir, serão apresentados três dos simuladores mais utilizados pela comunidade científica, analisando o quanto eles auxiliam em cada uma das fases de uma simulação, bem como suas características, vantagens e desvantagens. Serão apresentados e analisados os simuladores *Stuttgart Neural Network Simulator* (SNNS) [Zell 95a], *Open Simulator* [Leber 92, Leber 93] e *Sesame* [Linden 92].

3.3 Stuttgart Neural Network Simulator

O *Stuttgart Neural Network Simulator*, mais conhecido como SNNS, foi criado na Universidade de Stuttgart na Alemanha pelo Professor Zell e sua equipe [Zell 95a]. Atualmente, o SNNS é um dos simuladores mais utilizados no meio acadêmico.

O SNNS é um *software* que pode ser obtido de forma gratuita mediante a autorização dos autores, mas não é um *software* de domínio público. O simulador possui uma grande quantidade de modelos de Redes Neurais já implementadas, tais como: *FeedForward*, *Time Delay*, *ART1*, *ART2*, *ARTMAP*, *Kohonen*, *Jordan*, *Elman*, *Hopfield* e *Auto Associativas*. Alguns desses modelos também possuem variações em relação a sua versão original, tais como *Quick Propagation*, *RProp*, etc.

Este simulador tem a qualidade de poder ser utilizado em muitos tipos de máquinas com diferentes arquiteturas e sistemas operacionais.

A parte gráfica é completamente separada da implementação, isso é aconselhável já que são duas partes de um programa totalmente diferentes. O SNNS pode rodar em dois modos: Interativo e *Batch*. No modo interativo todo o processamento é realizado através de diálogos fornecidos pela Interface Gráfica de Usuário (*Graphic User Interface*, *XGUI*). No modo *Batch* o usuário pode realizar seus experimentos através de arquivos de processamento por lotes.

A Interface Gráfica de Usuário é aceitável em termos gerais, mas poderia ser melhorada com a utilização de ferramentas visuais. Dessa forma, seria muito mais fácil o tratamento dos parâmetros da simulação. Um exemplo típico desses problema surge quando o usuário deseja mudar a função de ativação de uma camada inteira. A única forma para resolver esse

problema é mudar a função em cada um dos neurônios, um por vez. Se a quantidade de neurônios for grande, isso representa uma limitação importante.

Entre os modelos de Redes Neurais Construtivas implementados no simulador SNNS estão disponíveis: *Cascade Correlation*, *Radial Basis Function with Dynamic Decay Adjustment (RBF-DDA)*, ART1 e ART2. O primeiro deles apresentou vários problemas ao se realizar uma simulação durante a sua utilização, por essa razão não pôde ser utilizado para realizar os experimentos apresentados neste trabalho.

Entre as vantagens de se utilizar o simulador SNNS pode-se destacar as seguintes:

- é multi-plataforma, podendo rodar sob várias arquiteturas com sistemas operacionais diferentes;
- pode trabalhar tanto no modo Interativo quanto em *Batch*;
- grande quantidade de algoritmos implementados;
- pode gerar o código da rede já treinada em linguagem C.

Entre suas desvantagens, podem ser mencionadas:

- a Interface Gráfica de Usuário é limitada;
- existem poucos modelos de Redes Construtivas;
- a modificação dos algoritmos existentes é muito complicada;
- não fornece ferramentas para a criação de novos algoritmos.

3.4 Open Simulator

O simulador *Open Simulator* foi desenvolvido por J.F. Leber [Leber 92, Leber 93] no *Institute for Signal and Information Processing* da ETH Zurich na Suíça. A distribuição desse software não é gratuita. O *Open Simulator 3.1* requer o Sistema Operacional *UNIX*, *XView* ou *X-Windows*. Nesse simulador, o usuário dispõe de uma interface gráfica baseada em menus, o simulador possui também um interpretador de comandos, com comandos para salvar a configuração ativa, a topologia que está sendo utilizada e a matriz de pesos, mas, tudo isso através de uma linha de comandos.

O *Open Simulator* pode ser utilizado por mais de um usuário ao mesmo tempo, gerando para cada um deles uma versão estendida do simulador em um espaço de trabalho próprio.

As funções do simulador podem ser divididas em duas categorias: funções do *Núcleo do Simulador (Kernel)* e suas correspondentes funções interativas. Ambos os tipos de funções são utilizados durante o processo de criação de uma Rede Neural. Essa criação é realizada

por meio de um menu, o qual permite a criação de *templates* para todos os arquivos que o usuário desejar. Portanto, o usuário pode inserir seu próprio código nos arquivos criados. No código fonte são utilizadas macros para adicionar variáveis, substituir ou adicionar funções, adicionar menus e desenhar objetos na tela. O usuário pode facilmente adicionar novas macros, assim como definir novas topologias. Para isso pode-se tanto utilizar a interface gráfica quanto um arquivo que contém chamadas a funções em C++.

Para a construção de Redes Neurais estão disponíveis as seguintes ferramentas:

- bibliotecas para o trabalho com matrizes dos tipos: *short*, *integer*, *float*, *double* e tipos de dados complexos;
- bibliotecas gráficas;
- uma classe para o treinamento com o algoritmo *BackPropagation*, uma classe para camadas de *perceptrons* (*perceptron layer class*), e uma topologia que conserva as principais características de um mapa de classes.

Entre as vantagens oferecidas pelo *Open Simulator 3.1* podem ser destacadas as seguintes:

- interface gráfica interativa e também um interpretador de comandos;
- *Kernel* claramente separado da parte interativa;
- baseado em bibliotecas compartilhadas.

Entre as suas desvantagens podem-se mencionar as seguintes:

- a ajuda é limitada e a documentação de usuário está disponível apenas em alemão;
- a interface gráfica possui uma grande quantidade de menus, sendo fácil o usuário se sentir perdido no meio de tantos menus.

A última versão da qual se obteve informação é a *Open Simulator 4.0*. Nessa versão, os objetos de uma simulação podem ser distribuídos em uma rede contendo diferentes arquiteturas. Os algoritmos com uma grande demanda de processamento computacional podem ser executados eficientemente em um super computador, enquanto a parte gráfica pode ser processada em uma *WorkStation*. Dessa forma o programador de um novo módulo não precisa se preocupar com os detalhes da parte gráfica. A implementação dos algoritmos está separada da parte gráfica, como é interessante que seja. O *Open Simulator 4.0* é mais modular e flexível que sua versão anterior, mais simples de utilizar e também mais fácil de manter.

3.5 Sesame

O simulador *Sesame 4.5* foi desenvolvido por Linden e Tietz [Linden 92] no *GMD Schlo Birlinghoven*, na Alemanha, e se encontra disponível na Internet ¹. O simulador *Sesame 4.1* requer o Sistema Operacional *UNIX*, *Inter Views* ou *X-Windows*, além de algumas outras ferramentas e bibliotecas *GNU*. A interface de usuário é baseada em um interpretador de comandos e o estado da simulação pode ser armazenado a qualquer momento. O *Sesame* é baseado em módulos que podem conter qualquer tipo de informação, desde um simples contador ou uma Rede Neural complexa até uma interface gráfica. Os novos módulos podem ser criados com base nos módulos já existentes ou podem ser escritos em C++. No último dos casos, a exportação de variáveis ou nomes de comandos é realizada através de chamadas de funções, e seu manuseio é completamente coberto pelas classes implementadas. Isso inclui a visualização de parâmetros mediante a utilização da interface do usuário. Os parâmetros também podem ser salvos para serem utilizados em futuros experimentos.

Uma simulação é definida como uma sequência de comandos interpretados (escritos ou lidos diretamente de um arquivo) os quais organizam a parte **Estática** e a parte **Dinâmica** do fluxo de dados. O fluxo de dados estático é descrito utilizando comandos, enquanto o fluxo dinâmico de dados é descrito por um ou mais procedimentos, que podem conter chamadas a outros procedimentos, a outros módulos ou simplesmente podem ser estruturas de controle. A junção desses procedimentos e comandos pode ser feita através de linhas de comandos e todos os comandos podem ser armazenados junto com o estado atual dos módulos em arquivos fáceis de ler.

Os simulador *Sesame* fornece as seguintes ferramentas:

- módulos para a criação de Redes Neurais clássicas como MLP, RBF e Kohonen, assim como gerenciamento de memória distribuída;
- Interface Gráfica para análise estatística dos resultados;
- gerenciamento de matrizes e vetores para trabalhar com variáveis de ponto flutuante;
- funções para o gerenciamento de listas, pilhas, arquivos comuns, e arquivos de padrões.

Entre as vantagens do simulador *Sesame 4.5* encontram-se as seguintes:

- provê ajuda *On-Line*, boa documentação e tutorial;
- é fácil de entender e utilizar;
- o seu código fonte é fácil de manter e entender;
- é muito flexível na hora de utilizar os módulos como parte de uma simulação completa.

¹ *Sesame* pode ser obtido via ftp e usuário *anonymous* do endereço <ftp.gmd.de>.

Entre as desvantagens podem-se destacar as seguintes:

- não possui uma Interface Gráfica para a execução da simulação, sendo a execução dos experimentos realizada apenas em linhas de comando;
- é excessivamente dividido em pequenos módulos, causando confusão na hora de realizar alguns experimentos.

Vale ressaltar que tanto o *Open Simulator 3.1* quanto o simulador *Sesame 4.5* permitem a criação de Redes Neurais de Alta Ordem (*Redes Neurais Ontogênicas*) que são redes mais complexas. Mais detalhes sobre esse tipo de redes podem ser encontrados em [Thimn 94].

3.6 Conclusões

Neste capítulo foram apresentados alguns dos simuladores de Redes Neurais existentes, cada um deles com suas características. É necessário conhecer os diferentes simuladores existentes no mercado para poder aproveitar as melhores características de cada um deles.

Um simulador de Redes Neurais deve fornecer ao usuário as ferramentas necessárias para que ele possa criar seus próprios algoritmos. Dessa forma, o simulador poderia continuar crescendo de acordo com as necessidades do usuário. A falta de ferramentas visuais apropriadas é um problema comum a todos os simuladores analisados. A medida que a Interface Gráfica se torna mais natural para o usuário, ele pode se concentrar mais na realização dos seus experimentos. Uma vez que o domínio da Interface Gráfica representa mais um conjunto de comandos a serem aprendidos, a quantidade desses comandos deve ser minimizado.

No próximo capítulo será apresentado o Simulador Kipu. Esse simulador foi criado com o objetivo de minimizar algumas das dificuldades apresentadas pelos simuladores citados anteriormente. O Simulador Kipu foi projetado para a simulação de Redes Neurais Construtivas e foi desenvolvido com ferramentas e técnicas atuais.

Capítulo 4

Kipu, Um Simulador de Redes Neurais Construtivas

4.1 Introdução

Dentro das atuais linhas de pesquisa em Redes Neurais, o projeto de simuladores cumpre um papel preponderante no que se refere a confiabilidade e facilidade na definição de arquiteturas de redes para a solução de problemas. Embora exista um grande e diversificado conjunto de simuladores disponíveis, é difícil prever o aparecimento de novos algoritmos para trabalhar com Redes Neurais.

Devido a popularidade alcançada pelas RNA, há um crescimento no número de sistemas que utilizam o paradigma conexionista. Um simulador de Redes Neurais deve ser capaz de incluir novos modelos de RNA, assim como de modificar os algoritmos existentes sem maiores problemas. A facilidade de inclusão de novos modelos é um ponto chave. A modularidade e extensibilidade dos simuladores são muito importantes para a aplicação, abrangência e eficiência dos mesmos. Um dos objetivos a serem alcançados com o simulador proposto neste capítulo parte deste ponto. O que se pretende é construir um simulador capaz de incluir, sem maiores esforços, qualquer modelo de rede e que suporte o aparecimento de novos algoritmos. Desta maneira estarão incluídos os algoritmos construtivos, assim como as redes mais comuns, como os algoritmos para redes do tipo MLP [Freeman 93]. Futuramente pretende-se implementar mais algoritmos para outros modelos de Redes Neurais tais como ART1, ART2, RBF, e RBF-DDA.

Um dos objetivos comuns a todos os simuladores existentes é o de reduzir o tempo e o esforço envolvidos na utilização de Redes Neurais para a resolução de problemas. Para alcançar tais objetivos, o simulador proposto neste capítulo deve possuir uma interface gráfica maleável e de fácil utilização, onde a criação de uma rede e a especificação dos seus parâmetros possam ser feitos interativamente utilizando técnicas e ferramentas visuais.

Para poder reduzir o tempo de processamento da rede e garantir a extensibilidade do simulador, o mesmo possuirá internamente uma estrutura de dados implementada usando o

paradigma da Orientação a Objetos, facilitando assim o entendimento e a manipulação da estrutura de dados escolhida.

Para cumprir com os objetivos traçados, realizou-se um estudo sobre os recursos oferecidos pelo sistema operacional *Windows 95*, principalmente no que se refere à criação e gerenciamento de DLLs, já que, esse foi um aspecto fundamental para a construção do simulador.

O principal incentivo para a construção do simulador Kipu ¹ foi a necessidade de investigar o comportamento das RNC (Capítulo 2). Os simuladores de Redes Neurais existentes não suportam várias dessas redes, em vista disso, foi necessária a construção de uma ferramenta apropriada para possibilitar a simulação de diferentes modelos de RNC. As RNC requerem estruturas muito mais flexíveis que as empregadas pelas Redes Neurais com topologia fixa. Isto ocorre devido à possibilidade do crescimento do número de elementos da rede durante seu treinamento. Mais detalhes podem ser encontrados em [Cuadros 97].

