

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO



**PREDIÇÃO DE TRÁFEGO, USANDO REDES NEURAIS
ARTIFICIAIS, PARA GERENCIAMENTO ADAPTATIVO DE
LARGURA DE BANDA EM ROTEADORES**

TIAGO PRADO OLIVEIRA

Uberlândia - Minas Gerais

2014

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO



TIAGO PRADO OLIVEIRA

**PREDIÇÃO DE TRÁFEGO, USANDO REDES NEURAIS
ARTIFICIAIS, PARA GERENCIAMENTO ADAPTATIVO DE
LARGURA DE BANDA EM ROTEADORES**

Dissertação de Mestrado apresentada à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como parte dos requisitos exigidos para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Sistemas de Computação.

Orientador:

Prof. Dr. Jamil Salem Barbar

Co-orientador:

Prof. Dr. Alexsandro Santos Soares

Uberlândia, Minas Gerais
2014

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da UFU, MG, Brasil.

O48p
2014

Oliveira, Tiago Prado, 1990-
Predição de tráfego, usando redes neurais artificiais, para
gerenciamento adaptativo de largura de banda em roteadores / Tiago
Prado Oliveira. - 2014.
211 f. : il.

Orientador: Jamil Salem Barbar.
Coorientador: Alexsandro Santos Soares.
Dissertação (mestrado) - Universidade Federal de Uberlândia,
Programa de Pós-Graduação em Ciência da Computação.
Inclui bibliografia.

1. Computação - Teses. 2. Redes de computadores - Teses. 3. Redes
neurais (Computação) - Teses. 4. Algoritmos - Teses. I. Barbar, Jamil
Salem, 1962- II. Soares, Alexsandro Santos. III. Universidade Federal de
Uberlândia. Programa de Pós-Graduação em Ciência da Computação.
IV. Título.

CDU: 681.3

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da dissertação intitulada “**Predição de Tráfego, Usando Redes Neurais Artificiais, para Gerenciamento Adaptativo de Largura de Banda em Roteadores**” por **Tiago Prado Oliveira** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 19 de Dezembro de 2014

Orientador:

Prof. Dr. Jamil Salem Barbar
Universidade Federal de Uberlândia

Banca Examinadora:

Prof. Dr. Flávio de Oliveira Silva
Universidade Federal de Uberlândia

Prof. Dr. Mehran Misaghi
Sociedade Educacional de Santa Catarina

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Data: 19 de Dezembro de 2014

Autor: **Tiago Prado Oliveira**
Título: **Predição de Tráfego, Usando Redes Neurais Artificiais, para Gerenciamento Adaptativo de Largura de Banda em Roteadores**
Faculdade: **Faculdade de Computação**
Grau: **Mestrado**

Fica garantido à Universidade Federal de Uberlândia o direito de circulação e impressão de cópias deste documento para propósitos exclusivamente acadêmicos, desde que o autor seja devidamente informado.

Autor

O AUTOR RESERVA PARA SI QUALQUER OUTRO DIREITO DE PUBLICAÇÃO DESTE DOCUMENTO, NÃO PODENDO O MESMO SER IMPRESSO OU REPRODUZIDO, SEJA NA TOTALIDADE OU EM PARTES, SEM A PERMISSÃO ESCRITA DO AUTOR.

Agradecimentos

A Deus, em primeiro lugar, pela oportunidade de superar mais uma etapa e por estar ao meu lado durante todos os momentos.

Aos meus pais, Antônio Nunes de Oliveira e Célia Maria Prado Oliveira, e irmão, Felipe Prado Oliveira, pelo apoio incondicional. Sem vocês esse trabalho não seria realizado.

A toda a minha família, pelos ensinamentos, momentos de convivência e felicidade, que me moldaram na pessoa que sou hoje.

Ao meu orientador, Prof. Dr. Jamil Salem Barbar, não só pelos conhecimentos técnicos, mas também pela experiência compartilhada, que foram decisivos na conclusão deste trabalho.

Ao Prof. Dr. Alexsandro Santos Soares, pela ajuda na co-orientação, pelos conhecimentos técnicos compartilhados, pelos desafios proporcionados que foram essenciais para meu crescimento e pela ajuda e dedicação que foram fundamentais na conclusão deste trabalho.

Ao Sr. Erisvaldo Araújo Fialho, pela competência e prestatividade em ajudar todos pertencentes ao Programa de Pós-Graduação em Ciência da Computação da Faculdade de Computação.

Ao Prof. Dr. Rafael Pasquini, pela ajuda no laboratório de pesquisa e fornecimento dos equipamentos necessários para realizar este trabalho.

Aos amigos do mestrado e graduação, pela convivência, aprendizado e ajuda durante os momentos de dificuldade.

À Universidade Federal de Uberlândia, pela infraestrutura fornecida.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), pelo apoio financeiro.

Resumo

Desde a criação das redes de computadores, é presenciado um aumento de seu tamanho e complexidade. Além disso, também ocorre um aumento na taxa de transmissão de dados nos enlaces de comunicação, podendo trazer problemas de indisponibilidade, decorrentes de congestionamentos. Esses problemas podem ser tratados com métodos de alocação de recursos e controle de congestionamento, que são problemas complexos e bastante estudados. Portanto, para tratar esses problemas e aprimorar a Qualidade de Serviço (QoS), este trabalho apresenta um algoritmo de gerenciamento de redes de computadores, que aloca a largura de banda de forma adaptativa, baseada na previsão do tráfego de rede. Esse algoritmo foi chamado de Gerenciamento Preditivo de Largura de Banda usando Redes Neurais (GPLNEURO). O propósito é alocar a largura de banda, das interfaces de conexão de um roteador, de modo justo, onde cada interface recebe somente o recurso necessário para o tráfego previsto, ou seja, a largura de banda se adapta para receber o tráfego futuro em cada interface do roteador, de forma independente. O algoritmo GPLaB proposto, usa o protocolo SNMP para o monitoramento da rede de computadores, que coleta as séries temporais do tráfego que será previsto. A disciplina Hierarchical Token Bucket (HTB) é usada para controlar a largura de banda de cada interface de conexão do roteador. A largura de banda é alocada de acordo com o tráfego previsto. Os métodos usados para a predição do tráfego foram baseados nas Redes Neurais Artificiais (RNAs). Foram comparadas RNAs tradicionais com RNAs com aprendizagem profunda (RNAP), cuja popularidade aumentou bastante nos últimos anos. A popularidade das novas redes neurais com aprendizagem profundas vêm aumentando em muitas áreas, porém há uma falta de estudos em relação à predição de séries temporais, como o tráfego de Internet. As principais contribuições deste trabalho são o (i) estudo comparativo da predição de tráfego, usando RNAs e RNAP, e o (ii) algoritmo de gerenciamento adaptativo de largura de banda, que pode ser usado como ferramenta auxiliar no gerenciamento de desempenho de redes de computadores.

Palavras chave: Aprendizagem Profunda, Gerenciamento Adaptativo, Largura de Banda, Predição, Redes de Computadores, Redes Neurais Artificiais, Series Temporais.

Abstract

Since the formation of computer networks, a considerable increase in its complexity and size has been seen. Furthermore, there is also an increase in the rate of data transmission across communication links. This may cause problems of unavailability due to network congestion. The resource allocation and congestion control methods can be used to handle network congestion, however, they are complex issues and have been the subject of much study. Therefore, to attend these issues and improve Quality of Service (QoS), this work presents an algorithm for a network management system, which allocates adaptively the router bandwidth, based on the network traffic prediction. This algorithm is called Bandwidth Predictive Management with Neural Networks (GPLNEURO). The bandwidth allocation occurs in the router interfaces and focuses on fairness allocation, each interface receives only the necessary resource, accordingly to the predicted traffic, i.e., the bandwidth adapts to receive the future traffic in each interface. The proposed GPLaB algorithm uses the SNMP to monitor the network traffic data, collecting the time series to predict the future traffic. The bandwidth of the router interfaces is controlled through the Hierarchical Token Bucket (HTB) queueing discipline and depends of the predicted traffic. The network traffic prediction methods, used in this paper, were based on Artificial Neural Networks. Traditional neural networks were compared with deep learning neural networks, which popularity has greatly increased in recent years. The popularity of the new deep learning neural networks has increased in several areas, but there is a lack of studies regarding time series prediction, such as Internet traffic. The two main contributions of this work is a (i) comparative study of traditional neural networks and deep learning ones for Internet traffic prediction and the (ii) adaptive bandwidth management algorithm to be used as an aid for computer network performance management.

Keywords: Adaptive Management, Artificial Neural Networks, Bandwidth, Computer Networks, Deep Learning, Prediction, Time Series.

Sumário

Lista de Figuras	xvii
Lista de Tabelas	xxi
Lista de Abreviaturas e Siglas	xxiii
1 Introdução	27
2 Predição de Séries Temporais	35
2.1 Definição de Predição	36
2.2 Séries Temporais Para Predição	36
2.2.1 Composição das Séries Temporais	39
2.2.1.1 Tendência em Séries Temporais	39
2.2.1.2 Sazonalidade em Séries Temporais	40
2.2.1.3 Ciclo em Séries Temporais	41
2.2.1.4 Residual em Séries Temporais	43
2.3 Análise de Séries Temporais	43
2.3.1 Tipos de Métodos de Predição	44
2.3.2 Estacionariedade e Diferenciação	46
2.3.3 Autocorrelação	48
2.3.4 Variáveis Preditoras e Modelos Básicos de Predição de Séries Tem- porais	49
2.4 Predição de Séries Temporais	51
2.4.1 Diferentes Horizontes de Predição	52
2.4.2 Diagnóstico dos Erros Residuais da Predição	52
2.5 Modelos de Predição	53
2.5.1 Métodos Simples de Predição	54
2.5.2 Suavização Exponencial e Holt-Winters	54

2.5.3	Modelagem ARIMA	56
2.5.3.1	Modelo Autorregressivo	56
2.5.3.2	Modelo de Média Móvel	57
2.5.3.3	Modelo Autorregressivo de Média Móvel	57
2.5.3.4	Modelo Autorregressivo Integrado de Média Móvel	58
2.6	Redes Neurais Artificiais	59
2.6.1	Arquiteturas e Topologias das Redes Neurais Artificiais	63
2.6.1.1	Redes Neurais Diretas de Camada Simples	63
2.6.1.2	Redes Neurais Diretas de Múltiplas Camadas	64
2.6.1.3	Redes Neurais Recorrentes	65
2.6.1.4	Redes Neurais Reticuladas	66
2.6.2	Processos de Aprendizagem de Redes Neurais Artificiais	67
2.6.2.1	Aprendizado Supervisionado	67
2.6.2.2	Aprendizado Não Supervisionado	69
2.6.2.3	Aprendizado por Reforço	69
2.6.3	Redes Neurais Artificiais com Aprendizagem Profunda	70
2.6.3.1	Algoritmos de Aprendizagem Profunda	71
2.6.4	Modelos de Redes Neurais Artificiais	74
2.6.4.1	<i>Multilayer Perceptron</i> e Treinamento <i>Backpropagation</i>	74
2.6.4.2	<i>Jordan Neural Network</i>	76
2.6.4.3	<i>Stacked Autoencoder</i> e Aprendizagem Profunda	76
2.7	Avaliação da Precisão da Predição	77
3	Gerenciamento de Redes de Computadores	79
3.1	Áreas do Gerenciamento de Redes	80
3.1.1	Gerenciamento de Falta	80
3.1.2	Gerenciamento de Contabilidade	81
3.1.3	Gerenciamento de Configuração	81
3.1.4	Gerenciamento de Segurança	81
3.1.5	Gerenciamento de Desempenho	82
3.2	Sistemas de Gerenciamento de Redes	83
3.3	Congestionamento nas Redes de Computadores	84
3.3.1	Controle de Congestionamento e Alocação de Recursos	85
3.3.2	Congestionamento em Roteadores	88
3.3.3	Critérios Básicos para o Controle de Congestionamento	89
3.4	Protocolo SNMP	91
3.4.1	Estrutura da MIB	94
3.5	Controle da Largura de Banda	98