Este capítulo está organizado da seguinte maneira. A Seção 4.2 ilustra as características do Interface Gráfica do Usuário do simulador proposto. Dentro da Interface Gráfica do Usuário na Seção 4.2.1 é apresentada a Interface para a Criação de Redes. Na Seção 4.2.2 é apresentada a Interface para o Controle dos Parâmetros da Simulação, e na Seção 4.2.3 é apresentada a Interface para que o usuário possa acompanhar o processo da simulação. Na Seção 4.3 são detalhadas as características da implementação do Simulador Kipu. Uma descrição da estrutura de dados usada pelo simulador é apresentada na Seção 4.3.1. Na Seção 4.3.3 são apresentados os algoritmos implementados no simulador, e na Seção 4.3.2 são apresentadas as técnicas que foram utilizadas para a Geração de Código para criar Novos Algoritmos. Na Seção 4.4 são apresentados os utilitários implementados dentro do Simulador Kipu. Dentre os utilitários existentes destacam-se: o módulo para Geração de Arquivos de Padrões (ver Seção 4.4.1) e o Módulo para a Visualização de Padrões (Ver Seção 4.4.2). Finalmente, na Seção 4.5 são apresentadas as conclusões deste capítulo.

4.2 Interface Gráfica do Usuário

Nesta seção apresenta-se a Interface Gráfica do Usuário (GUI) do Simulador Kipu. Formando parte da GUI, serão apresentados os recursos implementados no simulador Kipu, tais recursos servirão como ferramenta de apoio a cada uma das fases citadas no Capítulo 3. O simulador Kipu foi desenvolvido inteiramente na linguagem C++ devido a flexibilidade, controle total da simulação e ao fato de que a estrutura de um programa que usa o paradigma da orientação a objetos reflete a natureza do problema com maior facilidade. O compilador utilizado foi Microsoft Visual C++ 5.0.

¹Kipu foi o instrumento utilizado pela cultura Inca para a codificação numérica. Um Kipu é composto por um conjunto de cordas de diferentes cores e nós a situados em diferentes alturas o que permite representar qualquer número.

4.2.1 Criação de Redes

O fato do usuário ter que aprender a utilizar um novo programa (neste caso um simulador), representa um conjunto de procedimentos e/ou operações que ele deve incorporar ao seu conhecimento, portanto, a tarefa principal no que se refere a Interface Gráfica é fazer com que os comandos existentes no programa sejam simples de utilizar. Um outro objetivo da Interface Gráfica é minimizar a quantidade de procedimentos e/ou funções que o usuário precisa aprender, desta forma, o usuário verá o seu trabalho minimizado e poderá resolver seus problemas sem maiores dificuldades.

O simulador Kipu, foi desenvolvido com base nessa filosofia. Deve ser simples para o usuário desenhar uma rede, assim como modificar as características da mesma, isto é, modificar a quantidade de unidades de uma camada, criar e eliminar unidades, conectar neurônios com um ou mais neurônios ou camadas, modificar as funções utilizadas para a inicialização, ativação e saída de qualquer elemento na rede e aproveitar a interface visual para mudar a escala, posição de uma rede, etc. Na Figura 4.1 é apresentada uma pequena rede criada aproveitando a Interface Gráfica do simulador Kipu.

Figura 4.1: Desenho Visual de uma Rede Neural utilizando o simulador Kipu.

Os usuários do simulador Kipu devem estar familiarizados com a interface do Sistema Operacional *Windows 95*, portanto, é possível e recomendável aproveitar o conhecimento prévio que o usuário já possua. No Sistema Operacional *Windows 95* é comum que os programas forneçam a possibilidade de analisar as propriedades de um objeto, ativando o *Menu Popup*

que aparece quando é ativado o botão direito do *Mouse*, e escolhendo depois disso a opção *Properties*. Por causa disso pensou-se na implementação do diálogo de *Properties* (ver Figuras 4.2, 4.3 e 4.4) que é ativado da mesma forma que em todos os outros programas desse Sistema Operacional. A opção *Properties* está disponível para todos os objetos existentes em uma rede (neurônios, camadas e a mesma rede). Por outro lado, é adequado aproveitar o mesmo diálogo para todos os elementos de uma rede, assim, o usuário não precisa sair do diálogo e entrar novamente quando desejar analisar as propriedades de objetos que não sejam do mesmo tipo. Para a realização desta tarefa, o diálogo de *Properties* possui uma divisão vertical com dois painéis distribuídos em forma horizontal como pode ser observado na Figura 4.2. O painel da esquerda contém uma árvore hierárquica contendo todos elementos existentes em uma mesma janela, isto é, redes, camadas e neurônios. Se o usuário desejar ver as propriedades de um outro objeto só terá que escolhê-lo na árvore da esquerda e automaticamente o painel da direita será trocado pelo diálogo adequado de acordo com o tipo de objeto selecionado (rede, camada ou neurônio). Nas Figuras 4.3 e 4.4 são apresentados os diálogos correspondentes às propriedades de uma camada e de um neurônio respectivamente.

Figura 4.2: Diálogo de propriedades de uma Rede Neural.

No nível de rede, os parâmetros variam de acordo com o tipo da rede simulada. Por exemplo, se for uma rede Perceptron Multi-Camadas, será possível escolher o tipo de algoritmo de aprendizado que será utilizado para o treinamento. Já no nível de camada existem mais parâmetros, como, por exemplo: o nome da camada, o tipo de camada, o número de neurônios que a camada possui, assim como o número de colunas, que serve para distribuir os neurônios em forma de uma matriz bidimensional. No diálogo *Properties* de uma camada também existe um grupo que se refere às funções de inicialização, ativação e saída. Se todos

os neurônios da camada compartilharem a mesma função, está será selecionada automaticamente, caso contrário nenhuma função é selecionada. Por outro lado, se o usuário escolher uma nova função para qualquer dos três casos, a função escolhida será aplicada a todos os neurônios da camada. No caso do diálogo de propriedades para um neurônio (ver Figura 4.4), as três funções anteriormente citadas podem também ser trocadas, a única diferença é que o efeito é voltado unicamente para aquele neurônio.

Figura 4.3: Diálogo de propriedades de uma camada.

Uma vez concluída a fase do desenho visual da rede, um ou mais conjuntos de dados devem ser carregados na memória para a realização do treinamento. A leitura dos dados pode ser realizada de arquivos de diferentes formatos, tais como, arquivos gerados por outros simuladores, como SNNS (*Stuttgart Neural Network Simulator*) [Zell 95a] que são arquivos em formato ASCII. Pretende-se implementar no futuro a leitura de planilhas de cálculo como o MS-EXCEL. A seguir, na Seção 4.2.2 são descritos os recursos implementados no simulador Kipu para poder acompanhar o processo de simulação.

Figura 4.4: Diálogo de propriedades de um neurônio.

4.2.2 Controle dos Parâmetros da Simulação

Para que o usuário possa acompanhar o processo de simulação com facilidade, é recomendável a existência de recursos adequados que permitam apresentar os resultados da simulação tanto de forma gráfica, quanto de forma textual. Gráficos e tabelas comparativas foram implementadas com o intuito de permitir uma rápida análise do desempenho da rede e assim chegar à melhor arquitetura com maior rapidez. Entende-se como a melhor arquitetura para um determinado problema, aquela arquitetura que consegue resolver o problema alcançando um nível aceitável de erro a um custo computacional mínimo.

Dentro do *Menu Popup* de uma rede existe a opção *Training Network ...* que permite ao usuário escolher o algoritmo que será utilizado. Na Figura 4.5 pode-se observar o diálogo que permite realizar o treinamento de uma rede. Nesse diálogo existem três áreas. Na parte superior esquerda existe uma árvore com os algoritmos carregados. Uma vez escolhido o algoritmo com que será realizado o processo de treinamento da rede, o diálogo apropriado para esse algoritmo é ativado na parte inferior do diálogo. No exemplo, pode ser observado o diálogo de parâmetros para o treinamento de uma rede utilizando o algoritmo *Tower*.

Na parte superior direita existe um diálogo cuja função é oferecer ao usuário a possibilidade de carregar e/ou escolher os arquivos de padrões que serão utilizados junto a algum dos

algoritmos existentes. Uma vez escolhidos os arquivos adequados para realizar a simulação, o usuário pode determinar os parâmetros adequados para o treinamento da rede. Depois de ter determinado os parâmetros de treinamento, é possível começar o seu treinamento. Na divisão inferior da Figura 4.5 pode-se observar o diálogo utilizado para determinar os parâmetros de uma simulação utilizando o algoritmo *Tower*.

Figura 4.5: Entrada de dados utilizando um diálogo contido em uma DLL.

4.2.3 Monitoramento da Simulação

Cada algoritmo foi implementado dentro de um arquivo DLL que é gerado separado do simulador. O uso de DLLs (Dynamic Link Libraries) [O'Reilly 96a] possibilita a extensibilidade do simulador, ou seja, os usuários podem criar suas próprias DLLs e usá-las junto ao simulador. Dessa forma, podem ser criados e adicionados novos algoritmos de treinamento em separado. O uso de DLLs é vantajoso porque elimina a necessidade de recompilar o código do programa principal (neste caso do simulador). Além disso, o simulador não precisa ter todos os algoritmos em memória ao mesmo tempo, e as DLL só devem ser carregadas quando for necessário otimizando assim, o uso da memória do computador.

Por outro lado cada algoritmo precisa de um certo conjunto de parâmetros específicos, portanto, o diálogo que serve para a entrada de dados de cada um deles deve estar contido dentro da DLL. Desta forma cada algoritmo pode trabalhar com os seus parâmetros específicos sem problemas, dispondo para isso de um diálogo personalizado. No momento que o simulador precisa carregar um certo algoritmo contido em uma DLL, ele carrega também o diálogo para a entrada de parâmetros que deve estar contido na mesma, e o insere no diálogo de treinamento realizando a operação conhecida como *Plug-In* de forma transparente.

Na Figura 4.6 pode ser observada a curva de erro e validação durante o processo de treinamento de uma RNA utilizando o algoritmo *Backpropagation*. Na Figura 4.7 pode-se observar os dados do treinamento de forma numérica, assim como o resultado da fase de teste.

Figura 4.6: Curva de erro durante o treinamento de uma RNA.

Figura 4.7: Resultados obtidos durante o treinamento de uma RNA e no teste.

Para que o simulador possa suportar qualquer modelo de rede sem comprometer o desempenho das simulações, o mesmo deve estar baseado em uma estrutura de dados que seja robusta, e flexível. Na Seção 4.3 são apresentadas as características de implementação do simulador Kipu.

4.3 Implementação do Simulador Kipu

Como já foi observado no Capítulo 2, as RNC requerem uma estrutura de dados flexível para poderem inserir elementos na rede ao longo do processo de simulação. A estrutura de dados proposta na Seção 4.3.1 foi projetada para suportar tanto a adição de elementos (que é necessário para as RNC), quanto a remoção dos mesmos (no caso de aplicar técnicas de *Pruning*). Na Seção 4.3.2 é apresentado o módulo para Criação e Adição de Novos Algoritmos que também foi implementado no simulador Kipu.

4.3.1 Estrutura de Dados

O tempo total de uma simulação de uma RNA pode ser determinado por um intervalo de tempo T , que pode ser definido como:

$$T = T_i + T_s \quad (4.1)$$

onde T_s é o tempo utilizado pelos algoritmos implementados usando o simulador, e T_i é o tempo consumido internamente pelo próprio simulador. O tempo T_s está fora do ambiente de controle, por causa disso, o simulador deve minimizar ao máximo o tempo T_i .

As operações básicas realizadas por uma Rede Neural são somas e multiplicações, que são computacionalmente simples. O problema está na grande quantidade destas operações que a rede precisa executar, o que provavelmente implica em acessos frequentes às unidades da rede. Ao considerar a grande quantidade de operações, e o consequente número de acessos, deve-se levar em conta que é necessário realizar um acesso veloz a qualquer unidade da rede, com o objetivo de minimizar T_i .

A escolha de vetores estáticos traz consigo a definição de um tamanho fixo para eles, o que limitaria o dinamismo da rede. Isto faz com que esta escolha não seja apropriada para implementações de RNC, as quais precisam adicionar ou eliminar unidades com frequência. Uma alternativa envolve a utilização de uma lista encadeada para armazenar tais unidades. Isto implicaria em um gasto adicional de espaço pela própria natureza de uma lista encadeada. Este espaço pode ser muito significativo, visto que é comum existirem redes com uma grande quantidade de unidades. Além disso, existe o *overhead* ligado à busca de informações na lista.

Para solucionar o problema de acesso direto e dar uma maior flexibilidade ao crescimento ou redução dos elementos da rede (camadas, neurônios, conexões), foi modelado um *template* chamado *TFastArray* [Mumit 95, Kirman 95, Bowden 95], que encapsula todas as operações relacionadas à manipulação de vetores dinâmicos. Algumas operações consideradas foram:

- inserção de elementos;
- remoção de elementos;
- crescimento automático regulável;
- aplicação de uma operação a todos os elementos;
- procura do primeiro elemento que satisfaça uma condição.

Um *template* é um padrão que serve para implementar operações com Tipos Abstratos de Dados (TAD) [Venter 97]. Pelo próprio conceito de TAD, um *template* só precisa ser escrito uma vez e o compilador gera automaticamente as diversas variações de código, segundo o tipo de dado com o qual esteja sendo utilizado. O crescimento do vetor representa um *overhead* devido à necessidade de alocar um novo espaço na memória, copiar os dados nesse

novo espaço e liberar o espaço antigo. Para reduzir este *overhead*, a taxa de crescimento do vetor pode ser regulável de acordo com uma variável que assume valores maiores que 1, o que significa que é possível aumentar ou reduzir o tamanho do vetor em quantidades múltiplas desta variável.

Cada rede é um objeto da classe *TNet*, sendo composta por um vetor de ponteiros que apontam para objetos da classe *TLayer* (camadas), numerados de 0 até $i-1$, onde i é o número total de camadas. Uma vantagem adicional do uso de um vetor de ponteiros é que isso facilita a movimentação de objetos sem a necessidade de mover fisicamente os mesmos.