4	Redes Neurais para a Predição de Tráfego	101
4.1	Predição com Redes Neurais	102
4.2	Descrição dos Experimentos Realizados	104
4.2.1	Análise Prévia do Tráfego de Internet	107
4.2.2	Normalização e Conjunto de Treinamento	108
4.2.3	Execução dos Experimentos	110
4.2.3.1	Experimento Exaustivo 1 — JNN	111
4.2.3.2	Experimento Exaustivo 2 — MLP	111
4.2.3.3	Experimento Exaustivo 3 — SAE	112
4.2.4	Apresentação dos Resultados dos Experimentos	113
4.2.4.1	Treinamento das Redes Neurais para a Predição	113
4.2.4.2	Arquiteturas e Topologias de cada Rede Neural	115
4.2.5	Análise dos Resultados dos Experimentos de Predição	118
4.3	Conclusão do Capítulo	120
5	Algoritmo de Gerenciamento Adaptativo de Largura de Banda em Roteadores	123
5.1	Arquitetura do GPLNEURO	124
5.2	Algoritmo GPLNEURO	126
5.2.1	Módulo Monitoramento	126
5.2.2	Módulo Treinamento	130
5.2.3	Módulo Predição	131
5.2.4	Módulo Análise	132
5.2.5	Módulo Controle	135
5.3	Experimento com o Algoritmo GPLNEURO	136
5.3.1	Ambiente de Testes	136
5.3.2	Execução do Experimento de Gerenciamento	136
5.3.3	Análise dos Resultados do Experimento de Gerenciamento	138
5.4	Conclusão do Capítulo	140
6	Considerações Finais	143
	Referências	147
A	Códigos Fonte dos Experimentos	153
A.1	Códigos para os Experimentos de Predição de Tráfego	153
A.2	Códigos do Algoritmo GPLNEURO	184
B	Configuração dos Equipamentos de <i>Hardware</i>	205
B.1	Configuração do Roteador Implementado em um Servidor	206
B.2	Configuração dos Computadores Conectados ao Roteador	206

B.3	Configuração do <i>Switch</i> TRENDnet Usado para Criar as VLANs	207
-----	--	-----

Lista de Figuras

2.1	Exemplo de uma série temporal discreta	38
2.2	Exemplo de uma série temporal contínua	38
2.3	Exemplo de uma série temporal com uma tendência crescente	40
2.4	Exemplo de série temporal com alta sazonalidade	41
2.5	Exemplo de sazonalidade	41
2.6	Exemplo de série temporal com ciclos	42
2.7	Série temporal estacionária	47
2.8	Série Temporal Dow Jones e suas Diferenciações	48
2.9	Exemplo da Autocorrelação e Autocorrelação da Primeira Diferenciação . .	49
2.10	Observações de uma série temporal qualquer, com previsões de origem t e horizonte h	52
2.11	Exemplo de métodos simples de previsões	55
2.12	Analogia entre neurônios biológicos e artificiais	61
2.13	Diagrama em blocos do sistema nervoso humano e neurônios artificiais . .	61
2.14	Tipos de função de ativação	62
2.15	Exemplo da arquitetura de uma Rede Direta de Camada Simples	64
2.16	Exemplo da arquitetura de uma Rede Direta de Múltiplas Camadas	65
2.17	Exemplos de arquitetura de Redes Neurais Recorrentes	66
2.18	Exemplos de Redes de Kohonem	66
2.19	Diagrama representado o funcionamento do Aprendizado Supervisionado .	68
2.20	Diagrama representando a Aprendizagem Não Supervisionada	69
2.21	Diagrama representando o Aprendizado por Reforço	70
2.22	Exemplo dos níveis de representação no reconhecimento de rostos humanos	72
2.23	Arquitetura de uma DBN com suas camadas de entrada, ocultas e saída . .	73
2.24	Arquitetura de uma RBM	73
3.1	Exemplo de um gargalo que pode gerar congestionamento	85
3.2	Ilustração gráfica do congestionamento	86

3.3	Arquitetura de um roteador	88
3.4	Razão da vazão pelo atraso	90
3.5	Atraso em função da carga	91
3.6	Configuração típica dos protocolos para o SNMP	93
3.7	Papel do SNMP, agente e gerente	94
3.8	Sequências das mensagens de comunicação SNMP	95
3.9	Árvore de objetos SMI	96
3.10	Disciplina de escalonamento HTB.	100
4.1	Exemplo de uma janela deslizante de previsão de uma série temporal . . .	104
4.2	Séries temporais com o tráfego (em bits) utilizadas nos experimentos . . .	106
4.3	Autocorrelação das séries temporais dos experimentos	109
4.4	Série temporal original dividida em duas partes: conjunto de treinamento e conjunto de testes.	110
4.5	Comparação do MSE obtido pelo SAE e pela MLP-BP, durante os 50 pri- meiros períodos de treinamento, para a série temporal B-5m.	114
4.6	Comparação do MSE obtido pela JNN e pela MLP-RP, durante os 50 primeiros períodos de treinamento, para a série temporal B-5m.	115
4.7	Comparação da série temporal Original e a série temporal Predita usando a JNN com uma camada oculta, para a série B-5m	116
4.8	Comparação da predição do SAE com 4 camadas ocultas, para a série temporal B-5m, apresentando a série temporal Original e a série Predita .	116
4.9	Arquiteturas das redes neurais utilizadas nos experimentos	117
4.10	Proporção entre a complexidade da rede neural e os MSEs da predição da série temporal B-5m	118
5.1	Exemplo da conexão do roteador entre a Internet e as sub-redes.	124
5.2	Arquitetura do Roteador usando o GPLNEURO.	125
5.3	Diagrama de atividades do algoritmo GPLNEURO.	127
5.4	Fluxograma do módulo Monitoramento	128
5.5	Subárvores <code>ifInOctets</code> e <code>ifOutOctets</code> do gerenciamento das interfaces da MIB	129
5.6	Fluxograma do módulo Treinamento	130
5.7	Fluxograma do módulo Predição	132
5.8	Fluxograma do módulo Análise	133
5.9	Fluxograma do módulo Controle	135
5.10	Roteador onde foi implementado o GPLNEURO para os experimentos re- ferentes ao gerenciamento da largura de banda.	137
5.11	Série temporal do tráfego gerado na interface <code>eth3</code> do roteador, em um período de 1020 minutos (17 horas).	138

5.12	Comparação entre o tráfego gerado original e o tráfego predito pela JNN da interface <i>eth3</i> do roteador.	139
5.13	Tempo médio de atraso (ms) para cada minuto do tráfego passado na interface de rede <i>eth3</i>	139

Lista de Tabelas

4.1	O intervalo de tempo e o tamanho de cada série temporal.	110
4.2	Comparação da média do tempo de treinamento.	114
4.3	Comparação dos erros normalizados (NRMSE) obtidos.	118
4.4	Ranking das redes neurais mais rápidas para cada série temporal.	119
4.5	Ranking das redes neurais com os menores NRMSEs para cada série temporal.	120
5.1	Atraso (ms) médio e desvio padrão das interfaces do roteador relativos ao uso do algoritmo GPLNEURO.	140
B.1	Configuração do Roteador.	206
B.2	Configuração do Computador 1.	206
B.3	Configuração do Computador 2.	207
B.4	Configuração do Computador 3.	207
B.5	Switch Status	207
B.6	PORT Status 10/100/1000 Mbps	208
B.7	Port-based VLAN Settings	208
B.8	TRUNK Status	209
B.9	Mirror Status	209

Lista de Abreviaturas e Siglas

ACF	<i>Autocorrelation Function</i>
AIC	<i>Aikaike's Information Criterion</i>
ANS.1	<i>Abstract Sintaxe Notation One</i>
AR	<i>Autoregressive</i>
ARMA	<i>Autoregressive Moving Average</i>
ARIMA	<i>Autoregressive Integrated Moving Average</i>
CBQ	<i>Class Based Queuing</i>
CBR	<i>Constant Bit Rate</i>
DBN	<i>Deep Belief Networks</i>
EGP	<i>Exterior Gateway Protocol</i>
ES	<i>End System</i>
FARIMA	<i>Fractionally Autoregressive Integrated Moving Average</i>
FDDI	<i>Fiber Distributed Data Interface</i>
GPLNEURO	<i>Gerenciamento Preditivo de Largura de Banda usando Redes Neurais</i>
HTB	<i>Hierarchical Token Bucket</i>
HTTP	<i>HyperText Transfer Protocol</i>
HW	<i>Holt-Winters — Suavização Exponencial Sazonal de Holt-Winters</i>
IAB	<i>Internet Activities Board</i>
ICMP	<i>Internet Control Message Protocol</i>
IETF	<i>Internet Engineering Task Force</i>
IP	<i>Internet Protocol</i>
IPv4	<i>Internet Protocol version 4</i>
IS	<i>Intermediate System</i>
ISO	<i>International Organization for Standardization</i>
ISP	<i>Internet Service Provider</i>
JNN	<i>Jordan Neural Network</i>
LAN	<i>Local Area Network</i>

LDR	<i>Long Range Dependence</i>
MA	<i>Moving Average</i>
MAPE	<i>Mean Absolute Percentage Error</i>
MIB	<i>Management Information Base</i>
MLP	<i>Multilayer Perceptron</i>
MLP-BP	<i>Multilayer Perceptron com Backpropagation</i>
MLP-RP	<i>Multilayer Perceptron com Resilient Backpropagation</i>
MSE	<i>Mean Squared Error</i>
NRMSE	<i>Normalized Root Mean Square Error</i>
OID	<i>Object Identifier</i>
OSI	<i>Open Systems Interconnection</i>
PRR	<i>Packet Round Robin</i>
QoS	<i>Quality of Service</i>
RBF	<i>Radial Basis Function</i>
RBM	<i>Restricted Boltzmann Machine</i>
REG	<i>Random Early Detection</i>
RFC	<i>Request for Comments</i>
RMSE	<i>Root Mean Square Error</i>
RMON	<i>Remote Network Monitoring</i>
RNA	Rede Neural Artificial
RNAP	Redes Neurais com Aprendizagem Profunda
RNN	<i>Recurrent Neural Network</i>
RPROP	<i>Resilient Backpropagation</i>
SAE	<i>Stacked Autoencoder</i>
SEH	Suavização Exponencial de Holt
SES	Suavização Exponencial Simples
SGALB	Sistema de Gerenciamento Adaptativo de Largura de Banda
SMI	<i>Structure of Management Information</i>
SNMP	<i>Simple Network Management Protocol</i>
SNMPv2	<i>Simple Network Management Protocol version 2</i>
SNMPv3	<i>Simple Network Management Protocol version 3</i>
SRD	<i>Short Range Dependence</i>
SSE	<i>Sum Squared Error</i>
SVM	<i>Support Vector Machine</i>
tanh	Função tangente hiperbólica
TCP	<i>Transmission Control Protocol</i>
UDP	<i>User Datagram Protocol</i>
VBR	<i>Variable Bit Rate</i>
VOQ	<i>Virtual Output Queue</i>

VLAN	<i>Virtual Local Area Network</i>
WAN	<i>Wide Area Network</i>
WRR	<i>Weighted Round Robin</i>

Introdução

Desde a criação das redes de computadores, é notório um aumento do tamanho dessas redes e de sua complexidade. Além disso, presencia-se um aumento considerável na taxa de transmissão de dados dos enlaces de comunicação. Esse aumento ocorre, em parte, devido à utilização de meios físicos mais apropriados e ao uso de técnicas mais eficientes de transmissão, codificação e empacotamento. Ao mesmo tempo, esse crescimento foi acompanhado por uma redução no custo de transmissão e pela popularização dessas redes, tornando os enlaces de alta velocidade mais acessíveis (Comer, 2007; Haykin e Moher, 2009; Kurose e Ross, 2012; Peterson e Davie, 2011).

As redes de computadores foram criadas para substituir o modelo antigo de usar somente um computador para suprir as necessidades de uma organização. Uma rede de computadores é composta por computadores autônomos, porém interconectados. Computadores interconectados são aqueles capazes de trocar informações entre si. As redes de computadores, comumente também chamadas somente de redes (não confundir com as redes neurais, que serão apresentadas mais a frente), podem possuir diferentes tamanhos e formas. Essas redes usualmente são conectadas com outras redes, formando uma rede de computadores ainda maior. A Internet é o exemplo mais conhecido de uma rede de redes (Kurose e Ross, 2012; Tanenbaum e Wetherall, 2010).

Além dos computadores, uma rede também é constituída por outros equipamentos, como por exemplo: roteadores, comutadores (*switches*), concentradores (*hubs*), impressoras, digitalizadores e vários outros dispositivos. A conexão desses equipamentos pode ser feita fisicamente por meio de cabos ou por ar, usando cabos coaxiais, fio de cobre, fio de alumínio, cabos de fibra óptica, ondas eletromagnéticas, micro-ondas, infravermelho, ou até mesmo satélites de comunicação (Haykin e Moher, 2009; Tanenbaum e Wetherall, 2010). O roteador é um dos mais importantes elementos na interconexão de dispositivos

em rede. A principal função do roteador é o encaminhamento, também conhecido pelo termo “roteamento”, que é a tarefa na qual ocorre a transferência de pacotes a partir de uma interface de entrada para uma das interfaces de saída do roteador (Peterson e Davie, 2011; Tanenbaum e Wetherall, 2010). Os dados trafegados em uma rede de computadores, portanto, são transferidos de um roteador para um outro componente de rede, até chegar em seu destino final.

A largura de banda (*bandwidth*) e o atraso (*delay*) são fatores que limitam a velocidade em que os dados são passados através de um roteador. Normalmente, a largura de banda de um enlace de comunicação refere-se ao número de bits por segundo que podem ser transmitidos nesse enlace, ou seja, é a capacidade total disponível para transmissão. Já a vazão (*throughput*), é uma medida de desempenho, este relativo ao número de bits por segundo que realmente é transmitido. Essa diferença da capacidade total para transmissão e o número de bits que de fato é transmitido ocorre, principalmente, por causa de implementações ineficientes dos enlaces de comunicação, tanto em *hardware* quanto em *software* (Peterson e Davie, 2011).

Mesmo com a alta capacidade de transmissão das redes de computadores atuais, o controle do compartilhamento da largura de banda das interfaces do roteador ainda é uma questão importante, pois essa capacidade de transmissão ainda é finita. O grande crescimento das redes e de sistemas de processamento distribuído trouxe uma necessidade maior do gerenciamento de redes de computadores. O gerenciamento de redes se torna crítico para as grandes organizações, empresariais ou não, nas quais necessitam de redes maiores, mais complexas e que suportem mais aplicações e usuários. Devido a esse aumento na escalabilidade e complexidade, os problemas podem ser mais comuns, trazendo sérios problemas de desempenho ou incapacitando a rede, ou parte dela (Kurose e Ross, 2012).

O fluxo de dados em uma rede de computadores geralmente não é constante. O ideal seria levar em consideração esse fluxo variável, fazendo com que o compartilhamento da largura de banda dos elos de comunicação seja adaptativo (Comer, 2007). Os elementos de redes padrões utilizam uma alocação de largura de banda estática, desse modo pode haver a subutilização ou superutilização de uma interface de saída do roteador. Assim, há uma limitação da largura de banda na rede de computadores, podendo levar a escassez desse recurso. Esse problema não permite que se garanta qualidade de serviço, ou *Quality of Service* (QoS), em aplicações sensíveis a largura de banda (Kurose e Ross, 2012), que é a motivação principal do trabalho proposto. Então, torna-se necessário um sistema de gerenciamento adaptativo, que atue em um roteador e realize o controle da largura de banda em uma rede de computadores, compartilhando-a de maneira justa e atendendo aos critérios pré-determinados.

O objetivo do gerenciamento adaptativo da largura de banda de um roteador é proporcionar uma distribuição apropriada desse recurso, entre as interfaces dos roteadores,

doravante denominadas de somente interfaces. Isso é feito tanto por questões administrativas quanto para prover QoS para aplicações, serviços e protocolos. Uma solução para a realização desse gerenciamento é a utilização da predição do tráfego da rede de computadores, que identifica as demandas futuras do tráfego nas interfaces para então, alterar a largura de banda (aumentando ou diminuindo), de acordo com o necessário. O aumento ou diminuição do limite da largura de banda, em cada interface, terá como base as informações anteriores ao momento atual sobre o tráfego da rede de computadores.

O congestionamento dos pacotes ocorre quando a vazão de dados nos elementos das redes de computadores é superior à capacidade de transmissão desses dados nesses elementos. Como os elementos de comunicação são compartilhados entre diversos computadores, cada computador acaba utilizando somente uma parte das interfaces de saída de um roteador, por exemplo. A alta transmissão de dados de diferentes computadores, em um curto período de tempo, para um roteador, acaba gerando um atraso na transmissão dos pacotes e, caso essa alta vazão persista, ocasionará o congestionamento. O congestionamento fará com que ocorra perda de pacotes, que implicará em retransmissões dos pacotes, acarretando um aumento do tráfego e atrasos na entrega dos dados, afetando negativamente aos parâmetros de QoS (Kurose e Ross, 2012; Tanenbaum e Wetherall, 2010). O controle de congestionamento e alocação de recursos são problemas complexos e são objetos de estudo desde a criação das redes de computadores (Peterson e Davie, 2011).

A utilização de predição de tráfego em curto prazo é uma forma para efetuar o controle dinâmico da alocação de largura de banda para cada interface do roteador, já que as características estatísticas do tráfego agregado podem ser bastante imprevisíveis a longo prazo (Siripongwutikorn *et al.*, 2003). A predição de tráfego se baseia na análise das características passadas do tráfego, identificando correlações como tendências e períodos sazonais. Para isso, são necessários registros do histórico do tráfego de dados passados, isto é, uma série temporal do tráfego que será usado na predição. Além do tráfego passado, também são necessários métodos matemáticos de previsão baseados em séries temporais, esses métodos de previsão podem usar tanto cálculos simples quanto complexos (Morettin e de Castro Toloi, 2006).

O algoritmo de gerenciamento adaptativo proposto neste trabalho tem o foco no gerenciamento de desempenho em roteadores, atuando no controle da largura de banda com a finalidade de ajudar a evitar congestionamentos. Para esse tipo de gerenciamento de desempenho são necessárias, basicamente, duas etapas, uma para monitoramento e outra para controle. O monitoramento faz a coleta de dados do tráfego transmitido em um enlace de comunicação. O controle, portanto, deve tomar a decisão de alterar ou manter a largura de banda das interfaces. A largura de banda não será alocada de acordo com as proporções dos dados em cada interface, de acordo com o tráfego futuro previsto.

As interfaces que necessitarem mais tráfego, receberão um limite de largura de banda maior e mais justo. Primeiramente, o sistema de gerenciamento proposto será centrali-

zado. A etapa de monitoramento usará o protocolo *Simple Network Management Protocol* (SNMP) para coletar o tráfego das interfaces de rede. O SNMP foi escolhido devido sua portabilidade e escalabilidade, para que em trabalhos futuros o mesmo sistema possa ser usado em um sistema de gerenciamento distribuído (Stallings, 1999). O controle da largura de banda será efetuado por meio da disciplina de escalonamento de filas *Hierarchical Token Bucket* (HTB), que permite a mudança do limite da largura de banda para cada interface do roteador (Antony *et al.*, 2012).

Uma série temporal é um conjunto de dados observados sequencialmente na linha do tempo. Séries temporais são úteis para prever algo que está mudando com o passar do tempo. Existem diversas técnicas, métodos e modelos para a análise e predição de séries temporais, essas podem ser técnicas estatísticas mais simples ou modelos híbridos avançados.

As técnicas de predição mais simples são as regressões lineares, decomposição de séries temporais e suavização exponencial. Os métodos mais complexos são métodos Bayesianos, transformadas Wavelets, modelos que combinam autorregressão e médias móveis e modelos de aprendizado de máquina (*Support Vector Machine* — SVM, *Restricted Boltzmann Machine* — RBM e Rede Neural Artificial — RNA) (Hornik e Leisch, 2000; Hyndman e Athanasopoulos, 2013). Cada técnica possui suas vantagens e desvantagens, que estão relacionadas à precisão da predição, complexidade de implementação, complexidade computacional, tempo de predição e outros.

Na literatura existem estudos sobre a predição de tráfego em uma rede de computadores. O uso de redes neurais para a predição do tráfego é muito estudado (Bolshinsky e Freidman, 2012; Cortez *et al.*, 2006, 2008, 2012; Crone *et al.*, 2011; Pandey *et al.*, 2013), e mostra-se competitivo em comparação com os métodos mais tradicionais como Holt-Winters (Hyndman e Athanasopoulos, 2013; Kalekar, 2004), ARIMA (Hyndman e Athanasopoulos, 2013) e Transformada Wavelet (Feng e Shu, 2005; Wang *et al.*, 1998; Zhang *et al.*, 2012). Porém, há poucos estudos que comparam o uso de redes neurais tradicionais com novas abordagens, como o aprendizado profundo.

As redes neurais são candidatas naturais para predição devido a sua capacidade de tolerar ruídos e sua não linearidade. O uso de redes neurais para a predição do tráfego começou no final da década de 80 com resultados encorajadores, e esse campo de pesquisa tem aumentado bastante desde então (Cortez *et al.*, 2006; Hosseini *et al.*, 2012; Pandey *et al.*, 2013). O principal diferencial desse trabalho será o uso de uma nova técnica de aprendizado de máquina, chamada de *Deep Learning*, ou Aprendizagem Profunda (Bengio, 2009), para o aprendizado de tráfegos de redes.

Redes Neurais Artificiais (RNA), ou somente redes neurais, são modelos matemáticos inspirados na estrutura do cérebro humano. Uma RNA é composta por neurônios, que são estruturas de processamento simples, separados em unidades fortemente conectadas. Os neurônios de uma RNA são organizados em camadas, uma camada pode ter vários

neurônios e uma rede neural pode ter duas ou mais camadas (pelo menos uma camada de entrada e uma de saída). Os neurônios são capazes de trabalhar em paralelo para processar dados, armazenar conhecimento e usar esse aprendizado para inferir novos dados.

O conhecimento da RNA é armazenado pelo peso sináptico, que é a força de conexão entre os neurônios de uma RNA. O conhecimento da rede é adquirido através do processo de aprendizagem, também conhecido como algoritmo de aprendizagem ou algoritmo de treinamento (Haykin, 1998). Após o aprendizado, a RNA está pronta para reconhecer os padrões da série temporal, por exemplo. As redes neurais oferecem as mesmas funcionalidades dos neurônios do cérebro humano para resolver problemas complexos, tais como a não linearidade, alto paralelismo, robustez, tolerância a erros e ruídos, adaptabilidade, aprendizado e generalização (Cortez *et al.*, 2012; Haykin, 1998).

Historicamente, o uso de redes neurais era limitado em relação ao número de camadas ocultas (escondidas), devido a dificuldade de treinar redes neurais com muitas camadas e neurônios (Bengio, 2009). Algoritmos de treinamento convencionais, como o *Backpropagation*, não tem um bom desempenho em RNAs profundas, com mais de três camadas ocultas (Erhan *et al.*, 2009). Porém, em 2006, Hinton apresentou redes neurais com aprendizagem profunda para o problema de reconhecimento de imagens, com um algoritmo de treinamento eficiente, baseado em um algoritmo guloso de aprendizagem que treina uma camada de cada vez (Hinton *et al.*, 2006). Ele chamou essas redes neurais de *Deep Belief Networks* (DBN).

A partir da DBN proposta por Hinton *et al.* (2006), pesquisas encontraram vários bons resultados, relativos ao uso de Redes Neurais com Aprendizagem Profunda (RNAP), que usam dados não rotulados e algoritmo de treinamento guloso não supervisionado, atingindo erros menores do que as redes neurais mais comuns. Graças ao treinamento guloso a DBN possui um tempo de treinamento mais rápido do que RNAs tradicionais. Logo, outro objetivo deste trabalho é analisar diferentes tipos de redes neurais e encontrar qual o melhor modelo para a predição de séries temporais, considerando o erro médio e o tempo médio de treinamento e predição de cada modelo.