A classe *TLayer* implementa as operações referentes às camadas da rede. Cada objeto *TLayer* possui um vetor de ponteiros, que apontam para objetos da classe *TNeuron* (neurônios) com elementos de 0 até $j-1$, onde j é o número de neurônios dentro de uma camada.

A classe *TNeuron* encapsula as operações próprias dos neurônios, assim como as funções de combinação usadas pelas RNC. Cada neurônio possui dois vetores com elementos numerados de 0 até $k-1$, onde k é o seu número de conexões de entrada. O primeiro deles armazena ponteiros que apontam aos neurônios que lhe fornecem a entrada. Tais unidades podem ser neurônios ou funções de combinação. O segundo vetor armazena os pesos associados a cada entrada.

Modelou-se também uma classe *TPatternManager*, que encapsula as operações referentes à comunicação entre as camadas de entrada e saída com os arquivos de padrões. Esta classe deve fornecer as entradas a partir de arquivos com diferentes formatos, tais como, os gerados por planilhas de cálculo e outros simuladores (por exemplo, SNNS). No caso de comunicação com outros softwares, poderia ser utilizado um canal de comunicação entre o simulador e o software do tipo DDE (Dynamic Data Exchange) [O'Reilly 96b] ou OLE (Object Linking and Embedding) [O'Reilly 96c]. Para o segundo caso serão implementadas as rotinas apropriadas para reconhecer os dados do arquivo. O desenho da estrutura interna do simulador é apresentado na Figura 4.8.

processo de *Link* (ligação) realizado imediatamente após a compilação, nesse caso o programa principal não poderá rodar se a DLL não existir.

O enlace de forma **Dinâmica** ocorre quando o programa principal não conhece com antecedência o nome da DLL. Um exemplo muito conhecido no qual acontece este tipo de enlace é quando instala-se um *Driver* de um novo dispositivo, por exemplo, um monitor, *CD-ROM*, etc. É muito comum que o fabricante crie seu próprio *driver* e o usuário o instale sem problemas. Isto ocorre de forma transparente porque foram estabelecidos alguns protocolos para possibilitar uma boa comunicação entre o software sendo instalado e o Sistema Operacional.

Este segundo tipo de enlace requer muito mais cuidado que o enlace estático. Antes de mais nada, devem estar definidos os mecanismos de comunicação, isto é, é necessário definir um protocolo de comunicação comum para ambos os lados.

Uma DLL possui basicamente três tipos de elementos: os convencionais que são utilizados internamente pela DLL, os do tipo exportáveis, e os que precisam ser importados. Tais elementos podem ser: funções, implementações de classes, variáveis, etc.

Os elementos convencionais são aqueles que são utilizados internamente pela DLL. Esses elementos não precisam ser compartilhados, eles são de uso exclusivo da DLL. Os elementos exportáveis são aqueles que serão utilizados por outros programas ou DLLs. Se os elementos fossem variáveis, elas estariam declaradas dentro da DLL, e se fossem funções ou métodos eles seriam implementados dentro dela. Os elementos importáveis são aqueles que a DLL precisa quando é carregada na memória, mas não estão contidos nela. Para a declaração desses dois últimos tipos de membros em uma DLL são utilizadas as palavras reservadas *dllexport* e *dllimport*, respectivamente. Na Figura 4.9 são apresentados alguns exemplos de declarações de variáveis que foram declaradas para serem importadas ou exportadas².

```
_declspec(dllexport) int x;  
_declspec(dllimport) CString title;  
  
_declspec(dllexport) CRuntimeClass* GetAlgorithmClass()  
{  
    return RUNTIME_CLASS(CPyramid);  
}  
  
class _declspec(dllimport) CTower; // declaração forward  
class _declspec(dllexport) CPyramid : public CTower  
{  
    // membros da classe  
};
```

Figura 4.9: Membros importáveis e exportáveis declarados dentro de uma DLL.

As funções, classes e variáveis exportáveis são acessíveis de fora da DLL e podem ser uti-

²A implementação apresentada corresponde à sintaxe do compilador *Microsoft Visual C++*.

lizadas depois de realizar um enlace dinâmico. No projeto do Simulador Kipu, as DLL que contêm um algoritmo possuem uma função chamada `GetAlgorithmClass`, com uma estrutura similar à apresentada no exemplo anterior. A única diferença de uma DLL para outra está na classe que ela retorna. Na figura apresentada, a função `GetAlgorithmClass` corresponde ao algoritmo *Pyramid*. Esta função é o único ponto de entrada da DLL, e também é suficiente para que o Simulador possa se comunicar com o algoritmo implementado.

Para a implementação do simulador *Kipu* foi utilizando o compilador *Microsoft Visual C++ 5.0*, com MFC (*Microsoft Foundation Classes*) 4.2. Inicialmente foi criada uma classe chamada *CAlgorithm* herdada da classe *CObject*³. A classe *CAlgorithm* é a base para a criação de qualquer algoritmo, ela é uma **Classe Base Abstrata**, isto é, foi criada para ser herdada. Os algoritmos implementados inicialmente foram organizados em uma hierarquia de classes como pode ser observado na Figura 4.10.

Figura 4.10: Hierarquia de classes formada por diversos algoritmos.

Desde o início do projeto, um dos objetivos almejados foi possibilitar ao usuário a criação de seus próprios algoritmos. A tarefa de criar um novo algoritmo envolve um conjunto de operações que devem ser executadas em sequência. Esse conjunto é formado por aproximadamente doze operações. O principal problema desse método é que ele não é um procedimento prático. Alguns dos passos executados são procedimentos mecânicos que podem ser realizados automaticamente por um programa. Em vista disso estudou-se a possibilidade de que o simulador gerasse automaticamente o código para a criação de novos algoritmos. A tarefa de gerar código não é fácil, o principal problema foi a falta de exemplos de programas geradores de código. Os exemplos fornecidos no *CD-ROM* do *Visual C++ 5.0* referentes à geração de código não funcionaram como se esperava mas, o código fonte dos mesmos serviu como ponto de partida para a implementação um gerador de código próprio.

A idéia é muito simples: para gerar um arquivo é necessária a existência de um arquivo *Template*. Esse arquivo *Template* pode ser um arquivo de texto normal, mas contém algumas

³ *CObject* é uma classe pertencente a hierarquia da MFC.

macros delimitadas pela sequência \$\$⁴, por exemplo, \$\$CLASSNAME\$\$\$. Na Figura 4.11 é apresentado um pequeno arquivo que serve como exemplo para entender a estrutura de um arquivo *Template* comum.

```
//
// Este arquivo foi gerado por: $$AUTHOR$$
//
#include "$$CLASSEBASE$.h"

class C$$CLASSNAME$$: public $$CLASSEBASE$$
{
    C$$CLASSNAME$$()
        { AfxMessageBox("Construtor da classe C$$CLASSNAME$$");
    ~C$$CLASSNAME$$()
        { AfxMessageBox("Destrutor da classe C$$CLASSNAME$$");
public:
    virtual void Run();
};

void
C$$CLASSNAME$$::Run()
{
    $$CLASSEBASE$$::Run();
}
```

Figura 4.11: Modelo de um arquivo *Template* utilizado para geração de código.

O módulo de geração de código interage com um dicionário de sinônimos, para poder substituir as macros pelos símbolos adequados. Com a ajuda desse dicionário de sinônimos, a macro \$\$CLASSNAME\$\$ poderia ser substituída, por exemplo, pelo símbolo *Tiling* gerando assim a classe *CTiling*. Desta forma, podem ser gerados os arquivos .cpp, .h, .def, o projeto (.dsp) or recursos (por exemplo, o diálogo para os parâmetros, etc), e até um arquivo explicando a organização dos arquivos gerados (Readme.txt). Depois de ter realizado a criação do projeto, o usuário deve modificar o método *Run()* criado junto ao algoritmo para que realize as operações do novo algoritmo. Uma vez realizadas as modificações necessárias no código, o projeto inteiro deve ser compilado para a geração do arquivo DLL⁵.

O arquivo DLL gerado deve ser adicionado ao simulador. Para realizar essa operação, na Figura 4.5 (Pág. 40) o painel da parte superior esquerda possui um *Menu Popup*, que pode ser ativado clicando com o botão direito do *mouse*. Nesse menu existe a opção *Add new algorithm ...* que foi criada com essa finalidade. Depois de ter escolhido essa opção do Menu, um diálogo é apresentado onde o usuário poderá escolher o arquivo DLL que deseje carregar. Finalmente, o novo algoritmo é adicionado à lista dos já carregados e o seu diálogo de parâmetros é ativado no painel da parte inferior do mesmo diálogo.

⁴O símbolo \$\$ foi escolhido porque o *Visual C++* gera seu código utilizando esse padrão e é muito simples de entender.

⁵O projeto inteiro é gerado para ser compilado com *Visual C++ 5.0*.

A criação de um novo algoritmo é realizada por meio do *Menu Utilities| Wizard to create new Algorithms ...*. O diálogo para a criação de um novo algoritmo é apresentado na Figura 4.12

Figura 4.12: Diálogo utilizado para a criação de um novo algoritmo.

No exemplo apresentado na Figura 4.12, o usuário está criando uma nova classe chamada *CRBF_DDA* herdada da classe *CAlgorithm*. Para poder escolher o algoritmo que servirá de base à nova classe, o usuário dispõe do botão *Choose* que permite a escolha da classe base. A seguir pode se observar na Figura 4.13 a hierarquia de classes onde o usuário pode realizar a escolha da classe base do novo algoritmo.

Figura 4.13: Diálogo utilizado para a criação de um novo algoritmo.

Na próxima seção, serão apresentados os algoritmos que foram implementados no simulador Kipu.

4.3.3 Algoritmos Implementados no Simulador Kipu

Os algoritmos que foram implementados para a realização dos experimentos neste trabalho de pesquisa são os seguintes:

- *BackPropagation Padrão*;
- *Cascade Correlation*;
- *Tower*;
- *Pyramid*.

Como já foi apresentado na Figura 4.10, a classe *CAlgorithm* é a base para a criação de todos os algoritmos. Essa classe é uma classe abstrata, quer dizer, não é possível criar um objeto dessa classe, unicamente podem ser herdadas novas classes com o objetivo de criar novos algoritmos. A seguir são apresentadas uma a uma as implementações dos algoritmos mencionados anteriormente.

Algoritmo BackPropagation

Este algoritmo foi o primeiro a ser implementado, dentro dessa classe foram implementados os métodos básicos para propagar os sinais através da rede tanto para a fase *Forward*, quanto para a fase *Backward*. O sinal da rede é propagado da Camada de Entrada até a camada de Saída utilizando o método *ShowOnePattern* como pode ser observado a seguir.

```
void _convention
CStdBackPropagation::ShowOnePattern()
{
    // Forward Phase ...
    m_pNet->ApplyOnNeurons( TNeuron::Process );
    Error_Pattern = 0.0;
    //CalcError for output_layer
    output_layer->ApplyOnNeurons( TNeuron::CalcError );
    // Sum the error on Error_Pattern
    output_layer->ForEach( ::SumError, &Error_Pattern );
    (this->*lpfnAfterShowPattern)();
}
```

A variável *m_pNet* é um ponteiro que aponta para a rede que esta sendo treinada. Todos os algoritmos podem acessar livremente essa variável. Todas os algoritmos dispõem também dos ponteiros *input_layer*, e *output_layer* que apontam à camada de entrada e à camada de saída, respectivamente. O método *ApplyOnNeurons* está disponível tanto para objetos da classe *TNeural_Network* (redes), quanto para os da classe *TLayer* (camadas). Esse método

aplica um outro método a todos os neurônios que estejam contidos dentro de um dado objeto (rede ou camada). No exemplo anterior, a instrução: `m_pNet->ApplyOnNeurons( TNeuron::Process );` executa o método *Process* para cada um dos neurônios da rede ⁶. Observe-se que a função *ShowOnePattern*, além de propagar o sinal de um padrão da Camada de Entrada até a Camada de Saída, calcula o erro para aquele padrão na variável *Error_Pattern* que está disponível dentro do algoritmo.