As redes com aprendizagem profunda referem-se a um método de aprendizado de máquina baseado em modelos de redes neurais com vários níveis de representação de dados, permitindo aprender dados mais complexos. Os níveis hierárquicos de representação de dados são organizados por abstrações, características e conceitos. Níveis mais altos são definidos pelos níveis mais baixos, onde a representação dos níveis mais baixos podem definir várias características diferentes dos níveis mais altos (Bengio, 2009; Hinton *et al.*, 2006).

Em uma rede neural com aprendizagem profunda, quanto mais alto o nível, mais abstrato e não linear será a representação dos dados. Esses níveis hierárquicos são representados pelas camadas da RNA. Desse modo, a profundidade da rede neural se diz respeito ao número de níveis de composição das operações não lineares perante os dados

treinados, i.e., quanto mais camadas, mais complexa, não linear e profunda será a RNA.

Este trabalho analisa quatro modelos de predição baseados em redes neurais artificiais. Avaliações foram feitas comparando *Multilayer Perceptron* (MLP) ou Perceptrons de Múltiplas Camadas, Redes Neurais Recorrentes (*Recurrent Neural Networks* — RNN) e *Stacked Autoencoder* (SAE). MLP é uma rede neural direta (*feed-forward neural network*) com múltiplas camadas, que usa um treinamento supervisionado. SAE é uma rede neural profunda (*deep learning neural network*) que usa um algoritmo guloso como treinamento não supervisionado. Para a MLP foram comparados dois algoritmos de treinamento diferentes, o *Backpropagation* padrão e o *Resilient Backpropagation* (RPROP).

Esses métodos de predição foram selecionados com o objetivo de confirmar o quão competitivo as abordagens mais simples (RNN e MLP) são, quando comparadas com as mais complexas (SAE e MLP mais profundas). A análise foca em predições de curto prazo. Os testes foram feitos usando amostras de séries temporais de tráfego de Internet, que foram obtidas na base de dados DataMarket (Hyndman).

O sistema de gerenciamento adaptativo de largura de banda foi implementado em uma rede de computadores que utiliza o protocolo de rede *Internet Protocol version 4* (IPv4), onde não ocorre nenhuma diferenciação de pacotes. A rede de testes foi implementada com LANs que se conectam com a Internet por meio de um roteador, no qual o sistema de gerenciamento proposto será executado. A política do controle da largura de banda será dada de acordo com o tráfego de rede previsto pela RNA. Para o controle foi usado a disciplina de escalonamento HTB e para o monitoramento do tráfego no roteador foi usado o protocolo SNMP.

Esta dissertação tem em sua totalidade seis capítulos, incluindo este capítulo introdutório. A seguir, são apresentados os assuntos abordados nos demais capítulos deste trabalho.

No Capítulo 2, são apresentados os conceitos básicos relacionados à predição de séries temporais. Inicialmente são mostradas as composições e características das séries temporais, seguindo de como deve ser feita sua análise e predição. Em sequência, são detalhados os métodos e modelos de predição, como os modelos estatísticos mais utilizados (ARIMA, modelos Bayesianos e transformadas Wavelets), as redes neurais tradicionais e as redes neurais com aprendizagem profunda.

O Capítulo 3 aborda, inicialmente, os sistemas de gerenciamento de redes de computadores. Sequencialmente, é discutido o congestionamento em redes de computadores, mostrando causas, consequências e as soluções propostas na literatura. Também são apresentados os detalhes do protocolo SNMP, usado no sistema de gerenciamento proposto, e as disciplinas de escalonamento de pacotes em filas, para realizar o controle da largura de banda no roteador.

No Capítulo 4, detalha-se todo o desenvolvimento dos modelos de predição, informando as análises feitas com os diferentes modelos de redes neurais utilizados. Os resultados são

comparados para verificar qual foi o melhor modelo e arquitetura de rede neural para a predição do tráfego de Internet, levando em consideração a precisão da predição, o tempo de treinamento e a complexidade da RNA. Os modelos de rede neurais estudados foram: MLP com *Backpropagation* como algoritmo de treinamento; MLP com o algoritmo *Resilient Backpropagation* para o treinamento; RNN com uma camada de contexto; SAE com aprendizagem profunda.

No Capítulo 5, é descrito a arquitetura do algoritmo de gerenciamento de largura de banda em roteadores proposto. Apresenta-se o uso do sistema de predição usando o modelo de rede neural que se saiu melhor nos experimentos do Capítulo 4. São descritas, então, as etapas referentes ao gerenciamento, que são: monitoramento, treinamento, predição, análise e controle. Essas etapas são detalhadas, mostrando como o gerenciamento foi aplicado, usando a rede neural para a predição de tráfego, as chamadas usadas do protocolo SNMP e o controle da largura de banda feito com o HTB. Por fim, é apresentado os resultados do sistema de gerenciamento implementado em uma rede de computadores de testes.

O Capítulo 6 encerra a dissertação com as considerações finais, informando as conclusões relativas ao trabalho feito. Nele é feito um levantamento da contribuição do trabalho para a comunidade científica, sugestões de melhorias e propostas para trabalhos futuros.

O Apêndice A contém os códigos fontes utilizados para os testes de predição, com as redes neurais, e os códigos fontes do sistema de gerenciamento de redes implementado.

Por fim, o Apêndice B mostra as configurações dos equipamentos utilizados na realização dos testes que utilizaram o algoritmo de gerenciamento adaptativo de largura de banda.

Predição de Séries Temporais

Existem vários modos de se fazer a predição do tráfego de redes. O trabalho proposto nesta monografia usa as séries temporais do tráfego de rede nas interfaces de conexão de um roteador com as LANs, isto é, o conjunto das observações do número de bytes transmitidos nas interfaces de saída de um roteador, coletadas em intervalos iguais de tempo. O uso de séries temporais para a predição é bastante usado, pois elas podem representar múltiplos eventos em contextos variados. Entretanto, a análise de uma série temporal deve ser feita com cautela, levando em consideração o evento que deu origem à ela e a técnica mais adequada para o tratamento e previsão da mesma (Morettin e de Castro Toloi, 2006).

As séries temporais são amplamente utilizadas para a predição. Métodos antigos de predição, como os modelos estatísticos de autorregressão e de médias móveis, ainda funcionam e tem bons resultados (Kalekar, 2004), contudo, os métodos mais novos, como as redes neurais artificiais, modelos integrados e sazonais, se adaptam melhor a vários tipos diferentes de séries. Existem vários estudos recentes sobre predições que usam modelos baseados em séries temporais, como por exemplo: predição de consumo de energia (Busseti *et al.*, 2012; Kouhi e Keynia, 2013); taxa de câmbio (Chao *et al.*, 2011); e, no foco deste trabalho, tráfego de rede de computadores (Bolshinsky e Freidman, 2012; Chao *et al.*, 2011; Cortez *et al.*, 2012; Feng e Shu, 2005; Rutka e Lauks, 2007; Stepnicka *et al.*, 2011; Zhang *et al.*, 2012; Zhani *et al.*, 2008).

2.1 Definição de Predição

Predição é o mesmo que previsão, que é um modo de descobrir algo que existirá no futuro (Morettin e de Castro Toloi, 2006). A predição é útil em várias situações como: prever a demanda futura de energia elétrica para decidir se será construída uma usina elétrica em um local; prever o volume de ligações para agendamento de funcionários em um *call center*; prever as necessidades de estoque para estocar um inventário; ou para o caso deste trabalho, predizer o tráfego futuro de uma rede de computadores para gerenciar os componentes da mesma (Hyndman e Athanasopoulos, 2013).

Predições podem ser necessárias em dois casos. O primeiro é prever um longo período de tempo, por exemplo, no caso de prever a demanda de energia elétrica do próximo mês ou ano. O segundo, que é o caso em que este trabalho se enquadra, é prever um curto período de tempo, por exemplo, prever segundos ou minutos adiante (Hyndman e Athanasopoulos, 2013; Morettin e de Castro Toloi, 2006).

A precisão, ou seja, a taxa de acerto da predição é determinada pelos métodos de predição e pelo nível de entendimento dos fatores que contribuem para o evento. Os métodos podem ser simples, usando a observação mais recente como previsão, conhecido como método ingênuo, ou usando estruturas complexas como redes neurais artificiais e modelos estatísticos de regressão não linear (Hyndman e Athanasopoulos, 2013).

2.2 Séries Temporais Para Predição

Séries temporais são úteis para prever algo que está mudando com o passar do tempo, como por exemplo, preços das ações, números de vendas, lucros. Os métodos para predição dependem dos dados disponíveis das séries temporais. Em relação aos dados, duas coisas são importantes. A primeira, é que os dados devem conter informações numéricas do passado. A segunda, é que exista um certo padrão que é seguido no passar do tempo, ou seja, alguns aspectos do passado continuarão no futuro (Hyndman e Athanasopoulos, 2013). A maioria desses métodos usam dados transversais (*cross-sectional*) ou séries temporais. Os dados transversais são coletados em um único ponto no tempo, já as séries temporais são coletadas em intervalos regulares de tempo (Hyndman e Athanasopoulos, 2013; Morettin e de Castro Toloi, 2006).

Uma série temporal é qualquer conjunto de observações ordenadas no tempo e em intervalos regulares, sendo uma parte de uma trajetória, dentre muitas que poderiam ter sido observadas (Morettin e de Castro Toloi, 2006). Basicamente, tudo que é observado sequencialmente na linha do tempo é uma série temporal. Na previsão de séries temporais, deve-se estimar como a sequência se comportará no futuro (Hyndman e Athanasopoulos, 2013). Além disso, o valor que será previsto é um número aleatório que pertence a um intervalo.

Na maioria das situações de previsão, principalmente em modelos baseados em regressão, a variação associada diminui na medida em que o evento se aproxima em relação ao tempo. Em outras palavras, quanto mais longe no tempo for a previsão, menor a probabilidade de acerto, ou seja, maior a incerteza da precisão do valor previsto (Hyndman e Athanasopoulos, 2013; Morettin e de Castro Tolo, 2006).

As séries temporais podem representar qualquer fenômeno que se deseje estudar, desde que haja valores numéricos para representar esse fenômeno. Desse modo, a escala temporal em que os dados do fenômeno serão coletados pode variar de acordo com o problema, por exemplo:

- (a) Registro de marés de Recife;
- (b) Produção de leite no Estado de São Paulo, composta de 60 dados medidos a cada mês;
- (c) Fluxo de pessoas em um caixa, composto de dados de uma semana, coletados a cada hora;
- (d) Batimentos cardíacos por minuto, de uma pessoa, em uma hora;
- (e) Número de vendas, coletados diariamente durante 100 dias, de uma revista no Brasil;
- (f) Número de carros que passam por um cruzamento a cada 30 minutos, em um dia da semana;
- (g) Tráfego de bits em uma rede de computadores, coletados a cada segundo durante uma hora.

As séries temporais podem ser divididas em dois tipos, séries temporais discretas e séries temporais contínuas. Dos exemplos anteriores, os itens de (b)-(g) são séries temporais discretas (exemplo da Figura 2.1), enquanto o item (a) é uma série temporal contínua (Figura 2.2). A variável observada, ser discreta ou contínua, não influencia no tipo da série temporal, ou seja, uma série temporal contínua pode ser composta de variáveis discretas ou variáveis contínuas. Em outras palavras, para séries temporais discretas, por exemplo, os valores no eixo do tempo é que devem ser discretos, já o eixo variáveis observadas, pode conter variáveis discretas ou contínuas (Chatfield, 2000).

Em muitos casos, uma série discreta é obtida através da amostragem de uma série contínua em intervalos de tempos iguais, por exemplo, amostrar a série contínua em intervalos de tempo de uma hora ou acumular seus valores em períodos de tempo (Chatfield, 2000; Morettin e de Castro Tolo, 2006). O tráfego usado neste trabalho usa séries temporais discretas, composta das observações do número de bytes transmitidos nas interfaces de saída do roteador.