Para realizar as fases *Forward* e *Backward* juntas para um padrão, dispõe-se do método *TrainOnePattern* que está implementado da seguinte maneira:

```
void _convention
CStdBackPropagation::TrainOnePattern()
{
    // Forward phase ...
    ShowOnePattern();
    // Backward phase
    m_pNet->ApplyOnNeuronsInverted( TNeuron::CorrectWeightsBP );
}
```

Para realizar um ciclo de treinamento (fases *Forward* e *Backward* juntas para todos os padrões uma única vez), dispõe-se do método *TrainAllPatterns*. Esse método foi implementado da seguinte forma:

```
void _convention
CStdBackPropagation::TrainAllPatterns()
{register long i;

    // Train with all the patterns
    m_pActiveDataSet->rewind();
    Error_Cycle = 0.0;
    for( i = 0 ; i < npatterns ; i++ )
    {
        TrainOnePattern();
        Error_Cycle += Error_Pattern;
        m_pActiveDataSet->Next();
    }
}
```

O método *TrainAllPatterns* além de treinar a rede com todos os padrões do conjunto de treinamento, calcula o erro do ciclo, o mesmo que, é armazenado na variável *Error_Cycle*. Para poder realizar uma validação é necessário apresentar todos os padrões do conjunto de treinamento uma única vez sem modificar os valores dos pesos da rede. Essa operação é realizada pelo método *ShowAllPatterns* que foi implementado da seguinte maneira:

⁶O método *Process* da classe *TNeuron* processa as entradas de um neurônio, calcula seu valor de ativação, e gera seu valor de saída.


```

void _convention
CStdBackPropagation::ShowAllPatterns()
{register long i;

    GetActiveData()->rewind();
    Error_Cycle = 0.0;
    for( i = 0 ; i < npatterns ; i++ )
    {
        ShowOnePattern();
        Error_Cycle += Error_Pattern;
        // Move to the next pattern
        m_pActiveDataSet->Next();
    }
}

```

Os quatro métodos apresentados anteriormente foram fundamentais para a implementação dos algoritmos construtivos que serão apresentados nas próximas seções, assim como, para a criação do próprio algoritmo *BackPropagation* cujo método principal é apresentado a seguir:

```

void _convention
CStdBackPropagation::Run()
{register long i;

    if( parameters->m_InitWeights ) // is the flag set in the dialog ?
        InitWeights();
    m_pNet->ForEach( ::SetLearningRate, &parameters->m_LearningRate);
    for( i = 0; i < parameters->m_Cycles; i++)
    {
        // Shuffle the data ?
        SetActiveData( TrainingSet );
        if( parameters->m_Shuffle )
            GetActiveData()->Shuffle();
        TrainAllPatterns();
        UpdateGraphicError( (Error_Cycle/npatterns)/number_outputs,
                            TRAINING_LINE, i, parameters->m_Cycles );
        if( parameters->m_Validation )
            if( i % parameters->m_Validation == 0 ) // Validation each
                Validation(i); // parameters->m_Validation times
        ::PumpWaitingMessages();
    }
}

```

Algoritmo Cascade Correlation

O algoritmo *Cascade Correlation* foi implementado tomando como base o algoritmo *Back-Propagation*. A escolha de *CStdBackPropagation* como classe base foi devido ao fato que este algoritmo já possui implementados os métodos para a realização da fase *Forward*, portanto pode-se aproveitar esses métodos que já estão implementados.

```
void _convention
CCascadeCorrelation::Run()
{register long i;
 long patience,
     insert_position = 1;
 double Training_Error;
 CString name;

 number_candidates = parameters->m_Number_of_Candidates;
 pc = new TLayer( m_pNet->GetWindowParent(),
                 number_candidates, SpecialLayer );
 m_pNet->AddAt( pc, 1 );
 nlayers++;
 pc->ForEach( ::SetInitFunction, TNeuron::Init_Default);
 pc->ForEach( ::SetActivationFunction, TNeuron::Sigmoid_Activation);
 pc->ForEach( ::SetOutputFunction, TNeuron::Output_Default);
 m_pNet->ForEach( ::SetLearningRate, &parameters->m_Learning_Rate);

 deltax = deltax = 0;
 point = CPoint(input_layer->right + 10, input_layer->top + 10);
 lpfnAfterShowPattern = &CAlgorithm::Empty;

 ::ForEach(pc, &TNeuron::ChangeAuxFunction,
           &TNeuron::lpfnAfterCalcActivation, &TNeuron::Sum_Activations);
 ::ForEach(output_layer, &TNeuron::ChangeAuxFunction,
           &TNeuron::lpfnAfterCalcError, &TNeuron::Sum_Errors);
 last_error = 3.0;
 Training_Error = 0.0;
 // Connect input layer with output layer ...
 input_layer->ConnectWith( output_layer );
 if( parameters->m_InitWeights ) // is the flag set in the dialog ?
     InitWeights();
 i = 0;
 for( patience = 0 ; patience < parameters->m_Patience &&
      i < parameters->m_Cycles ; i++ )
 {
     if( fabs( Training_Error - last_error ) < parameters->m_ErrorChange )
         patience++;
 }
```

```

output_layer->ApplyOnNeurons( TNeuron::Init_Errors );
last_error = Training_Error;
SetActiveData( TrainingSet );
if( parameters->m_Shuffle )      // Shuffle the data ?
    GetActiveData()->Shuffle();
TrainAllPatterns();
Training_Error = Error_Cycle;
UpdateGraphicError( (Error_Cycle/npatterns)/number_outputs,
                    TRAINING_LINE, i, parameters->m_Cycles );
if ( parameters->m_Validation )
    if( i % parameters->m_Validation == 0 )
        Validation(i);
    ::PumpWaitingMessages();
}
CreateCorrelationMatrix();
input_layer->ConnectWith( pc );
if( parameters->m_InitWeights ) // is the flag set in the dialog ?
    InitWeights();
for( ; i < parameters->m_Cycles && Error_Cycle ; )
{
    InitCorrelationMatrix();
    SetActiveData( TrainingSet );
    ShowAllPatterns();
    SetCandidatesWeights(); // Train candidate weights
    selected_candidate = SearchCandidate();
    // Create new hidden layer ...
    new_layer = new TLayer( m_pNet->GetWindowParent() );
    name.Format( "Hidden %d", insert_position );
    new_layer->SetName( name );
    deltax= new_layer->Width() + 50;  deltay= new_layer->Height() + 60;
    point.x += deltax;
    point.y += deltay;
    new_layer->MoveTo( point );
    output_layer->MoveTo( CPoint ( output_layer->left + 90,
                                output_layer->top ) );
    // Create new neuron as a copy of the selected candidate ...
    TNeuron *new_neuron =
        new TNeuron( m_pNet->GetWindowParent(),
                    (TNeuron *)(*(Childs *)pc)[selected_candidate] );
    new_neuron->SetLearningRate( parameters->m_Learning_Rate );
    new_layer->Add( new_neuron );
    new_layer->ConnectWith( output_layer );
    m_pNet->AddAt( new_layer, insert_position++ );
    nlayers++; // the new layer was inserted
    new_layer->ConnectWith( pc );
}

```

```

// Train with a new hidden unit until patience criteria or
// no significant error change
last_error = 3.0;
Training_Error = 0.0;
for( patience = 0; patience < parameters->m_Patience &&
    i < parameters->m_Cycles; i++ )
{
    if( fabs( Training_Error - last_error ) < parameters->m_ErrorChange )
        patience++;
    last_error = Training_Error;
    SetActiveData( TrainingSet );
    TrainAllPatterns();
    Training_Error = Error_Cycle;
    UpdateGraphicError( (Error_Cycle/npatterns)/number_outputs,
        TRAINING_LINE, i, parameters->m_Cycles );
    if ( parameters->m_Validation )
        if( i % parameters->m_Validation == 0 )
            Validation(i);
    ::PumpWaitingMessages();
}
}
DestroyCorrelationMatrix();
pc->Destroy();
}

```

O algoritmo *Cascade Correlation* é o mais complexo dos algoritmos implementados. Essa complexidade se deve à necessidade de realizar muitas um grande número de operações adicionais tais como calcular a matriz de correlação antes de inserir um novo neurônio na rede.

Algoritmo Tower

A implementação do algoritmo *Tower* também esta baseada no algoritmo *BackPropagation*. O corpo do método principal deste algoritmo pode ser analisado a seguir:

```

void _convention
CTower::Run()
{register long i, j;
  double min_dif = 0.0001;
  double Training_Error = 0;
  double last_error = -1;
  CString name;
  int cont = 1;

```

```

m_pNet->ForEach( ::SetLearningRate, &parameters->m_LearningRate);

VerifyFunctions();
output_layer->ForEach( ::SetActivationFunction,
                        &TNeuron::Activation_Default );
output_layer->ForEach( ::SetOutputFunction,
                        &TNeuron::Threshold_Output );
lpfnAfterShowPattern = &CAlgorithm::Empty;
::ForEach(output_layer, &TNeuron::ChangeAuxFunction,
           &TNeuron::lpfnAfterCalcError, &TNeuron::Sum_Errors);
Error_Cycle = LONG_MAX;
//Connect input with output
input_layer->ConnectWith( output_layer );
if( parameters->m_InitWeights ) // is the flag set in the dialog ?
    InitWeights();
for( i = 0 ; Error_Cycle && i < parameters->m_Cycles ; i++ )
{
    point = CPoint(output_layer->right + 20, output_layer->top);
    last_error = Training_Error;
    Training_Error = LONG_MAX;
    for( ; fabs( Training_Error - last_error ) > min_dif &&
          i < parameters->m_Cycles &&
          Training_Error > parameters->m_MinimumSSE ; i++ )
    {
        last_error = Training_Error;
        output_layer->ApplyOnNeurons( &TNeuron::Init_Errors );
        SetActiveData( TrainingSet );
        if( parameters->m_Shuffle ) // Shuffle the data ?
            GetActiveData()->Shuffle();
        TrainAllPatterns();
        Training_Error = Error_Cycle;
        UpdateGraphicError( (Error_Cycle/npatterns)/number_outputs,
                           TRAINING_LINE, i, parameters->m_Cycles );
        if ( parameters->m_Validation )
            if( i % parameters->m_Validation == 0 )
                Validation(i);
        ::PumpWaitingMessages();
    }
    if( Training_Error > parameters->m_MinimumSSE &&
        nlayers < ( parameters->m_MaxLayers + 2 ) )
    {
        (this->*pmSpecializedConnections)();
        new_layer->MoveTo( point );
        m_pNet->Add( new_layer );
        nlayers++;
    }
}

```

```

        output_layer->ConnectWith( new_layer );
        // is the flag set in the dialog ?
        if( parameters->m_InitWeights )
            new_layer->InitWeights();
        output_layer->SetTypeLayer( HiddenLayer );
        name.Format( "Hidden %d", cont++ );
        output_layer->SetName( name );
        name.Format( "Output" );
        new_layer->SetName( name );
        // New layer becomes the output layer
        output_layer = new_layer;
        for( j = 0 ; j < number_outputs ; j++)
            ((TNeuron *)(*output_layer)[j])->SetOutputBuffer(output_values+j);
    }
}
}

```

Algoritmo Pyramid

A única diferença entre o algoritmo *Pyramid* e o *Tower* (Ver Figura 2.16, Pág. 22) é que as novas camadas são conectadas totalmente à camada de entrada e a todas as camadas anteriores. Tirando essa diferença, o algoritmo *Pyramid* é idêntico ao algoritmo *Tower*. Em vista disso, a classe *CPyramid* foi herdada da classe *CTower* e o corpo do seu método principal só precisa de duas linhas de código como pode ser observado a seguir:

```

void _convention
CPyramid::Run()
{
    pmSpecializedConnections = (LPFN_EXTRA)PyramidConnection;
    CTower::Run();
}

```

O ponteiro *pmSpecializedConnections* pertence à classe *CTower*. Esse ponteiro é utilizado para chamar algum método quando uma nova camada é criada (Ver método *CTower::Run* apresentado na Seção 4.3.3). No método *Tower*, o ponteiro *pmSpecializedConnections* aponta para um método que cria uma camada com as conexões adequadas para esse algoritmo. Para que o algoritmo *Pyramid* funcione de forma adequada é necessário apenas modificar esse ponteiro para que ele aponte para um outro método (no exemplo, o método *Pyramid-Connection*) que crie as conexões necessárias para uma rede do tipo *Pyramid*. Ou seja, que conecte a nova camada a todas as camadas anteriores.

O processo de criação de uma DLL que contenha um algoritmo pode ser um processo um pouco extenso no começo, por causa disso, foi criado o módulo para Geração e Adição automática de Novos Algoritmos que será apresentado na próxima seção.

4.4 Utilitários

Além das classes implementadas para o funcionamento do simulador, foi necessária a implementação de outras classes complementares. Entre os utilitários existentes destacam-se os seguintes módulos: Gerador de Arquivos de Padrões e Visualizador de Padrões, que serão apresentados com mais detalhes nas próximas seções.

4.4.1 Módulo de Geração de Arquivos de Padrões

O utilitário para a Geração de Arquivos de Padrões surgiu da necessidade de se particionar os arquivos de dados em arquivos de Treinamento, Validação e Teste de forma automática. Esta operação é executada através do *Menu Utilities|Generate pattern files ...*. O diálogo para realizar a geração dos arquivos pode ser observado na Figura 4.14.

Figura 4.14: Utilitário para a geração de arquivos de padrões.

Os arquivos de padrões carregados estão disponíveis em forma de lista na parte superior do diálogo (*Loaded files*) no qual pode ser escolhido o arquivo que servirá para a geração dos três subconjuntos mencionados anteriormente. O botão de *Read new Pattern File* está disponível para carregar mais arquivos de padrões sem sair do diálogo. Os nomes dos arquivos a serem gerados aparecem na parte inferior do diálogo, assim como a porcentagem de padrões que conterão em relação ao arquivo base. Na próxima seção será apresentado o módulo para visualização de padrões, que também foi implementado como parte do Simulador Kipu.

4.4.2 Módulo para a Visualização de Padrões

Este utilitário surgiu devido à necessidade de em determinadas situações apresentar graficamente os padrões utilizados para treinar as redes. Esta ferramenta é de grande utilidade porque, às vezes, é melhor ver um padrão de forma gráfica do que de forma numérica. O diálogo está disponível no *Menu Utilities|Inspect patterns*. Da mesma forma que no diálogo anterior, é possível carregar mais arquivos de padrões através do botão *Load pattern file*. Uma das entradas do diálogo refere-se ao número de colunas que servirá para desenhar o padrão no formato de matriz bidimensional. Finalmente, pode-se utilizar os botões de *Previous* e *Next* para retroceder e avançar de padrão, respectivamente. Na Figura 4.15 pode ser observado um dos padrões correspondentes a um navio que forma parte do conjunto de imagens enviados pela Marinha do Brasil que foram utilizados neste trabalho de pesquisa.

Figura 4.15: Utilitário para a visualização de padrões.

4.5 Conclusões

Neste Capítulo foi apresentado o simulador Kipu, que tem como objetivo principal servir como ferramenta de apoio para a simulação de RNC. Também, foi apresentada a Interface Gráfica do simulador Kipu, assim como o módulo de Monitoramento da Simulação. Um outro recurso importante oferecido pelo simulador Kipu é o Módulo para Geração de Código para a criação de novos algoritmos, assim como o módulo para a Geração de Código para Novos Algoritmos.

A estrutura de dados proposta neste capítulo permite a eliminação ou inserção dos elementos da rede em qualquer momento da simulação, esta última é um requisito fundamental para o

trabalho com RNC.

O fato do simulador Kipu não estar limitado a determinados modelos de Redes Neurais, faz com que o mesmo seja mais abrangente e não tenha limitações quanto ao crescimento futuro.