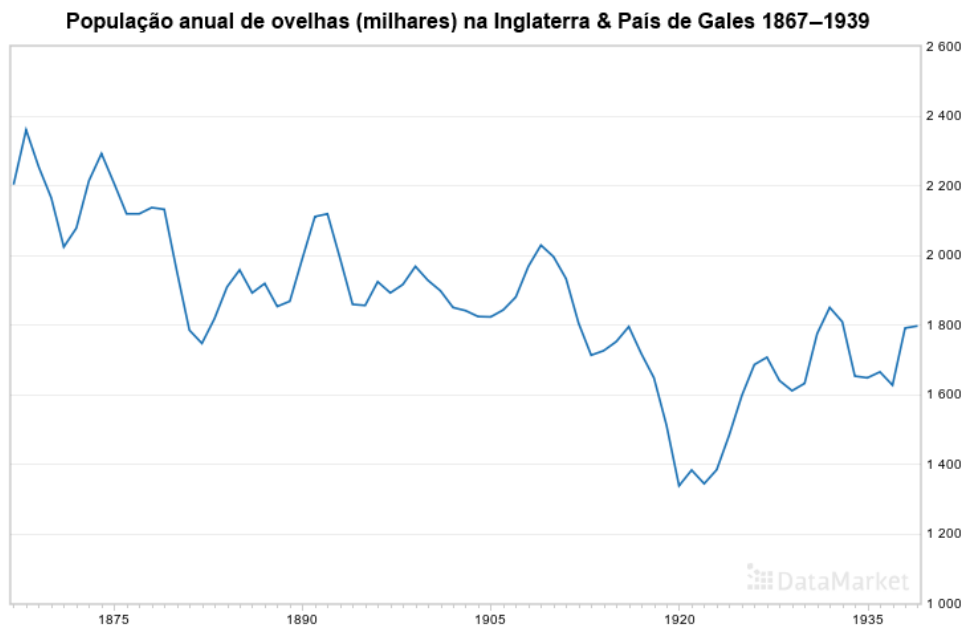


Figura 2.1: Exemplo de uma série temporal discreta, representando a população anual de ovelhas (em milhares, eixo y), na Inglaterra e País de Gales, entre os anos de 1867 e 1939 (eixo x). Fonte: Hyndman, *Time Series Data Library*, datamarket.com.

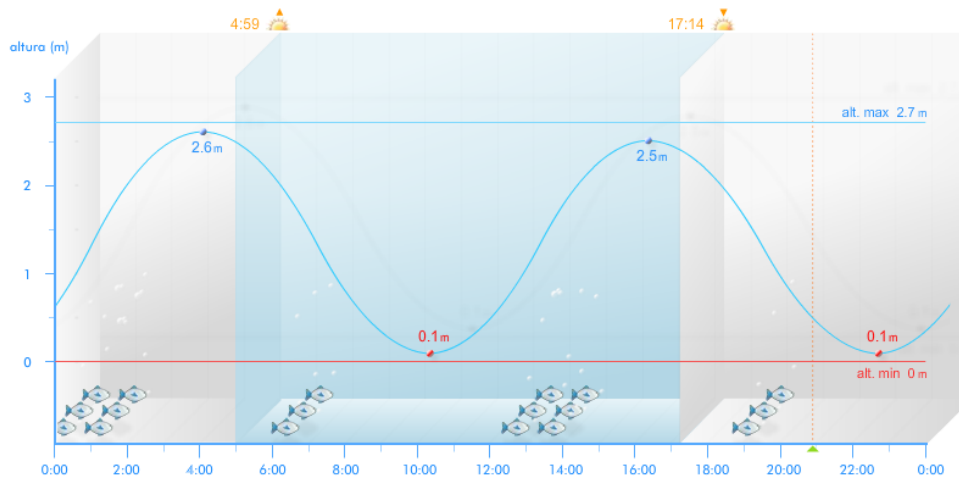


Figura 2.2: Exemplo de uma série temporal contínua, representando o registro de marés de Recife. No eixo y está a altura, em metros, da maré, e no eixo x está o horário em que essa altura foi registrada. Fonte: <http://www.tabuademares.com/br/pernambuco/recife>.

A representação matemática de uma série temporal é simplesmente um vetor de observações, cuja coleta foi feita em intervalos de tempos regulares. A representação básica de uma série temporal é um vetor Z , de ordem $n \times 1$, onde n é o número de observações coletadas. Entretanto, a forma mais comum de representar e armazenar uma série temporal é usando dois vetores, ou uma matriz de ordem $2 \times n$, que é apresentada na Equação (2.1). A primeira linha da matriz guarda os valores numéricos das observações (Z_n) e a

segunda linha guarda o tempo em que a observação foi coletada (T_n).

$$Z = \begin{bmatrix} Z_1, & Z_2, & Z_3, & \dots, & Z_n \\ T_1, & T_2, & T_3, & \dots, & T_n \end{bmatrix} \quad (2.1)$$

A Equação (2.2) é a forma simplificada da Equação (2.1), onde a série temporal completa (Z) é representada pelas observações y_n , sendo que n é o número de observações da série.

$$Z(t) = y_1, \quad y_2, \quad y_3, \quad \dots, \quad y_n \quad (2.2)$$

2.2.1 Composição das Séries Temporais

Uma série temporal, geralmente, é composta de três tipos de padrões, que são ciclos, tendências e períodos de sazonalidade. Uma estratégia para analisar uma série temporal é decompor a série nesses três padrões, ou seja, alterar a série temporal para que se evidencie cada um desses padrões (Hyndman e Athanasopoulos, 2013; Morettin e de Castro Tolo, 2006). A decomposição de uma série temporal é descrita por:

$$Z_t = S_t + T_t + C_t + E_t, \quad (2.3)$$

onde Z_t representa a série temporal, S_t é a sazonalidade da série no período de tempo t , T_t representa a tendência no período t , C_t indica a ciclicidade no período t e E_t representa o restante, ou seja, o erro ou ruído no período t .

2.2.1.1 Tendência em Séries Temporais

A tendência se caracteriza quando há uma mudança (aumento ou diminuição) de longo termo nos dados. Ela pode ser referenciada por uma “mudança de direção” nos dados, ou seja, quando ocorre uma mudança na tendência que os dados avançam, ela muda passando de tendência crescente para tendência decrescente, ou vice-versa (Hyndman e Athanasopoulos, 2013). A tendência não segue, necessariamente, uma proporção linear. O movimento crescente ou decrescente da série pode ser linear, polinomial ou exponencial (Hyndman e Athanasopoulos, 2013; Morettin e de Castro Tolo, 2006).

A Figura 2.3 mostra um exemplo de tendência em uma série temporal. Nela é clara a tendência crescente desde o ano 1949 até o ano de 1960. Em séries temporais, também pode ocorrer tendências mais drásticas, com um rápido acréscimo ou decréscimo diário em algum período. Isto parece ocorrer na série da Figura 2.3, com vários aumentos (picos) e diminuições (vales), todavia, essas mudanças se dão pelos períodos sazonais.

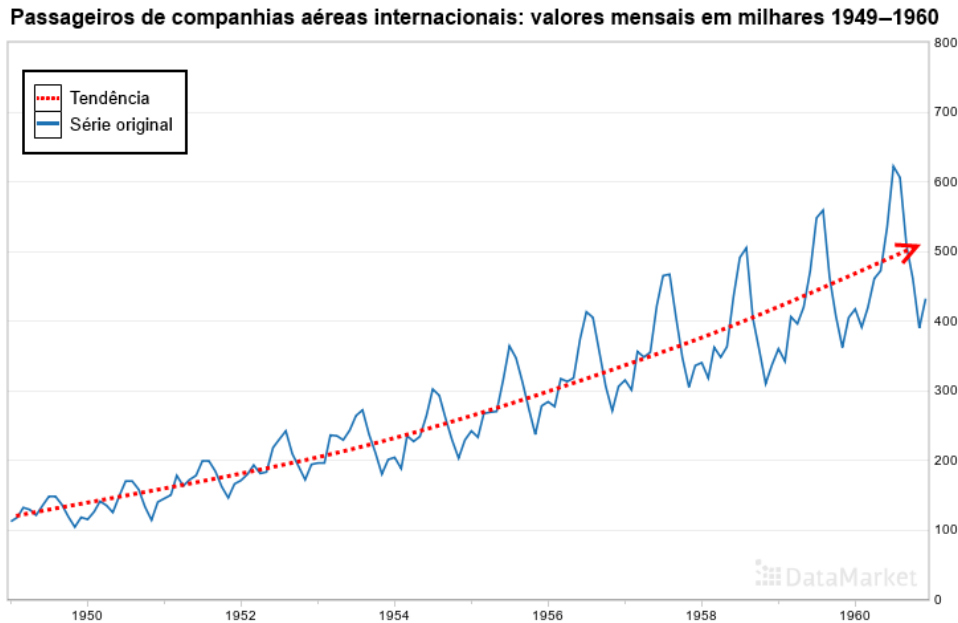


Figura 2.3: Exemplo de uma série temporal com uma tendência crescente. A série original é composta pelo número mensal de passageiros (em milhares) das companhias aéreas internacionais (eixo y), entre Janeiro de 1949 e Dezembro de 1960 (eixo x). Fonte: Adaptado de Hyndman, *Time Series Data Library*, datamarket.com.

2.2.1.2 Sazonalidade em Séries Temporais

O padrão sazonal ocorre quando a série é influenciada por fatores sazonais, ou seja, podem existir padrões que se repetem, por exemplo, no primeiro quarto do ano, padrões mensais, diários, influências de feriados e datas importantes. O importante em relação a sazonalidade é que ela sempre ocorre, sendo sempre fixa em um período conhecido (Hyndman e Athanasopoulos, 2013). A componente de sazonalidade é útil em séries temporais climáticas, econômicas e até mesmo tráfego de veículos, ou tráfego em geral (Morettin e de Castro Toloi, 2006).

Na Figura 2.4 é possível ver que, para todos os anos, ocorrem períodos ascendentes no começo do ano e depois ocorrem períodos descendentes até o fim do ano, este é um exemplo claro de sazonalidade. Nela, também é possível ver uma pequena tendência ascendente ao passar dos anos. Na Figura 2.5, são apresentados somente os dois primeiros anos da série da Figura 2.4, que mostra valores altos a partir do mês de março, de cada ano.

Ainda na Figura 2.5, é possível ver que, para cada ano, ocorre uma pequena queda do mês de Janeiro até o mês de Fevereiro, em seguida acontece um aumento da produção de leite até o mês de Maio. Sequencialmente, ocorre uma queda da produção até Setembro, um pequeno aumento até Outubro, seguido de uma queda até Novembro e, por fim, uma fase ascendente até o fim de Dezembro. Esses períodos, de aumentos e diminuições, ocorrem repetidamente em seus respectivos meses por todos os anos, por isso, essa é uma série temporal predominantemente sazonal.

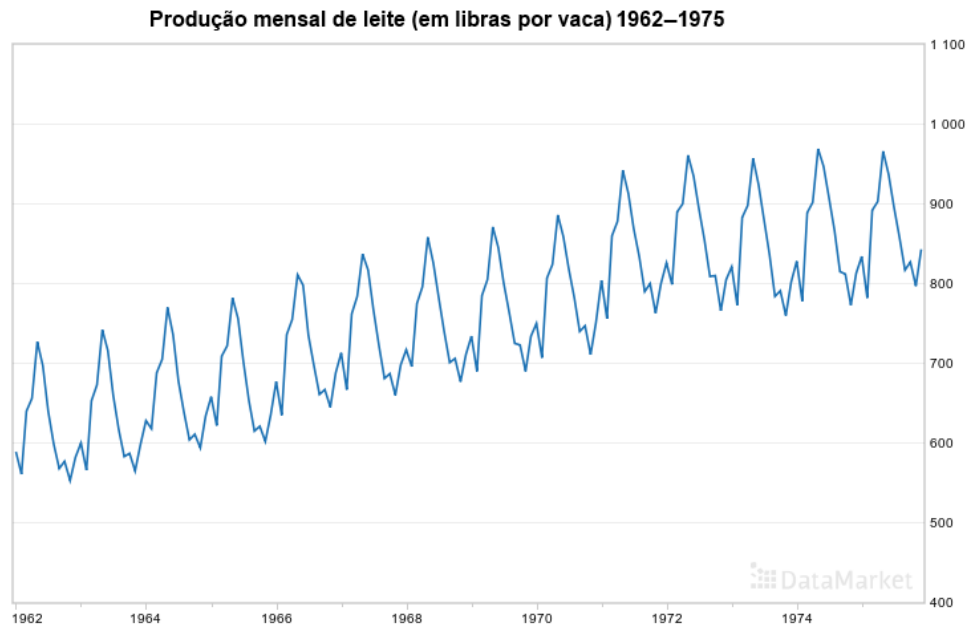


Figura 2.4: Exemplo de série temporal com alta sazonalidade. A série foi criada com a produção mensal de leite, em libras por vaca (eixo y), durante Janeiro de 1962 até Dezembro de 1975 (eixo x). Fonte: Hyndman, *Time Series Data Library*, datamarket.com.