Para finalizar, vale a pena mencionar que mesmo sendo muito bons e úteis, nenhum dos simuladores citados nas referências [Murre 93, Thimn 94, Zell 95a, Machado 94] possui todas as características desejáveis para ser um simulador completo.

Capítulo 5

Experimentos Utilizando o Simulador Kipu

5.1 Introdução

Neste capítulo será analisado o comportamento dos diversos algoritmos implementados utilizando o simulador Kipu. Até o momento, foram implementados os seguintes algoritmos:

- *BackPropagation Padrão*;
- *Cascade Correlation*;
- *Tower*;
- *Pyramid*.

Dos quatro algoritmos mencionados anteriormente, só o primeiro algoritmo (*BackPropagation*) não é construtivo. Para poder realizar os experimentos foram utilizados 3 conjuntos de dados:

- Imagens de Navios da Marinha Brasileira;
- Dados da Marinha Americana;
- Silhuetas de Veículos.

A menos dos experimentos realizados com o algoritmo *BackPropagation*, os parâmetros dos algoritmos utilizados não foram variados visando a obtenção do melhor desempenho possível. O objetivo desses experimentos foi o de mostrar o funcionamento dos algoritmos construtivos no simulador Kipu.

Todos os experimentos foram realizados segundo os critérios apresentados no artigo Proben1 [Prechelt 94]. Cada conjunto foi dividido cinco vezes. A cada divisão foram gerados três subconjuntos com os elementos distribuídos de forma aleatória. O primeiro subconjunto contém 50% do total de elementos e foi utilizado para o treinamento das redes. O segundo subconjunto contém 25% do total, e serviu para o processo de validação. O terceiro subconjunto contém o restante dos elementos (25%) e foi utilizado para a fase de teste. Cada padrão do conjunto original pode estar em até cinco posições diferentes devido ao fato de que o conjunto original foi embaralhado antes da geração dos subconjuntos. Assim, um padrão que foi utilizado para treinar durante o primeiro experimento, não necessariamente tem que estar na mesma posição nem no mesmo subconjunto nos experimentos seguintes.

5.2 Experimentos com Dados da Marinha Brasileira

Os dados enviados pelo Instituto de Pesquisas da Marinha (IPqM) correspondem a Imagens de Navios obtidas por Radar. Este conjunto é formado por 800 imagens, cada uma delas é composta por uma matriz de 32x32 pontos, totalizando 1024 entradas. O conjunto de padrões contém navios de 4 classes. Na Figura 5.1 podem ser observados os 4 tipos de navios, um de cada classe. A seguir serão apresentadas as taxas de acerto, erro e rejeição obtidas para cada algoritmo nas fases de Treinamento, Validação e Teste.

Figura 5.1: Imagens de quatro tipos de navios enviados pela Marinha.

A classificação dos padrões da Marinha Brasileira utilizando o algoritmo *BackPropagation* foi a mais demorada, devido ao grande número de conexões que a rede possui inicialmente. Foram realizados experimentos com algumas arquiteturas tentando encontrar aquela que melhor classificasse este conjunto de dados. A rede que obteve o melhor desempenho dentre as testadas este composta por 1024 neurônios na camada de entrada, 8 na camada intermediária, e 4 na camada de saída. O número de ciclos para alcançar um nível de erro mínimo nunca foi maior que 300. Aparentemente, os padrões são facilmente separáveis e por causa disso a rede consegue minimizar o erro rapidamente. A taxa de acerto máxima alcançada durante a fase de treinamento foi de 100% (400 padrões), o que aconteceu duas vezes. A taxa mínima de acerto foi de 99,25% (397 padrões). A Tabela 5.1 ilustra as taxas médias de acerto, erro e rejeição com o desvio padrão para cada uma das fases.

BackPropagation	Treinamento	Validação	Teste
Certo	99,70 $\pm$ 0,326	98,30 $\pm$ 1,151	98,40 $\pm$ 1,557
Errado	0,00 $\pm$ 0,000	0,30 $\pm$ 0,447	0,10 $\pm$ 0,224
Desconhecido	0,30 $\pm$ 0,326	1,40 $\pm$ 0,962	1,50 $\pm$ 1,458

Tabela 5.1: Dados da Marinha Brasileira processados com o algoritmo *BackPropagation*.

O algoritmo *Cascade Correlation* também obteve um bom desempenho. Foram utilizados 60, 240, 100, 40 e 60 ciclos, respectivamente, para atingir a menor taxa de erro possível. Por causa disso, os experimentos com o algoritmo *Cascade Correlation* consumiram menos tempo do que os experimentos com o *BackPropagation*. Só em um caso foi necessário inserir uma unidade adicional, isso aconteceu exatamente naquele experimento que precisou de um maior número de apresentações do conjunto de treinamento (240 ciclos). A seguir, na Tabela 5.2, pode-se observar o desempenho do algoritmo *Cascade Correlation* com este conjunto de dados.

Cascade Correlation	Treinamento	Validação	Teste
Certo	99,55 $\pm$ 0,326	96,40 $\pm$ 1,140	97,40 $\pm$ 1,387
Errado	0,00 $\pm$ 0,000	0,10 $\pm$ 0,224	0,10 $\pm$ 0,224
Desconhecido	0,45 $\pm$ 0,326	3,50 $\pm$ 1,061	2,50 $\pm$ 1,369

Tabela 5.2: Dados da Marinha Brasileira processados com o algoritmo *Cascade Correlation*.

O algoritmo *Tower* teve um comportamento muito surpreendente. Ele alcançou uma taxa de acerto de 100% em todas as rodadas de treinamento. O número de ciclos necessários para alcançar uma taxa de acerto de 100% foi muito reduzido (8, 15, 18, 5 e 16 ciclos, respectivamente). A rede convergiu rapidamente e não foi necessário continuar treinando-a, mesmo porque não é possível que a rede melhore o desempenho. Nos cinco experimentos realizados utilizando esta rede só em um deles foi necessário inserir uma camada adicional. Na Tabela 5.3 pode ser observado o desempenho obtido pelo algoritmo *Tower* com este conjunto de dados.

Tower	Treinamento	Validação	Teste
Certo	100,00 $\pm$ 0,0	95,70 $\pm$ 2,683	96,90 $\pm$ 1,084
Errado	0,00 $\pm$ 0,0	1,40 $\pm$ 1,387	0,40 $\pm$ 0,418
Desconhecido	0,00 $\pm$ 0,0	2,90 $\pm$ 1,710	2,70 $\pm$ 0,758

Tabela 5.3: Dados da Marinha Brasileira processados com o algoritmo *Tower*.

O algoritmo *Pyramid* apresentou um comportamento similar ao algoritmo *Tower*. Esse algoritmo conseguiu alcançar 100% de acerto rapidamente. Os números de ciclos necessários para alcançar 100% de taxa de acerto durante o treinamento foram 42, 16, 10, 5 e 16 ciclos, com 4, 1, 0, 0 e 1 camadas adicionais, respectivamente. A seguir, na Tabela 5.4, pode-se observar o desempenho desse algoritmo com os dados da Marinha Brasileira.

Os resultados obtidos anteriormente indicam que a Base de Dados da Marinha Brasileira é muito bem comportada, sendo por isso de fácil classificação. Isso não é o que ocorre na

Pyramid	Treinamento	Validação	Teste
Certo	100,00 $\pm$ 0,000	97,10 $\pm$ 1,782	97,50 $\pm$ 1,000
Errado	0,00 $\pm$ 0,000	0,50 $\pm$ 0,612	0,30 $\pm$ 0,447
Desconhecido	0,00 $\pm$ 0,000	2,40 $\pm$ 1,917	2,20 $\pm$ 0,908

Tabela 5.4: Dados da Marinha Brasileira processados com o algoritmo *Pyramid*.

maioria das bases de dados reais.

Com base nos dados apresentados anteriormente, é possível comparar o desempenho de todos os algoritmos analisados em termos de treinamento, validação e teste separadamente, como pode-se observar nas próximas seções.

5.2.1 Treinamento com os Dados de Navios da Marinha Brasileira

Nesta seção, o desempenho dos quatro algoritmos na fase de treinamento é comparado. Na Tabela 5.5 são apresentadas as taxas de acerto, erro e rejeição obtidas por cada um dos algoritmos utilizando as Imagens de Navios da Marinha Brasileira.

	Certo	Errado	Desconhecido
BackPropagation	99,70 $\pm$ 0,326	0,00 $\pm$ 0,000	0,30 $\pm$ 0,326
Cascade Correlation	99,55 $\pm$ 0,326	0,00 $\pm$ 0,000	0,45 $\pm$ 0,326
Tower	100,00 $\pm$ 0,000	0,00 $\pm$ 0,000	0,00 $\pm$ 0,000
Pyramid	100,00 $\pm$ 0,000	0,00 $\pm$ 0,000	0,00 $\pm$ 0,000

Tabela 5.5: Taxa de acerto durante o treinamento utilizando os Dados de Navios da Marinha Brasileira.

A diferença nos desempenhos obtidos durante a fase de treinamento utilizando os Dados da Marinha Brasileira, pode ser melhor visualizada no gráfico da Figura 5.2.

Figura 5.2: Taxa de acerto durante a fase de treinamento utilizando os Dados de Navios da Marinha Brasileira.

De acordo com esses resultados, os algoritmos *Tower* e *Pyramid* alcançaram o máximo de desempenho possível (100%). É importante ressaltar que essa é a média de todas as rodadas. Isso significa que em todos os experimentos o algoritmo classificou corretamente a totalidade dos padrões.

5.2.2 Validação com os Dados da Marinha Brasileira

Nesta seção são comparados os desempenhos obtidos durante a fase da validação, os quais, podem ser observados na Tabela 5.6.

	Certo	Errado	Desconhecido
BackPropagation	$98,30 \pm 1,151$	$0,30 \pm 0,447$	$1,40 \pm 0,962$
Cascade Correlation	$96,40 \pm 1,140$	$0,10 \pm 0,224$	$3,50 \pm 1,061$
Tower	$95,70 \pm 2,683$	$1,40 \pm 1,387$	$2,90 \pm 1,710$
Pyramid	$97,10 \pm 1,782$	$0,50 \pm 0,612$	$2,40 \pm 1,917$

Tabela 5.6: Taxa de acerto durante a Validação utilizando os Dados de Navios da Marinha Brasileira.

A diferença nos desempenhos obtidos durante a fase de validação, com os Dados da Marinha Brasileira, pode ser melhor visualizada no gráfico da Figura 5.3.

Figura 5.3: Taxa de acerto durante a fase de Validação utilizando os Dados de Navios da Marinha Brasileira.

Os quatro algoritmos obtiveram um alto nível de desempenho durante a validação. O algoritmo que apresentou o melhor desempenho foi o *BackPropagation*, e o que apresentou pior deles foi o algoritmo *Tower*. Os algoritmos *Tower* e *Pyramid*, mesmo apresentando desempenhos elevados, não foram tão bons quanto no treinamento.

5.2.3 Teste com os Dados da Marinha Brasileira

Durante a fase de teste, o melhor desempenho foi obtido com a utilização do algoritmo *BackPropagation* e o pior com o desempenho do algoritmo *Tower*. Os algoritmos *Cascade Correlation* e *Pyramid* obtiveram resultados muito similares, como pode ser observado na Tabela 5.7.

	Certo	Errado	Desconhecido
BackPropagation	98,40 $\pm$ 1,557	0,10 $\pm$ 0,224	1,50 $\pm$ 1,458
Cascade Correlation	97,40 $\pm$ 1,387	0,10 $\pm$ 0,224	2,50 $\pm$ 1,369
Tower	96,90 $\pm$ 1,084	0,40 $\pm$ 0,418	2,70 $\pm$ 0,758
Pyramid	97,50 $\pm$ 1,000	0,30 $\pm$ 0,447	2,20 $\pm$ 0,908

Tabela 5.7: Taxa de acerto durante o Teste utilizando os Dados de Navios da Marinha Brasileira.

A diferença nos desempenhos obtidos durante a fase de teste, utilizando os Dados da Marinha Brasileira, pode ser observada na Figura 5.4.

Figura 5.4: Taxa de acerto durante a fase de Teste utilizando os Dados de Navios da Marinha Brasileira.

5.3 Experimentos com os Dados da Marinha Americana

Os dados da Marinha Americana são formados por vetores de 11 características pertencentes a 8 classes. Esse conjunto possui um total de 2545 padrões. Depois de realizadas as partições, o conjunto de treinamento é composto por 1272 padrões (50%), o conjunto de validação por 636 padrões (25%), e o conjunto de teste por 637 padrões (25%). Os experimentos foram realizados da mesma forma que no caso da Marinha Brasileira. Nesses experimentos, foram utilizados os mesmos quatro algoritmos (*BackPropagation*, *Cascade Correlation*, *Tower* e *Pyramid*).

A classificação dos padrões da Marinha Americana utilizando o algoritmo *BackPropagation* foi mais simples que no caso dos dados da Marinha Brasileira. Mais uma vez, o principal problema foi achar uma topologia que fosse ótima para maximizar a taxa de acerto. A rede que teve o melhor desempenho na média possuía uma arquitetura 11-25-8. A quantidade de ciclos utilizados variou de 2500 até 5000, dependendo do ponto no qual a rede atingiu

o seu melhor desempenho. Na Tabela 5.8 pode-se observar o desempenho do algoritmo *BackPropagation* com o conjunto de dados da Marinha Americana.

BackPropagation	Treinamento	Validação	Teste
Certo	87,90 $\pm$ 4,263	82,55 $\pm$ 3,908	82,36 $\pm$ 5,325
Errado	1,27 $\pm$ 0,715	4,47 $\pm$ 1,003	4,09 $\pm$ 1,072
Desconhecido	10,83 $\pm$ 3,707	12,99 $\pm$ 3,796	13,55 $\pm$ 4,396

Tabela 5.8: Dados da Marinha Americana processados com o algoritmo *BackPropagation*.