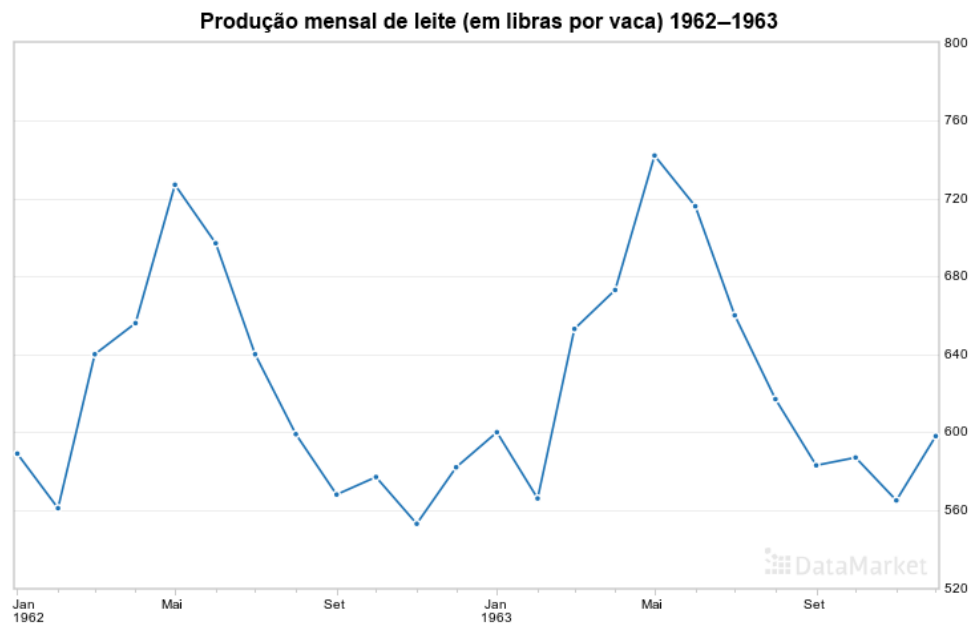


Figura 2.5: Os dois primeiros anos da série da Figura 2.4, nos quais ficam claros a sazonalidade da série, com um pico alto por volta do mês de Maio. A série foi criada com a produção mensal de leite, em libras por vaca (eixo y), durante Janeiro de 1962 até Dezembro de 1963 (eixo x). Fonte: Hyndman, *Time Series Data Library*, datamarket.com.

2.2.1.3 Ciclo em Séries Temporais

O ciclo existe quando os dados exibem aumentos e quedas que não são fixas por períodos. Em alguns casos, os ciclos podem ser repetições de tendências, porém o ciclo não

é o mesmo que sazonalidade (Hyndman e Athanasopoulos, 2013). A principal diferença do ciclo em relação ao componente sazonal é a sua maior extensão, ou seja, as variações ascendentes ou descendentes do ciclo são bem maiores, durando mais tempo do que os períodos sazonais (Morettin e de Castro Toloi, 2006).

Geralmente há confusões entre comportamento sazonal e cíclico, mas eles são bem diferentes. Se as flutuações não são fixas por períodos, então são ciclos. Caso há muitas mudanças em um período do calendário, então são sazonais (Hyndman e Athanasopoulos, 2013). Os períodos dos ciclos são maiores que os sazonais e a magnitude dos ciclos são mais variáveis que a sazonal. Para muitos casos, principalmente em séries econômicas e financeiras, a duração das flutuações cíclicas tem cerca de dois anos, podendo chegar a até dez anos, ou até mesmo mais de 50 anos para ciclos mais extensos (Hyndman e Athanasopoulos, 2013; Morettin e de Castro Toloi, 2006).

Um exemplo de uma série temporal com ciclos é apresentada na Figura 2.6. Nela, ao observar a série completa, é possível reconhecer uma tendência decrescente, no entanto, dentro desta tendência também existem ciclos, que duram cerca de 57 anos. Cada um desses ciclos são compostos basicamente de uma queda abrupta, inicialmente, e em seguida, há um aumento. Dentro desses ciclos, também existem ciclos menores, que duram cerca de cinco anos, formados por picos repetidos. Nesta série predominam os ciclos, pois as repetições de aumento e diminuição são superiores a um ano (diferentemente da série da Figura 2.4), ou seja, não são dadas pela sazonalidade, mas sim por um outro fator diferente.

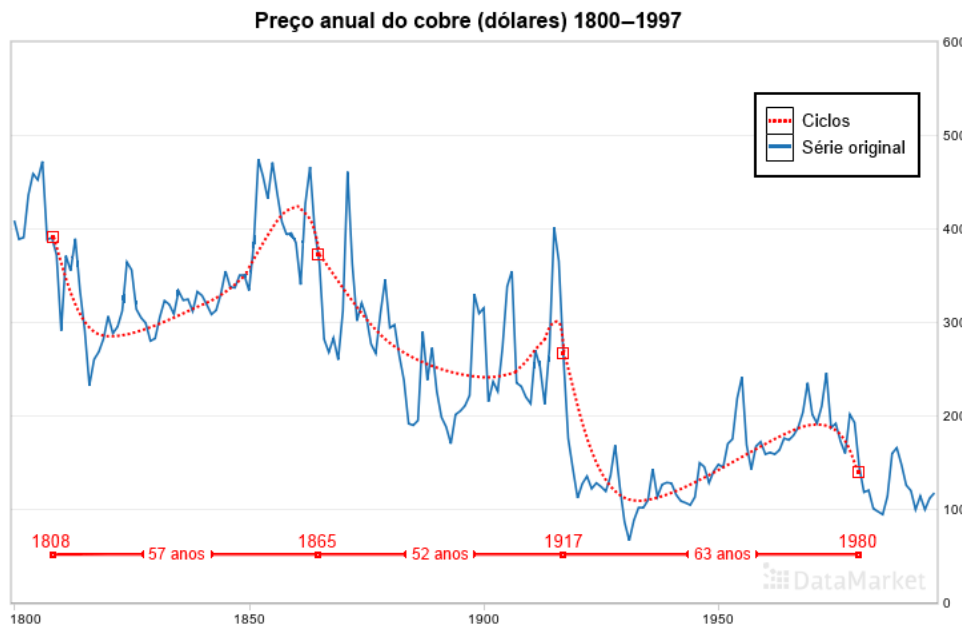


Figura 2.6: Exemplo de série temporal com ciclos. A série mostra o preço (em dólares) anual do cobre (eixo y), em cada ano. As coletas foram feitas durante os anos de 1800 até 1997 (eixo x). No gráfico, em vermelho, é indicado a duração de cada ciclo ao passar dos anos. Fonte: Adaptado de Hyndman, *Time Series Data Library*, datamarket.com.

2.2.1.4 Residual em Séries Temporais

Os termos “residual” ou “flutuações irregulares”, de uma série temporal, geralmente são usados para descrever qualquer variação que restou após remover efeitos de tendência, ciclo e sazonalidade, da série original. Essas flutuações podem ser completamente aleatórias, tornando impossível de prever. Todavia, elas ainda podem exibir algum tipo de correlação fraca, também chamada de correlação a curto prazo, que podem ajudar na previsão (Chatfield, 2004; Kirchgässner *et al.*, 2013).

O residual é representado em uma série temporal da seguinte maneira:

$$R(t) = Z(t) - (T(t) + S(t) + C(t)), \quad (2.4)$$

onde R é o residual, t é o instante em que a coleta do dado da série foi feita, Z é a série temporal, T é a tendência, S a sazonalidade e C representa o ciclo.

2.3 Análise de Séries Temporais

A análise de séries temporais pode ter vários objetivos e interesses diferentes. Por isso, essa análise possui diferentes etapas, nas quais cada etapa pode ser realizada de uma maneira diferente, dependendo da série temporal analisada e do objetivo desta análise (Chatfield, 2004; Morettin e de Castro Tolo, 2006). As principais etapas na análise de séries temporais são: (i) descrição; (ii) modelagem; (iii) predição; e (iv) controle.

A descrição (i) enfoca em descrever os dados, de forma resumida, na forma estatística e na forma gráfica. Geralmente é o primeiro passo na análise, onde são traçados os gráficos que descrevem a série temporal (Chatfield, 2004; Hyndman e Athanasopoulos, 2013). A partir daí, é possível verificar as propriedades principais da série. Por exemplo, construir o gráfico da série pode ajudar a encontrar períodos regulares sazonais, ciclos, tendências e valores discrepantes (*outliers*).

Além do gráfico da série (*time plot*), outros gráficos podem ajudar, como histogramas, diagramas de dispersão (*scatterplots*) e gráficos por períodos (*seasonal plot*) (Hyndman e Athanasopoulos, 2013; Morettin e de Castro Tolo, 2006). Para algumas séries, a variação dos dados é claramente dominada por características óbvias, como modelos simples que descrevem perfeitamente variações com tendências e sazonalidade. Porém, para outras séries, técnicas mais sofisticadas serão necessárias para fornecer uma análise adequada, como análise de um processo estocástico, estacionário, autocorrelação e análises residuais (Chatfield, 2000).

Modelagem (ii) ou explicação, se dá em encontrar um modelo estatístico adequado que descreve bem a geração dos dados. Leva-se em consideração os tipos das séries temporais (discreta ou contínua), os tipos de métodos de predição (univariado ou multivariado), as variáveis preditoras e o modelo matemático final usado na predição (Chatfield, 2000).

Encontrar um modelo que mais se aproxima dos dados originais pode ser muito difícil. Balancear a modelagem é importante, enquanto o modelo multivariado pode representar melhor a realidade, quanto mais variáveis preditoras, mais complexa e de difícil compreensão será a modelagem (Morettin e de Castro Toloi, 2006).

A modelagem da série temporal pode possuir dois enfoques, com propósitos e formas diferentes. Existe o enfoque no domínio temporal, que usa modelos paramétricos, com um número finito de parâmetros para representar a série temporal. O segundo enfoque é conduzido no domínio de frequências, no qual usa modelos não paramétricos (Morettin e de Castro Toloi, 2006).

Os modelos não paramétricos dão enfoque em aplicações em ciências físicas e engenharia, onde comumente decompõe a série em componentes de frequência ou similaridades. O exemplo mais comum, entre os de modelos paramétricos são os modelos ARIMA. Já as redes neurais, são modelos não paramétricos.

A Predição (iii), ou previsão, é a etapa onde de fato é feita a estimativa dos valores futuros de uma série. A maioria dos autores utilizam os termos “predição” e “previsão” como sinônimos, porém existem autores que diferenciam os termos, com “previsão” significando qualquer modo de olhar o futuro, e “predição” denotando um procedimento estatístico de fazer o mesmo (Chatfield, 2004). Nesta monografia, esses termos são tratados como sinônimos. As previsões podem ser a curto prazo, para séries de vendas, produção ou estoque de mercadorias; ou previsões a longo prazo, para séries populacionais, de produtividade ou expectativas de lucro anual (Morettin e de Castro Toloi, 2006).

Outra etapa importante para a conclusão do objetivo final é o controle (iv). As previsões tem um objetivo, que geralmente são antecipar ou controlar um dado processo de acordo com os dados obtidos na previsão, esse podendo ser um processo industrial, econômico, controle de qualidade ou qualquer outro (Chatfield, 2004). Por exemplo, em uma usina nuclear, a partir de medidas previstas de temperatura de um tanque, encontra-se a necessidade do controle de temperatura do mesmo. A fase de controle é feita a partir das previsões da série e depende dos motivos e objetivos da predição. Um controle bem feito depende principalmente da qualidade das predições, ou seja, quanto maior a taxa de acerto, melhor o controle poderá ser realizado.

2.3.1 Tipos de Métodos de Predição

Um método de predição é um procedimento para calcular as previsões a partir de valores do presente e do passado. O método de predição pode ser simplesmente um algoritmo que faz previsões a partir de regras, ou também pode usar algum modelo estatístico de probabilidades. Existem três tipos básicos de métodos de predição para séries temporais: predições baseadas em julgamentos; predições com métodos univariados; e métodos multivariados.

O método das predições a partir de julgamentos usa a experiência de uma pessoa para prever os valores futuros. Esse tipo de predição usa a experiência, intuição, julgamentos subjetivos, conhecimento do área e dos dados, e qualquer outro tipo de informação relevante para fazer as predições (Chatfield, 2000). Como esse método não possui bases matemáticas, ele não é muito usado, porém, ele é geralmente utilizado como uma análise prévia dos dados para ajudar a escolher o melhor modelo de predição. Além disso, esse tipo de predição também é usado quando não há dados dos históricos passados, como por exemplo, no lançamento de um produto novo, ou quando um competidor novo entra no mercado, ou quaisquer outras condições únicas. Para isso, a pessoa que faz a previsão deve possuir um amplo conhecimento e domínio sobre a área referida (Hyndman e Athanasopoulos, 2013).

Os métodos seguintes são os mais utilizados e são usados como base em vários modelos matemáticos de predição. A predição com métodos univariados é representada por uma série temporal univariada, na qual usa somente um tipo de dado ou característica, nas coletas das observações para a série. Esse tipo de predição depende somente dos valores do presente e do passado de uma série temporal única.

A predição com métodos multivariados utiliza uma série temporal multivariada para representar os dados. Esse método é chamado de multivariado pois possui muitas variáveis preditoras, ou seja, possui muitos tipos de dados diferentes no mesmo instante da observação. Essa série temporal, então, é formada por mais de uma série temporal, havendo uma série para cada característica diferente.

Uma série temporal multivariada é representada como:

$$Z(t) = [Z_1(t), \dots, Z_n(t)]', \quad (2.5)$$

onde $Z(t)$ é um vetor, de ordem $r \times 1$, que representa a observação de um fenômeno qualquer no instante t . O parâmetro t refere-se ao tempo, que varia uniformemente. Finalmente, os elementos $Z_i(t)$ são as características do fenômeno observado, com $i \in [1, n]$. Por exemplo, suponha que a série $Z(t)$ represente observações de demanda de eletricidade em uma região, cada observação $Z_i(t)$ dessa série poderia ter cinco elementos. Esses elementos podem representar, por exemplo, a temperatura atual da região, o poder econômico da população, o número de pessoas na população, a hora do dia e o dia da semana (Morettin e de Castro Tolo, 2006).

Uma série temporal univariada é representada do mesmo modo, porém é utilizado somente uma característica na observação, sendo representada como:

$$Z(t) = [Z_0(t)]', \quad (2.6)$$

onde $Z_0(t)$ é a única característica do fenômeno observado. Essa única característica pode ser os bits transmitidos em um roteador, por exemplo, para a predição do tráfego de redes.

2.3.2 Estacionariedade e Diferenciação

Série temporal estacionária é uma série que se desenvolve no tempo através de um processo estocástico, onde a média dos valores é constante (Morettin e de Castro Tolo, 2006), ou seja, ela se desenvolve no tempo de forma aleatória ao redor de uma média constante. No geral, séries estacionárias não possuem padrões previsíveis ao longo do tempo (Hyndman e Athanasopoulos, 2013) e exibem essencialmente as mesmas propriedades estatísticas (Haykin e Moher, 2009).

Existem dois tipos básicos de estacionariedade. Uma série temporal estritamente estacionária (*strictly stationary*) não depende do tempo que a série é obtida, isto é, a função de distribuição comum do processo estocástico não é alterada por uma mudança no tempo (Hyndman e Athanasopoulos, 2013; Kirchgässner *et al.*, 2013). Entretanto, esse conceito de estacionariedade estrita é muito difícil de ser encontrado na prática. A maior parte das séries temporais são alteradas de acordo com momentos no tempo, mas ainda assim possuem períodos estacionários, por isso são chamadas de séries temporais com estacionariedade fraca (*weak stationarity*) (Kirchgässner *et al.*, 2013). Essas séries podem ser estacionárias durante um período muito longo, ou durante um período muito curto, mudando de nível e inclinação (Morettin e de Castro Tolo, 2006).

De forma genérica, processos estacionários são gerados por fenômenos físicos que evoluíram para um estado estável de comportamento. Já os processos gerados por um fenômeno instável, são chamados de “não estacionários” (Haykin e Moher, 2009). Séries temporais com tendências, sazonalidade e ciclos, geralmente, possuem estacionariedade fraca ou são realmente não estacionárias, já que observações em tempos diferentes afetam os valores da série temporal. Como a maioria das séries estacionárias são de estacionariedade fraca, elas são comumente chamadas somente de séries estacionárias.

Um exemplo de série temporal estacionária é exibido na Figura 2.7, na qual verifica-se a existência de ciclos, a cada 10 anos em média. Todavia, mesmo com esses ciclos, a série ainda é estacionária, pois, eles são muito irregulares, onde as vezes ocorrem aumentos e diminuições muito mais bruscas ou com picos mais altos que o normal. Esses ciclos são causados quando a população de lince fica muito grande em relação a alimentação disponível, então, os lince param procriar e a população cai para números muito baixos. Em seguida, as fontes de comida aumentam, o que permite novamente o crescimento da população. Essas mudanças, no longo prazo, deixam os ciclos com baixa previsibilidade, portanto, a série é não estacionária (Hyndman e Athanasopoulos, 2013).

Muitos dos procedimentos de análise estatística de séries temporais supõem que estas sejam estacionárias. Assim, pode ser necessário transformar os dados originais em uma série estacionária, caso ela não seja (Morettin e de Castro Tolo, 2006). A transformação mais comum para tornar uma série estacionária é a diferenciação. Na diferenciação são calculadas diferenças sucessivas entre valores consecutivos da série original, até se obter

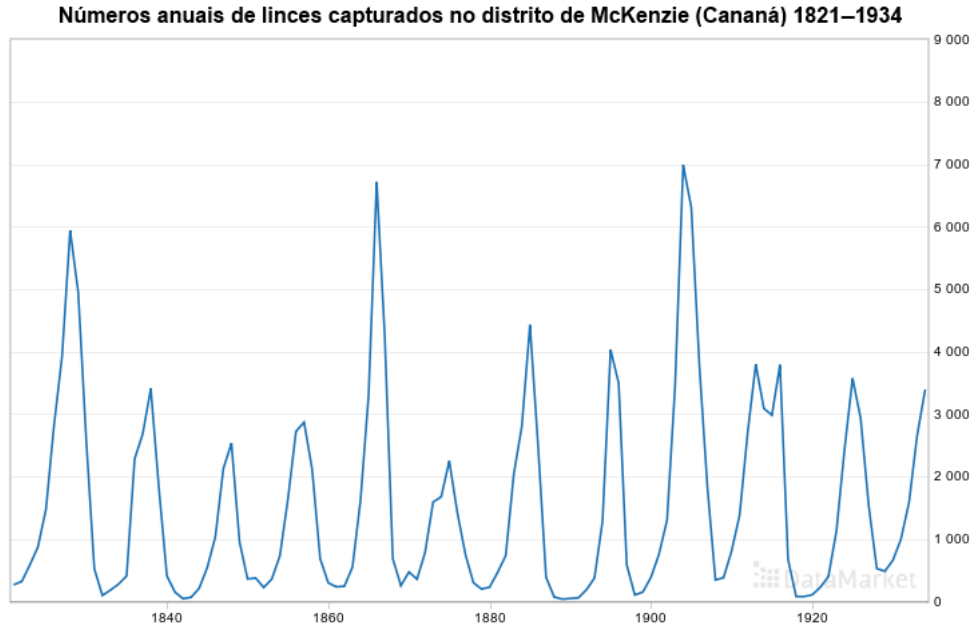


Figura 2.7: Série temporal estacionária. Série criada a partir do número de lincas capturados (eixo y) no distrito de McKenzie, região noroeste do Canadá. Dados coletados entre 1921 e 1934 (eixo x). Fonte: Hyndman, *Time Series Data Library*, datamarket.com.

uma série temporal estacionária. A primeira diferenciação pode não tornar a série estacionária, neste caso é necessário continuar calculando diferenciações sucessivas até chegar no resultado esperado.

A primeira diferenciação da série $Z(t)$ é definida por:

$$\Delta Z(t) = Z(t) - Z(t - 1), \quad (2.7)$$

a segunda diferenciação é dada por:

$$\begin{aligned} \Delta^2 Z(t) &= \Delta[\Delta Z(t)] \\ &= \Delta[Z(t) - Z(t - 1)] \\ &= Z(t) - Z(t - 1) - (Z(t - 1) - Z(t - 2)) \\ &= Z(t) - 2Z(t - 1) + Z(t - 2) \end{aligned} \quad (2.8)$$

Expandindo a segunda diferenciação para n diferenciações sucessivas, com $n > 1$, temos:

$$\Delta^n Z(t) = \Delta[\Delta^{n-1} Z(t)] \quad (2.9)$$

Na maioria dos casos, uma ou duas diferenciações são suficientes para tornar a série estacionária. A Figura 2.8 mostra uma série temporal não estacionária, à esquerda (a), que após as diferenciações de seus valores, a série temporal se tornou estacionária, à direita (b). Nota-se que a diferenciação estabilizou os valores médios da série temporal, removendo os fatores de tendência, sazonalidade e ciclos.

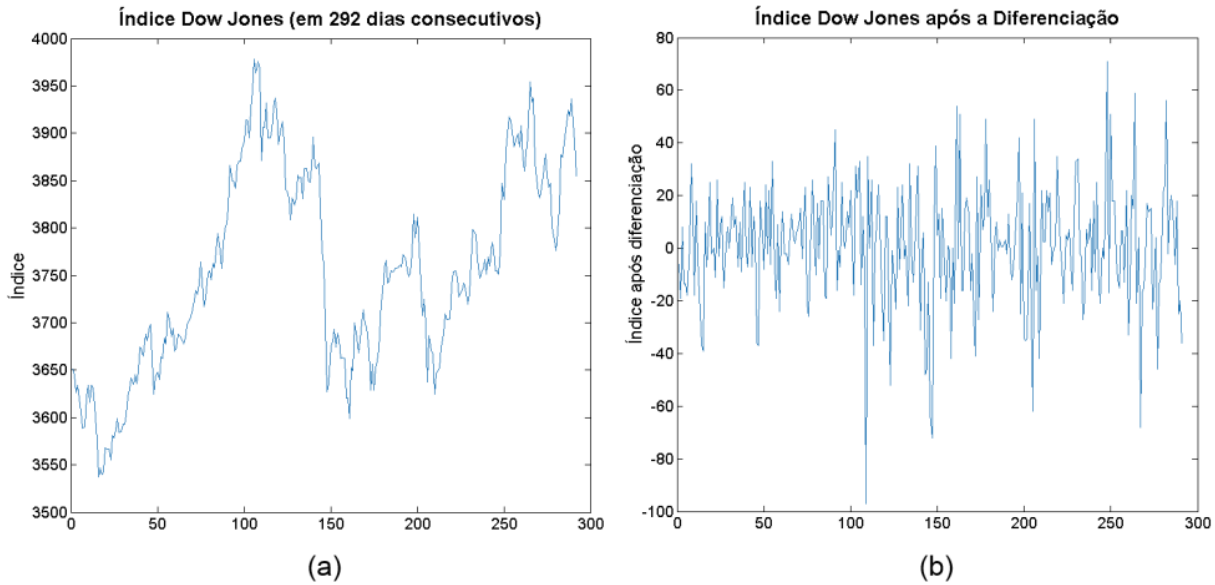


Figura 2.8: (a) Série temporal com o índice Dow Jones, em 292 dias consecutivos. (b) Série temporal com as diferenciações do índice Dow Jones anterior. Fonte da série temporal: Hyndman, *Time Series Data Library*, datamarket.com.