O algoritmo *Cascade Correlation* atingiu o seu menor erro com uma quantidade maior de ciclos que no caso do algoritmo *BackPropagation* (5400, 8200, 6000, 6000 e 6400 ciclos, respectivamente). Além disso o seu desempenho geral foi pior do que no caso do *BackPropagation*. Isso pode ser observado nas médias de acertos na fase de treinamento (72,69%), validação (69,65%) e teste (68,27%) que são menores do que as obtidas com o algoritmo *BackPropagation*. O número de unidades adicionais foi pequeno em todos os experimentos (6, 13, 8, 6 e 7, respectivamente). O treinamento foi interrompido quando a rede não melhorava o desempenho mesmo com a inserção de mais unidades. Um outro ponto que pode ser observado com relação ao desempenho desse algoritmo, é a maior taxa de padrões desconhecidos, mesmo depois da inserção de novas unidades. Isso pode significar que os padrões têm muita similaridade, dificultando sua classificação com uma certa margem de segurança. Na Tabela 5.9 pode-se observar o desempenho do algoritmo *Cascade Correlation* para este conjunto de dados.

Cascade Correlation	Treinamento	Validação	Teste
Certo	72,69 $\pm$ 3,212	69,65 $\pm$ 2,791	68,27 $\pm$ 3,915
Errado	1,79 $\pm$ 0,236	2,71 $\pm$ 0,964	2,42 $\pm$ 1,298
Desconhecido	25,52 $\pm$ 3,065	27,64 $\pm$ 3,306	29,31 $\pm$ 3,961

Tabela 5.9: Dados da Marinha Americana processados com o algoritmo *Cascade Correlation*.

Da mesma forma que nos dois algoritmos anteriores, o algoritmo *Tower* também teve problemas para diminuir o número de padrões desconhecidos durante a fase de treinamento (11,40%). O menor erro foi atingido com 10, 11, 12, 10 e 10 camadas adicionais, respectivamente. Nesses experimentos, o erro mínimo foi obtido com 2400, 2400, 5000, 1000 e 2600 ciclos, respectivamente. Na Tabela 5.10 pode-se observar o desempenho do algoritmo *Tower* com esse conjunto de dados.

Tower	Treinamento	Validação	Teste
Certo	82,76 $\pm$ 4,133	79,28 $\pm$ 4,677	80,44 $\pm$ 5,018
Errado	5,85 $\pm$ 2,617	7,64 $\pm$ 3,754	7,17 $\pm$ 3,981
Desconhecido	11,40 $\pm$ 4,685	13,08 $\pm$ 5,391	12,39 $\pm$ 5,103

Tabela 5.10: Dados da Marinha Americana processados com o algoritmo *Tower*.

O algoritmo *Pyramid* teve um comportamento discretamente superior ao algoritmo *Tower* durante o treinamento. Contudo, a diferença não foi muito significativa. Com esse algoritmo

foram necessários de 2400 a 4800 ciclos para atingir o erro mínimo, e as redes precisaram de 10, 10, 10, 12 e 12 camadas adicionais, respectivamente. Na Tabela 5.11 pode-se observar o desempenho do algoritmo *Pyramid* para o conjunto de dados da Marinha Americana.

Pyramid	Treinamento	Validação	Teste
Certo	83,76 $\pm$ 5,584	80,35 $\pm$ 1,397	78,67 $\pm$ 1,336
Errado	6,13 $\pm$ 3,610	9,12 $\pm$ 0,315	9,64 $\pm$ 1,277
Desconhecido	10,11 $\pm$ 2,450	10,54 $\pm$ 1,099	11,69 $\pm$ 0,790

Tabela 5.11: Dados da Marinha Americana processados com o algoritmo *Pyramid*.

5.3.1 Treinamento com os Dados da Marinha Americana

Nesta seção são comparados os resultados obtidos durante o processo de treinamento com a Base de Dados da Marinha Americana pelos diferentes algoritmos.

	Certo	Errado	Desconhecido
BackPropagation	87,90 $\pm$ 4,263	1,27 $\pm$ 0,715	10,83 $\pm$ 3,707
Cascade Correlation	72,69 $\pm$ 3,212	1,79 $\pm$ 0,236	25,52 $\pm$ 3,065
Tower	82,76 $\pm$ 4,133	5,85 $\pm$ 2,617	11,40 $\pm$ 4,685
Pyramid	83,76 $\pm$ 5,584	6,13 $\pm$ 3,610	10,11 $\pm$ 2,450

Tabela 5.12: Taxa de acerto durante a fase Treinamento utilizando os Dados da Marinha Americana.

Os resultados da tabela anterior, podem ser observados graficamente na Figura 5.5.

Figura 5.5: Taxa de acerto durante a fase de Treinamento utilizando os Dados da Marinha Americana.

No gráfico da Figura 5.5, pode-se observar que, o melhor desempenho obtido, em média, foi alcançado utilizando o algoritmo *BackPropagation* (87,90%). Nesses mesmos experimentos, os algoritmos *Tower* e *Pyramid* também obtiveram um bom desempenho (82,76% e 83,76%, respectivamente). O pior desempenho em média, durante a fase de treinamento, foi obtido pelo algoritmo *Cascade Correlation* (72,69%).

5.3.2 Validação com os Dados da Marinha Americana

Nesta seção podem ser observados os resultados obtidos durante o processo de validação com a Base de Dados da Marinha Americana.

	Certo	Errado	Desconhecido
BackPropagation	82,55 ± 3,908	4,47 ± 1,003	12,99 ± 3,796
Cascade Correlation	69,65 ± 2,791	2,71 ± 0,964	27,64 ± 3,306
Tower	79,28 ± 4,677	7,64 ± 3,754	13,08 ± 5,391
Pyramid	80,35 ± 1,397	9,12 ± 0,315	10,54 ± 1,099

Tabela 5.13: Taxa de acerto durante a fase Validação utilizando os Dados da Marinha Americana.

A diferença nos desempenhos obtidos durante a fase de validação, utilizando os Dados da Marinha Americana, pode ser observada na Figura 5.6.

Figura 5.6: Taxa de acerto durante a fase de Validação utilizando os Dados da Marinha Americana.

Na Figura 5.6 pode-se observar que os algoritmos *BackPropagation*, *Tower* e *Pyramid* obtiveram um nível de desempenho similar (82,55%, 79,28%, e 80,35%, respectivamente), mas o algoritmo *Cascade Correlation* apresentou um desempenho bem abaixo dos outros algoritmos (69,65%).

5.3.3 Teste com os Dados da Marinha Americana

Nesta seção podem ser observados os resultados obtidos durante o processo de teste com a Base de Dados da Marinha Americana.

	Certo	Errado	Desconhecido
BackPropagation	82,36 $\pm$ 5,325	4,09 $\pm$ 1,072	13,55 $\pm$ 4,396
Cascade Correlation	68,27 $\pm$ 3,915	2,42 $\pm$ 1,298	29,31 $\pm$ 3,961
Tower	80,44 $\pm$ 5,018	7,17 $\pm$ 3,981	12,39 $\pm$ 5,103
Pyramid	78,67 $\pm$ 1,336	9,64 $\pm$ 1,277	11,69 $\pm$ 0,790

Tabela 5.14: Taxa de acerto durante a fase de Teste utilizando os Dados da Marinha Americana.

Os mesmos dados da Figura 5.14 podem ser observados no gráfico de barras da Figura 5.7.

Figura 5.7: Taxa de acerto durante a fase de Teste utilizando os Dados da Marinha Americana.

Da mesma forma que no treinamento e na validação, o algoritmo *BackPropagation* foi o que obteve o melhor desempenho (82,36%). Os algoritmos *Tower* e *Pyramid* também atingiram um nível de acerto similar (80,44, e 78,67%, respectivamente). Finalmente, como já era previsível, o algoritmo *Cascade Correlation* obteve a menor taxa de acerto em média dentre os quatro algoritmos (68,27%).

5.4 Experimentos com Dados de Silhuetas de Veículos

O conjunto de dados de Silhuetas de Veículos é proveniente do *Turing Institute em Glasgow*, na Escócia. Esta base de dados é formada por 846 padrões. Cada um dos padrões é representado por um vetor de 18 características extraídas de imagens de Silhuetas de Veículos que pertencem a uma entre quatro classes. Segundo a literatura obtida junto à base de dados, esse conjunto de dados é particularmente difícil de classificar, conforme mostram as tabelas de resultados a seguir.

Durante os experimentos utilizando o algoritmo *BackPropagation*, foi necessário testar diversas arquiteturas antes de se encontrar a que obteve a maior taxa de acerto. Finalmente, a rede que melhor se comportou com este conjunto de dados possuía uma arquitetura com 18,

15 e 4 neurônios na camada de entrada, intermediária e de saída, respectivamente. Esse algoritmo, o *BackPropagation*, foi o que melhor se comportou na fase de treinamento (86,48%). Não aconteceu o mesmo nas fases de validação e de teste, onde as taxas de acerto, em média foram de 65,09% e 66,35%, respectivamente. Para os experimentos com essa rede foram utilizados 1800, 10000, 10000, 1800 e 3400 ciclos, respectivamente. Na Tabela 5.15 pode-se observar o desempenho do algoritmo *BackPropagation* com esta base de dados.

BackPropagation	Treinamento	Validação	Teste
Certo	86,48 ± 5,413	65,09 ± 2,083	66,35 ± 2,701
Errado	1,56 ± 1,077	14,72 ± 2,526	14,22 ± 2,979
Desconhecido	11,96 ± 4,614	20,19 ± 1,836	19,43 ± 2,616

Tabela 5.15: Dados de Silhuetas de Veículos processados com o algoritmo *BackPropagation*.

Dentro dos quatro algoritmos, o algoritmo *Cascade Correlation* foi mais uma vez, o que obteve pior desempenho nas três fases dos experimentos. Esse mesmo algoritmo, foi o que apresentou mais problemas para diminuir o número de padrões classificados como de classe desconhecida (35,84%). Isso pode ter ocorrido devido ao fato de que os padrões de duas das classes apresentarem muita similaridade. A quantidade de ciclos necessários para atingir as taxas de acerto máximas junto a este algoritmo foram 3500, 7500, 10000, 10000, 10000, respectivamente. Também foram necessárias 13, 12, 16, 18 e 38 unidades adicionais, respectivamente. Na Tabela 5.16 pode-se observar o desempenho do algoritmo *Cascade Correlation* com os dados de Silhuetas de Veículos.

Cascade Correlation	Treinamento	Validação	Teste
Certo	60,43 ± 3,364	55,47 ± 3,226	54,88 ± 1,936
Errado	3,74 ± 0,676	5,66 ± 1,598	7,01 ± 0,700
Desconhecido	35,84 ± 3,610	38,87 ± 3,691	38,10 ± 1,367

Tabela 5.16: Dados de Silhuetas de Veículos processados com o algoritmo *Cascade Correlation*.

Durante a fase de treinamento, o algoritmo *Tower* obteve uma taxa de acerto, em média, de 74,28%. O mesmo desempenho não foi obtido durante as fases de validação e teste, onde as médias foram de 66,79% e 65,03%, respectivamente. Nesses experimentos foram necessários 5000, 4000, 5000, 3500 e 10000 ciclos, onde foram criadas 20, 20, 20, 14 e 10 camadas adicionais, respectivamente. Na Tabela 5.17 pode-se observar o desempenho do algoritmo *Tower* com esse conjunto de dados.

Tower	Treinamento	Validação	Teste
Certo	74,28 ± 5,646	66,79 ± 5,952	65,03 ± 7,543
Errado	13,38 ± 4,079	19,06 ± 2,917	17,91 ± 5,876
Desconhecido	12,34 ± 6,420	14,15 ± 6,086	17,06 ± 12,049

Tabela 5.17: Dados de Silhuetas de Veículos processados com o algoritmo *Tower*.

Durante a fase de treinamento, o algoritmo *Pyramid* obteve a segunda melhor taxa de acerto em média dentre os quatro algoritmos (84,26%). Em alguns dos cinco experimentos com este

algoritmo, o treinamento foi finalizado porque observou-se que os padrões estavam sendo decorados pela rede. Isso está refletido na baixa taxa de acerto obtida na fase de validação (68,87%). Nesses experimentos foram necessários 2000, 5000, 3400, 1400 e 5000 ciclos. Também foi necessário inserir 20, 10, 20, 20 e 10 camadas adicionais, respectivamente. Na Tabela 5.18 pode-se observar o desempenho do algoritmo *Pyramid* com a base de dados de Silhuetas de Veículos.

Pyramid	Treinamento	Validação	Teste
Certo	84,26 $\pm$ 4,041	68,87 $\pm$ 5,739	67,87 $\pm$ 4,438
Errado	9,97 $\pm$ 2,465	24,34 $\pm$ 4,187	24,93 $\pm$ 2,476
Desconhecido	5,77 $\pm$ 2,766	6,80 $\pm$ 2,421	7,20 $\pm$ 2,690

Tabela 5.18: Dados de Silhuetas de Veículos processados com o algoritmo *Pyramid*.

5.4.1 Treinamento com os Dados de Silhuetas de Veículos

Nesta seção podem ser observados os resultados obtidos durante o processo de treinamento com a base de dados de Silhuetas de Veículos pelos diferentes algoritmos implementados.

	Certo	Errado	Desconhecido
BackPropagation	86,48 $\pm$ 5,413	1,56 $\pm$ 1,077	11,96 $\pm$ 4,614
Cascade Correlation	60,43 $\pm$ 3,364	3,74 $\pm$ 0,676	35,84 $\pm$ 3,610
Tower	74,28 $\pm$ 5,646	13,38 $\pm$ 4,079	12,34 $\pm$ 6,420
Pyramid	84,26 $\pm$ 4,041	9,97 $\pm$ 2,465	5,77 $\pm$ 2,766

Tabela 5.19: Taxa de acerto durante o Treinamento utilizando os Dados de Silhuetas de Veículos.

A seguir, na Figura 5.8 são apresentados os dados da Tabela anterior graficamente.

Figura 5.8: Taxa de acerto durante a fase de Treinamento com os Dados de Silhuetas de Veículos.

Durante o processo de treinamento, pode-se observar que o algoritmo que obteve o melhor desempenho foi o *BackPropagation* (86,48%), seguido de perto pelo algoritmo *Pyramid* com 84,26%. Um outro dado importante é que o algoritmo *Tower* teve a maior taxa de padrões classificados de forma errada (13,38%). Finalmente, o algoritmo *Cascade Correlation* foi o que teve a maior proporção de padrões com classe indefinida (35,84%).

5.4.2 Validação com os Dados de Silhuetas de Veículos

Nesta seção podem ser observados os resultados obtidos durante o processo de validação com a base de dados de Silhuetas de Veículos.

	Certo	Errado	Desconhecido
BackPropagation	65,09 $\pm$ 2,083	14,72 $\pm$ 2,526	20,19 $\pm$ 1,836
Cascade Correlation	55,47 $\pm$ 3,226	5,66 $\pm$ 1,598	38,87 $\pm$ 3,691
Tower	66,79 $\pm$ 5,952	19,06 $\pm$ 2,917	14,15 $\pm$ 6,086
Pyramid	68,87 $\pm$ 5,739	24,34 $\pm$ 4,187	6,80 $\pm$ 2,421

Tabela 5.20: Taxa de acerto durante o Validação utilizando os Dados de Silhuetas de Veículos.

A diferença nas taxas de acerto obtidas durante a fase de validação, pode ser observada na Figura 5.9.

Figura 5.9: Taxa de acerto durante a fase de Validação com os Dados de Silhuetas de Veículos.

Os quatro algoritmos obtiveram um baixo desempenho durante a validação. Como o algoritmo *BackPropagation* foi o melhor na fase de treinamento (86,48%), esperava-se que obtivesse um melhor desempenho na validação, mas isso não aconteceu (só obteve 65,09%). O algoritmo que obteve a melhor taxa de acerto, em média, nesta fase. A média obtida foi o *Pyramid* com 68,87%, seguido de perto pelo algoritmo *Tower* com 66,79%. Finalmente, o algoritmo *Cascade Correlation* foi o que obteve o pior comportamento obtendo apenas 55,47% de taxa de acerto. Esse último algoritmo também foi o que obteve a maior quantidade de padrões com classe desconhecida (38,87%).

5.4.3 Teste com os Dados de Silhuetas de Veículos

Nesta seção podem ser observados os resultados obtidos durante a fase de teste com a base de dados de Silhuetas de Veículos.

	Certo	Errado	Desconhecido
BackPropagation	66,35 ± 2,701	14,22 ± 2,979	19,43 ± 2,616
Cascade Correlation	54,88 ± 1,936	7,01 ± 0,700	38,10 ± 1,367
Tower	65,03 ± 7,543	17,91 ± 5,876	17,06 ± 12,049
Pyramid	67,87 ± 4,438	24,93 ± 2,476	7,20 ± 2,690

Tabela 5.21: Taxa de acerto durante o Teste utilizando os Dados de Silhuetas de Veículos.

Na Figura 5.10 pode ser visualizada com mais clareza, a diferença nas taxas de acerto obtidas pelos quatro algoritmos na fase de teste.

Figura 5.10: Taxa de acerto durante a fase de Teste com os Dados de Silhuetas de Veículos.

Durante a etapa de teste, os quatro algoritmos obtiveram baixas taxas de acerto. O que melhor se comportou foi o algoritmo *Pyramid* com 67,87%, seguido pelo algoritmo *Back-Propagation* com 66,35%, o *Tower* com 65,03% e, finalmente, o *Cascade Correlation* com uma média de 54,88% de taxa de acerto.

5.5 Conclusões

Neste capítulo foram apresentados os resultados obtidos em experimentos realizados utilizando o algoritmo *BackPropagation* e mais três algoritmos construtivos (*Cascade Correlation*, *Tower*, e *Pyramid*). Todos os testes foram realizados utilizando como ferramenta para a simulação o Simulador Kipu, que continua sendo desenvolvido como parte deste trabalho.

Para realizar os experimentos foram utilizados três conjuntos de dados: Imagens de Navios da Marinha Brasileira, Imagens de Navios na forma de vetor de características da Marinha Americana, e Imagens de Silhuetas de Veículos também na forma de vetor de características. Cada conjunto de dados foi dividido cinco vezes de forma aleatória para formar os conjuntos de Treinamento (50%), Validação (25%) e Teste (25%). Para criar os subconjuntos foi utilizado o módulo de **Geração de Arquivos Padrões** (Seção 4.3.2). Cada algoritmo foi utilizado cinco vezes, obtendo a média e o desvio padrão para cada um deles. As arquiteturas das redes *BackPropagation* apresentadas neste capítulo são resultado de muitas tentativas

para encontrar aquela que melhor se comportasse em média para cada uma das três bases de dados.

O pior desempenho apresentado pelos algoritmos construtivos de forma geral, se justificam pelo fato dos seus parâmetros não terem sido variados buscando o melhor desempenho ao contrário do que foi feito no algoritmo *BackPropagation*.

Genericamente, pode-se observar que os experimentos que utilizaram os dados da Marinha Brasileira foram os que obtiveram os melhores resultados. Isso sugere que essa Base de Dados é mais bem comportada que as demais. Com esse mesmo conjunto de dados foi possível observar que os Algoritmos Construtivos (especialmente os algoritmos *Tower* e *Pyramid*) atingiram 100% de taxa de acerto durante o treinamento, mas não conseguiram 100% nem na fase de validação nem na fase de teste. Esse comportamento pode ser atribuído ao fato de que o algoritmo aprende muito bem o conjunto de treinamento mas não apresenta uma grande capacidade de generalização.

Capítulo 6

Conclusões

Este trabalho de pesquisa diz respeito a construção do Simulador Kipu para o desenvolvimento, simulação e análise de Redes Neurais Construtivas. Esse simulador, será inicialmente utilizado para o Reconhecimento de Padrões. O trabalho desenvolvido nesta dissertação faz parte do Projeto SAPRI, cujo objetivo principal é a Automação do Processo de Aquisição, Processamento e Reconhecimento das Imagens de Navios capturadas através de Radar. Este projeto é um projeto PROTEM/CNPq que está sendo desenvolvido junto a outras universidades e ao Instituto de Pesquisas da Marinha do Brasil.

No Capítulo 2 foram apresentadas as Redes Neurais Construtivas. Esse tipo de rede tem a característica de inserir novos elementos durante a fase de treinamento. Alguns algoritmos construtivos foram apresentados nesse capítulo, tais como: *Cascade Correlation* (Ver Seção 2.3), *UpStart* (Ver Seção 2.4), *Tiling e MTiling* (Ver Seção 2.5), *Tower* (Ver Seção 2.6), *Pyramid* (Ver Seção 2.7), e *RBF-DDA* (Ver Seção 2.8). As RNC possuem características muito diferentes das redes tradicionais e portanto devem ser tratadas com muito mais cuidado. A falta de ferramentas apropriadas para o trabalho com RNC, motivou a construção do Simulador *Kipu* que foi modelado visando o trabalho com esse tipo de redes.

No Capítulo 3 são descritos alguns dos simuladores mais utilizados pela comunidade científica. Na Seção 3.2 são descritas as fases e alguns recursos que podem ser oferecidos por um simulador para que ele possa ser considerado uma eficiente ferramenta de apoio. Dentre os simuladores mais utilizados atualmente podem ser citados os seguintes simuladores: *Stuttgart Neural Network Simulator (SNNS)* (Ver Seção 3.3), *Open Simulator* (Ver Seção 3.4), e *Sesame* (Ver Seção 3.5). Foi visto também nesse capítulo que só o primeiro deles, o SNNS, tem algoritmos construtivos implementados: o *Cascade Correlation* e o *RBF-DDA*, sendo que, o primeiro desses algoritmos apresentou muitos problemas. Também podem ser encontradas no simulador SNNS implementações da família ART que são considerados algoritmos construtivos por alguns autores.

No Capítulo 4 foi apresentado o simulador Kipu, que é parte deste trabalho de pesquisa. Na Seção 4.2 apresentou-se a Interface Gráfica do Usuário do simulador Kipu. Dentro da Interface Gráfica do Usuário foram consideradas a Interface para a Criação de Redes (Ver Seção 4.2.1), e a Interface para o Controle dos Parâmetros da Simulação (Ver Seção 4.2.2).

Existe mais uma parte da Interface (Interface para o Monitoramento da Simulação) que foi apresentada na Seção 4.2.3.

A Seção 4.3 (Implementação do Simulador Kipu) apresentou as características do simulador do ponto de vista da implementação. A estrutura de dados utilizada pelo simulador é apresentada na Seção 4.3.1. Um Módulo de Geração de Novos Algoritmos é discutido que é apresentado na Seção 4.3.2. Esse módulo permite gerar a base para um novo projeto contendo uma nova classe herdada de algum dos algoritmos já existentes. Na Seção 4.3.3 foram apresentados os métodos necessários para o funcionamento de cada um dos algoritmos implementados no simulador. Na Seção 4.4 são apresentados os utilitários implementados dentro do Simulador Kipu. Na Seção 4.4.1 foi apresentado o módulo para Geração de Arquivos de Padrões, finalmente, na Seção 4.4.2 foi apresentado o Módulo para a Visualização de Padrões que permite ao usuário visualizar os padrões de entrada.

O Capítulo 5 foi dedicado a experimentos de Reconhecimento de Padrões realizados utilizando os quatro algoritmos implementados no simulador Kipu (*Backpropagation*, *Cascade Correlation*, *Tower*, e *Pyramid*). Para a realização dos experimentos, foram utilizados três conjuntos de dados: Imagens de Navios capturadas por radar da Marinha Brasileira, vetores de características de Imagens de Navios da Marinha Americana, e vetores de características de Silhuetas de Veículos. Nesse mesmo capítulo foi realizada uma análise dos processos de treinamento, validação e teste por separado para cada uma das três Bases de Dados. Dessa forma foi possível comparar o desempenho obtido pelos quatro algoritmos sob as mesmas condições.

6.1 Contribuições

A construção de um simulador de RNAs não é uma tarefa fácil. Após a realização dessa etapa do projeto, pode-se concluir que o esforço e tempo dedicado a ele foram amplamente compensados com a quantidade de conhecimento adquirido na fase de desenvolvimento.

Uma outra conclusão importante foi a observação de que as Redes Neurais Construtivas testadas nem sempre obtiveram a mesma capacidade de generalização que o algoritmo *BackPropagation*. Essa diferença de desempenhos pode ser observada com mais clareza comparando a fase de treinamento com as de validação e teste dos algoritmos *Cascade Correlation*, *Tower* e *Pyramid* (Ver Tabelas 5.2, 5.3, e 5.4, respectivamente). Contudo, deve ser lembrado que neste trabalho não foi se buscou otimizar o desempenho dessas redes.

A realização deste trabalho de pesquisa contribuiu ao estudo das técnicas envolvidas no trabalho com Redes Neurais Construtivas. Em alguns casos (por exemplo, no caso dos algoritmos *Tower* e *Pyramid*) não foi possível encontrar uma implementação em *software*. Tais algoritmos tiveram que ser implementados integralmente para que os experimentos pudessem ser realizados.

Outra contribuição deste trabalho foi a inclusão de facilidades que possibilitam a extensão do simulador. A técnica de criar componentes, e depois incorporá-los a um aplicativo de forma transparente é também conhecida como COM (*Component Object Module*). Essa técnica

é vista como uma das tecnologias de ponta no que se refere a Engenharia de *Software*. Utilizando essa técnica, foi possível criar os algoritmos dentro de DLLs e depois adicioná-los ao simulador sem problemas. Portanto, este trabalho contribui também, para o conhecimento dessa técnica e sua utilização.

Os experimentos realizados neste trabalho de pesquisa, contribuem também ao melhoramento das técnicas utilizadas atualmente pela Marinha do Brasil para o Reconhecimento de Alvos, possibilitando o desenvolvimento de um Sistema Inteligente para o Controle da Navegação.

O Simulador Kipu pode ser integrado ao trabalho realizado atualmente pela Marinha do Brasil, contribuindo dessa forma, ao avanço nas pesquisas referentes à aplicação de Redes Neurais para o Reconhecimento de Alvos.

A criação do simulador Kipu contribuiu também para o desenvolvimento do trabalho de iniciação científica do aluno Humberto de Sousa. Esse trabalho consistiu na implementação e aprimoramento dos algoritmos construtivos já apresentados anteriormente.

Este trabalho contribuiu também com escrita de três artigos, dois deles internacionais. O primeiro deles intitulado *A Multi-Threaded Object Oriented Simulator for Ontogenic Neural*, foi publicado na *International Conference on Computational Intelligence and Multimedia Applications* organizada pela *Monash University*, e realizada em Churchill, Victoria-Australia em fevereiro deste ano.

O segundo artigo intitulado "Utilização de Redes Neurais Construtivas para o Reconhecimento de Padrões" será apresentado na "Segunda Jornada Latino-americana de Sensoriamento Remoto por Radar: Técnicas de Processamento de Imagens" que se realizará em Santos-SP em Setembro deste ano. Esse mesmo artigo, foi convidado para ser publicado em inglês junto à Agência Espacial Européia (*European Space Agency (ESA)*).

O terceiro artigo foi intitulado "Um Simulador Multi-Threads de Redes Neurais Ontogênicas Orientado a Objetos" que foi publicado e apresentado no *IV Brazilian Symposium on Neural Networks* que foi realizado na cidade de Goiânia-GO, em dezembro de 1997.

Finalmente foram/estão sendo criados três Relatórios Técnicos. O primeiro deles intitulado "Simulador Multi-Thread de Redes Neurais Ontogênicas" (Relatório Técnico Nro 58) publicado junto ao ICMC/USP. O segundo dos relatórios técnicos se refere às Redes Neurais Construtivas, e o terceiro é intitulado "Kipu: Um Simulador de Redes Neurais Construtivas". Esses relatórios estão sendo submetidos para publicação junto ao ICMC/USP durante o mês de agosto de 1998.

6.2 Dificuldades Encontradas

Dentre as dificuldades encontradas na realização deste trabalho, pode-se mencionar principalmente o fato de que o projeto inicial envolveu dois subprojetos. O primeiro deles consistiu na construção de um simulador que pudesse trabalhar com RNC, e o segundo subprojeto consistiu em investigar o comportamento das RNC na tarefa de Reconhecimento de Alvos. Considerando essa dificuldade, um dos principais problemas foi a falta de tempo para

cumprir com o cronograma estabelecido. Apesar desse problema o cronograma foi cumprido integralmente.

Um outro problema foi ter que dominar a Hierarquia de classes MFC 4.2 (*Microsoft Foundation Classes*) que foi utilizada desde o início da implementação. O problema principal é que essa hierarquia provê uma grande quantidade de classes já implementadas, sendo complicado dominar todas elas. Esse problema foi superado depois de muita prática de forma constante e com a leitura de textos relativos ao tema.

No Sistema Operacional Windows 95/NT quando o processador esta sendo utilizado para realizar operações matemáticas, qualquer figura que estiver sendo desenhada na tela não aparece. Isto ocorre devido ao fato de que as tarefas de desenhar na tela têm menos prioridade que as de processar na CPU. Esse problema acontece no simulador quando uma rede está sendo treinada e ao mesmo tempo é necessário que o usuário veja o gráfico da curva de erro. O problema de desenhar na tela ao mesmo tempo em que um processamento está sendo realizado pode ser resolvido com a criação de uma *Thread* cuja única função é a de desenhar na tela. O problema de sincronização das *Threads* também foi um problema encontrado durante este trabalho de pesquisa.

Uma outra dificuldade foi o aprendizado das técnicas utilizadas para a construção de componentes (COM). Esse entrave pôde ser superado graças ao estudo de trabalhos similares que utilizavam a mesma técnica, fora do Brasil.

6.3 Trabalhos Futuros

No futuro, pretende-se implementar outros algoritmos construtivos, tais como: *MTiling*, *UpStart*, *RBF-DDA*, *ART1*, *ART2*, assim como outros algoritmos convencionais.

Um outro módulo que poderia ser de grande utilidade é o de leitura de dados de arquivos com outros formatos tais como: Planilhas de Cálculo do *MS-EXCEL*, e arquivos de texto.

Os algoritmos existentes (especialmente o *Cascade Correlation*) precisam ser otimizados, com o objetivo de reduzir a quantidade de código necessário para suas implementações.

Uma das novas tendências dentro da área de Redes Neurais se refere à combinação de vários módulos de RNAs dentro de um mesmo sistema. A criação de um módulo para realizar essa tarefa seria muito interessante.

Uma vez treinada uma RNA, é interessante otimizá-la, quer dizer eliminar aqueles elementos (camadas, neurônios ou conexões) que não influenciem na resposta da rede. Essa tarefa poderia ser realizada por um módulo que conteria técnicas de *Pruning*.

A Extração de Regras também poderia ser um aspecto interessante a ser investigado dentro do simulador. Esse módulo seria de grande utilidade para explicar as respostas fornecidas pela rede.

Entre os trabalhos de pesquisa que estão sendo realizados atualmente no LABIC-ICMC, existe um em particular que está desenvolvendo um sistema de navegação autônomo para

um robô móvel. Para tanto, pretende-se implementar um módulo no simulador que permita o trabalho com Redes Neurais Auto-Organizáveis. Um outro módulo que seria de grande interesse para o usuário de Redes Neurais se refere ao Processamento de Séries Temporais.

Os módulos de análise e apresentação dos resultados em forma gráfica, também poderiam ser melhorados com a criação de outras ferramentas, e com uma melhor sincronização com a *Thread* que desenha o gráfico na tela.

Referências Bibliográficas

- [Berthold 95] BERTHOLD, M. R. DIAMOND J. *Boosting the Performance of RBF Networks with Dynamic Decay Adjustment*. Proceedings of the 1994 Conference: Advances in Neural Information Processing Systems 7, Morgan Kaufmann. pp. 521-528. MIT Press. 1995.
- [Bowden 95] BOWDEN, W. *Using the The Standard Template Library with Borland C++ [OnLine]*. Janeiro 1996 [citado 20 Abril 1997]. Disponível na WWW: <URL:<http://www.cs.rpi.edu/~wiseb/stl-borland.html>>.
- [Carvalho 97] CARVALHO, A.; FAIRHURST, M. C.; BISSET, D. L. *Combining Boolean Neural Architectures for Image Recognition*. Connection Science. Vol. 9. Nro. 4. pp. 405-418. 1997.
- [Cuadros 97] CUADROS, E. V.; ÁLVAREZ M. V.; CARVALHO A.C.P.L.F. *Simulador Multi-Threads de Redes Neurais Ontogênicas Orientado a Objetos*. Relatório Técnico No. 59. Instituto de Ciências Matemáticas de São Carlos. Universidade de São Paulo, São Carlos-SP, Brasil, 1997.
- [Fahlman 90] FAHLMAN, S. E.; LABIERE, C. *The Cascade Correlation Learning Architecture*. School of Computer Science, Carnegie Mellon University. 1990.
- [Frean 89] FREAN, M. *The UpStart Algorithm: A method for constructing and training feed-forward Neural Networks*. PhD Thesis. Center for Cognitive Science. University of Edinburgh, UK. 1989.
- [Frean 90] FREAN, M. *Small Nets and Short Paths: Optimizing Neural Computation*. PhD Thesis. Center for Cognitive Science. University of Edinburgh, UK. 1990.
- [Freeman 93] FREEMAN, J.; SKAPURA, D. *Redes Neuronales, Algoritmos, Aplicaciones y Técnicas de Programación*. Addison Wesley Iberoamericana. 1993.
- [Fritzke 95a] FRITZKE, B. *A Growing Neural Gas Network Learns Topologies*. Proceedings of the 1994 Conference: Advances in Neural Information Processing Systems 7, Morgan Kaufmann. pp. 625-632. MIT Press. 1995.

- [Fritzke 95b] FRITZKE, B. *Incremental Learning of Local Linear Mappings*. Proceedings of the International Conference on Artificial Neural Networks (ICANN 94). 1994. pp. 217-222. 1994.
- [Gallant 90] GALLANT, S.I. *Perceptron Based Learning Algorithms*. IEEE Transactions on Neural Networks. Vol. I, No. 2, pp. 179-192. June 1990.
- [Gallant 93] GALLANT, S.I. *Neural Network Learning and Expert Systems*. Cambridge, Mass MIT Press. 1993.
- [Halgamuge 97] HALGAMUGE, S. *Self-Evolving neural Networks for Rule-Based Data Processing*. IEEE Transaction on Signal Processing. Vol. 45. Nro. 11. November 1997.
- [Hudak 92] HUDAK, M.J. *Rce Classifiers: Theory and Practice*. Cybernetics and Systems. pp 483-515. 1992.
- [Karnin 90] KARNIN, E. *A simple procedure for pruning back-propagation Neural Networks*. IEEE Transsaction on Neural Networks. Vol. 1. Nro. 2. pp. 239-242. Junho. 1990.
- [Kirman 95] KIRMAN, J. *A Modest Standard Template Library [On-Line]*. 1995 [citado 15 Maio 1997]. Disponível na WWW: <URL:<http://www.cs.brown.edu/people/jak/programming/stl-tutorial/tutorial.html>>.
- [Kock 96] KOCK G.; SERBEDZIJA N. B. *Simulation of Artificial Neural Networks*. Gordon & Breach Science Publishers. SAMS Vol. 27, pp. 15-59. 1996.
- [Leber 92] LEBER J.F.; MOSCHYTZ G. S. *An Acoustical Signal Recognizer Implemented on a Novel Interactive Object Oriented Neural Network Simulator*. Artificial Neural Networks, No. 2, pp 1291-1294. North-Holland/Elsevier, Amsterdam, The Netherlands, Set. 4-7. 1992.
- [Leber 93] LEBER J.F.. *The Recognition of Acoustical Signals Using Neural Networks and an Open Simulator*. Series Microelectronics, No. 20. Konstanz, Alemanha. 1993.
- [Linden 92] LINDEN A.; TIETZ C. *A Software Environment for Combining Multiple Neural Network Paradigms and Applications*. Artificial Neural Networks, No. 2, pp. 1265-1268. North-Holland/Elsevier, Amsterdam, The Netherlands, Set. 4-7. 1992.
- [Machado 94] MACHADO, P. D. L.; CARVALHO FILHO, E. C. B.; MEIRA, S. R. L.; GOMES, H. M. *Um Ambiente para Modelagem, Simulação e Análise de Redes Neurais Artificiais*. Anais do I Simpósio Brasileiro de Redes Neurais. pp. 43-48. 1994.

- [Mézard 89] MEZARD, M.; NADAL, J. *Learning in Feedforward Layered Networks: The Tiling Algorithm*. J. Phys. A: Math and Gen 22. No. 12, pp. 2191-2203. 1989.
- [Moody 89] MOODY, J. E. DARKEN, C. J. *Fast Learning in Networks of Locally-Tuned Processing Units*. Neural Computation, Vol.1. pp.281-294. 1989.
- [Mozer 89] MOZER, M.C.; SMOLENSKY, P. *Using Relevance to Reduce Network Size Automatically*. Connection Science. Vol. 1. pp. 3-16. 1989.
- [Mumit 95] MUMIT, K. *Mumit's Standard Template Library (STL) Newbie Guide [OnLine]*. 1995 [citado 10 Junho 1997]. Disponível na WWW: <URL:<http://www.xraylith.wisc.edu/~khan/software/stl/STL.newbie.html>>.
- [Murre 93] MURRE, J. *Neurosimulators, Handbook of Brain Research and Neuron Networks*. MIT Press. 1995.
- [O'Reilly 96a] O'REILLY & ASSOCIATES. *Dictionary of PC Hardware and Data Communications Terms [OnLine]*. 1996 [citado 25 Maio 1997]. Disponível na WWW: <URL:http://www.ora.com/reference/dictionary/terms/D/Dynamic_Link_Library.htm>.
- [O'Reilly 96b] O'REILLY & ASSOCIATES. *Dictionary of PC Hardware and Data Communications Terms [OnLine]*. 1996 [citado 25 Maio 1997]. Disponível na WWW: <URL:http://www.ora.com/reference/dictionary/terms/D/Dynamic_Data_Exchange.htm>.
- [O'Reilly 96c] O'REILLY & ASSOCIATES. *Dictionary of PC Hardware and Data Communications Terms [OnLine]*. 1996 [citado 25 Maio 1997]. Disponível na WWW: <URL:http://www.ora.com/reference/dictionary/terms/O/Object_Linking_and_Embedding.htm>.
- [Parekh 95] PAREKH, R.; YANG, J.; HONAVAR, V. *Constructive Neural Network Learning Algorithms for Multi-Category Pattern Classification*. Technical Report TR95-15a. 1995.
- [Poullard 95] POULLARD, H. *Barycentric Correction Procedure: A Fast Method of Learning Threshold Units*. Proceedings of the World Congress on Neural Networks (WCNN 94). Vol. I. pp 710-713. Washington D.C. Julho 1995.
- [Prechelt 94] PRECHELT, L. *PROBEN1 - A Set of Neural Network Benchmark Problems and Benchmarking Rules*. Technical Report 21/94. Fakultät für Informatik, Universität Karlsruhe. Alemanha. Setembro 1994.

- [Prechelt 95] PRECHELT, L. *Adaptive Parameter Pruning in Neural Networks*. Technical Report 95-009. International Computer Science Institute. Berkeley, CA, Março 1995.
- [Reilly 82] REILLY, D.L.; COOPER, D.L.; ELBAUM, C. *A neural model for category learning*. Biol. Cybernet. Vol 45. 1982.
- [Rumelhart 86] RUMELHART, D. E.; Mc CLELLAND, J. L. *Parallel Distributed Processing*. MIT Bradford Press. 1986.
- [SAPRI 97] *Sistema para Aquisição, Processamento e Reconhecimento de Imagens (SAPRI) [OnLine]*. 1997 [citado 15 Janeiro 1997]. Disponível na WWW: <URL:<http://www.di.ufpe.br/~sapri>>.
- [Tenorio 89] TENORIO, M. F.; LEE, W. T. *Self-Organizing Neural Nets for the Identification Problem*. Advances in Neural Information Processing Systems 1. Morgan Kaufmann. D.S. Touretzky (ed.). 1989.
- [Thimn 94] THIMN, G.; GRAU, R.; FIESLER, E. *Modular Object-Oriented Neural Networks Simulators and Topology Generalization*. Proceedings of the International Conference on Artificial Neural Networks (ICANN 94). Springer-Verlag. Vol I. Part 2: Mathematical Model. pp. 26-29; 747-750. Maio 1994.
- [Venter 97] VENTER, H. *Abstract Data Types (ADT) [OnLine]*. 1997 [citado 15 Maio 1997]. Disponível na WWW: <URL:<http://www.cs.upe.ac.za/staff/csabhv/slim/adt.html>>.
- [Zell 95a] ZELL, A. *SNNS Stuttgart Neural Network Simulator User Manual Version 4.1*. Report No. 6/95. 1995.
- [Zell 95b] ZELL, A. *Dynamic Decay Adjustment for RBFs [On-Line]*. 1995 [citado 15 Abril 1998]. Disponível na WWW: <URL:<http://www.informatik.uni-stuttgart.de/ipvr/bv/projekte/snns/UserManual/node192.html>>.