Além da análise gráfica da série temporal, a função de autocorrelação pode ajudar a verificar se a série é estacionária ou não. Essa análise com a função de autocorrelação é detalhada logo a seguir.

2.3.3 Autocorrelação

Em estatística a correlação, ou coeficiente de correlação, indica o nível de relacionamento ou dependência entre duas variáveis. A correlação refere-se somente à relação entre duas variáveis, não obstante, a correlação não implica em causalidade, ou seja, não necessariamente existe causa e efeito. A partir daí, chega-se no conceito de autocorrelação, que é a medida de correlação entre valores sucessivos de uma série temporal (Chatfield, 2004).

Em séries temporais com tendências e sazonalidade, é comum encontrar correlações entre residuais sucessivos, que estão separados por um intervalo curto de tempo. Isso é chamado de autocorrelação de curto prazo. Com isso, modelagens mais sofisticadas podem ser usadas para melhorar a predição.

A autocorrelação em uma série temporal é feita entre valores sucessivos da mesma. Então, dadas n observações sucessivas, $x_1, x_2, x_3, \dots, x_n$, em uma série temporal discreta, a autocorrelação é calculada para cada par sucessivo $(x_1, x_2), (x_2, x_3), \dots, (x_{n-1}, x_n)$. Na função de autocorrelação, esses pares são representados por *lags*, por exemplo, o $\text{lag} = 1$ representa o primeiro par (x_1, x_2) , e assim por diante até o $\text{lag} = n$.

A Figura 2.9, mostra a função de autocorrelação (*Autocorrelation Function* — ACF) aplicada na série temporal do índice Dow Jones e na série temporal das diferenciações

do índice Dow Jones (Figura 2.8). Pode-se identificar que a série (a), da figura, é não estacionária, pois o ACF diminui lentamente. Para a série (b), a autocorrelação chega até próximo de zero muito rapidamente, e os valores de autocorrelação seguintes se alteram também rapidamente, pois a série temporal é estacionária (Hyndman e Athanasopoulos, 2013).

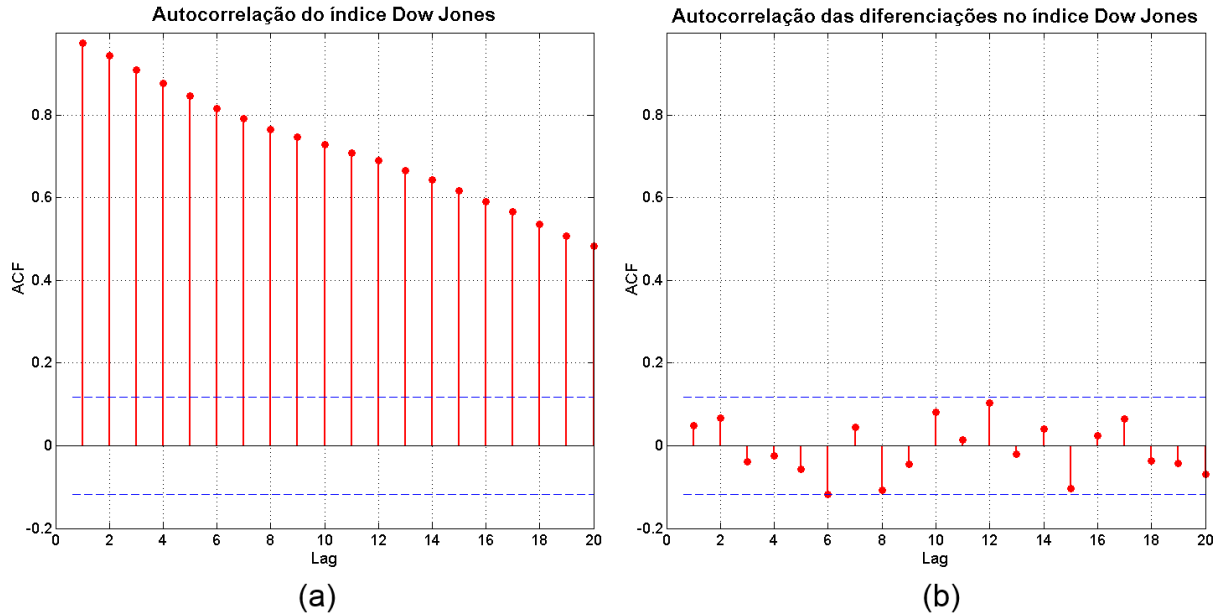


Figura 2.9: (a) Gráfico da função de autocorrelação aplicada na série temporal do índice Dow Jones (em 292 dias consecutivos). (b) Gráfico da função de autocorrelação aplicada na diferenciação do índice Dow Jones. Fonte da série temporal: Hyndman, *Time Series Data Library*, datamarket.com.

2.3.4 Variáveis Predictoras e Modelos Básicos de Predição de Séries Temporais

A modelagem de uma série temporal pode ser feita de três maneiras, modelo de série temporal simples, modelo explanatório com variáveis predictoras e modelo misto. O modelo misto mescla o modelo de série temporal com um modelo explanatório de variáveis predictoras (Hyndman e Athanasopoulos, 2013).

As variáveis predictoras são variáveis numéricas que identificam e representam uma situação da série temporal. Por exemplo, suponha que queremos prever a demanda de eletricidade (E) de hora em hora, de uma região quente, durante o verão. Um modelo com variáveis predictoras será da forma:

$$E = f(ta, pe, pop, hd, ds, erro), \quad (2.10)$$

onde ta representa a temperatura atual, pe indica o poder econômico, pop é a população,

hd é a hora do dia e ds representa o dia da semana. Esse relacionamento não é completamente exato, ou seja, existem mais variáveis que afetam a demanda da eletricidade. O *erro* usado, permite variações aleatórias dos efeitos que são relevantes, porém não inclusos no modelo. Isso é chamado modelo explanatório, porque ajuda a explicar o que causa a variação na demanda de eletricidade (Hyndman e Athanasopoulos, 2013).

Já um modelo de predição de série temporal para a demanda de eletricidade é da forma:

$$E_{t+1} = f(E_t, E_{t-1}, E_{t-2}, \dots, erro), \quad (2.11)$$

onde t é a hora atual; $t + 1$ é a próxima hora, ou seja, é o instante que será previsto; $t - 1$ é a hora anterior e assim por diante. Nesse modelo a predição é baseada somente nos valores passados da variável, por isso, é chamado de modelo univariado. Com isso, o modelo univariado não conta com variáveis externas, que também podem afetar o sistema (Hyndman e Athanasopoulos, 2013).

O terceiro modelo é o modelo misto, que combina os dois modelos anteriores. Ele é representado na forma:

$$E_{t+1} = f(E_t, ta, pe, pop, hd, ds, erro), \quad (2.12)$$

onde as variáveis são as mesmas das Equações (2.10) e (2.11), porém, nesse modelo elas são combinadas. O modelo explanatório misto possui várias nomenclaturas diferentes, como: modelo de regressão dinâmica; modelo de dados em painel; modelos longitudinais; modelo de função de transferência e modelo de sistemas lineares (assumindo que seja linear) (Hyndman e Athanasopoulos, 2013).

O modelo explanatório misto é útil pois, além do histórico dos valores da variável, ele usa a informação de outras variáveis para representá-lo. Entretanto, há várias razões para usar o modelo de série temporal e não o modelo explanatório misto (Hyndman e Athanasopoulos, 2013), sendo elas:

- No modelo explanatório misto o sistema pode ficar difícil de entender e de medir os relacionamentos que afetam o comportamento;
- É necessário saber como prever as variáveis preditoras para então prever a variável de interesse, isso pode ser muito difícil;
- O interesse principal pode ser somente de prever o que vai acontecer e não porquê acontece.
- O modelo de séries temporais pode ser mais preciso do que o modelo explanatório e o modelo misto.

2.4 Predição de Séries Temporais

A predição de uma série temporal é a estimação de seus valores futuros a partir de seus valores passados e valor atual, ou seja, usar os valores passados e o valor atual da série para aproximar ao máximo e descobrir como ela irá se comportar no futuro (Hyndman e Athanasopoulos, 2013). A Equação (2.13) descreve, de forma básica, a predição de uma série temporal, a partir dos valores atual e passados $(Z_t, Z_{t-1}, Z_{t-2}, \dots)$. A predição do próximo valor é dada por Z_{t+1} , que é o valor que foi previsto no próximo instante no tempo $(t + 1)$.

$$Z_{t+1} = f(Z_t, Z_{t-1}, Z_{t-2}, \dots) \quad (2.13)$$

Os passos básicos de uma predição são os seguintes (Hyndman e Athanasopoulos, 2013):

1. Definição do problema: pode ser o passo mais difícil. Definir o problema requer um entendimento de como a predição será usada; para que será usada; como os dados serão coletados; quais dados são importantes; e qual a precisão da predição que será necessária para atingir o objetivo.
2. Coleta de informações: existem pelo menos dois tipos de informações que podem ser usadas. A primeira são dados estatísticos, a segunda são as experiências acumuladas por especialistas no determinado assunto, que podem afetar as previsões de acordo com o necessário. Além disso, pode ser difícil obter um grande histórico de dados para ser usado na predição. Entretanto, dados muitos antigos também podem não ser muito úteis, devido a mudanças no sistema que será previsto.
3. Análise preliminar exploratória: deve-se analisar os dados de forma gráfica. Isso é importante para encontrar padrões consistentes, como as tendências; ciclos; sazonalidade; valores discrepantes; correlação dos dados; e entre outros.
4. Escolha do modelo adequado: o modelo depende da quantidade de dados disponíveis; da força do relacionamento entre variáveis que serão previstas; das variáveis que serão usadas para predição; e de como a previsão será usada. Cada modelo se baseia em uma técnica ou suposição diferente que afeta os dados. Existem vários modelos que serão mais detalhados adiante, como regressão, suavização exponencial, ARIMA e redes neurais artificiais.
5. Usar e avaliar o modelo de predição: depois do modelo ser escolhido, ele será usado para a predição. O desempenho do modelo somente pode ser devidamente avaliado depois que os dados para a predição estarem disponíveis. Para saber se o modelo escolhido realmente se saiu bem, existem vários métodos para avaliar a precisão da predição. Os métodos mais básicos usam os erros em escalas, erros de porcentagem ou erros residuais para saber se a predição obteve uma precisão aceitável.

2.4.1 Diferentes Horizontes de Predição

Durante o primeiro passo da predição, na definição do problema, também é definido o horizonte da predição, ou seja, o quão no futuro é necessário prever. Quanto maior o horizonte de predição, mais difícil e menos precisa a predição será (Morettin e de Castro Toloi, 2006). Previsões a curto prazo são aquelas que estão interessadas no próximo instante no tempo, ou em instantes muito próximos, como por exemplo: prever segundos adiantes, para predição de tráfego de rede; ou prever dias adiantes, para predição de vendas de uma loja. Já as previsões a longo prazo são aquelas que querem prever longos horizontes no tempo, como por exemplo: prever horas adiante, para predição de tráfego de rede; ou prever anos adiante, para predições de vendas de uma loja.

Dado o horizonte de previsão h e um conjunto de observações de uma série temporal até o instante t no tempo, o valor que será previsto será $t+h$ (Figura 2.10). Neste caso, a origem da previsão é t , e $\hat{Z}_t(h) = \hat{y}_t$ é a previsão do ponto $Z(t+h)$ de origem t e horizonte h . A partir da previsão é calculado o erro residual (e_i):

$$e_i = Z(t+h) - \hat{Z}_t(h) = y_i - \hat{y}_i, \quad (2.14)$$

onde $Z(t+h) = y_i$ é o valor real observado e $\hat{Z}_t(h) = \hat{y}_i$ é o valor previsto de y no instante i . Assim, o erro residual é a diferença *valor real* – *valor previsto*.

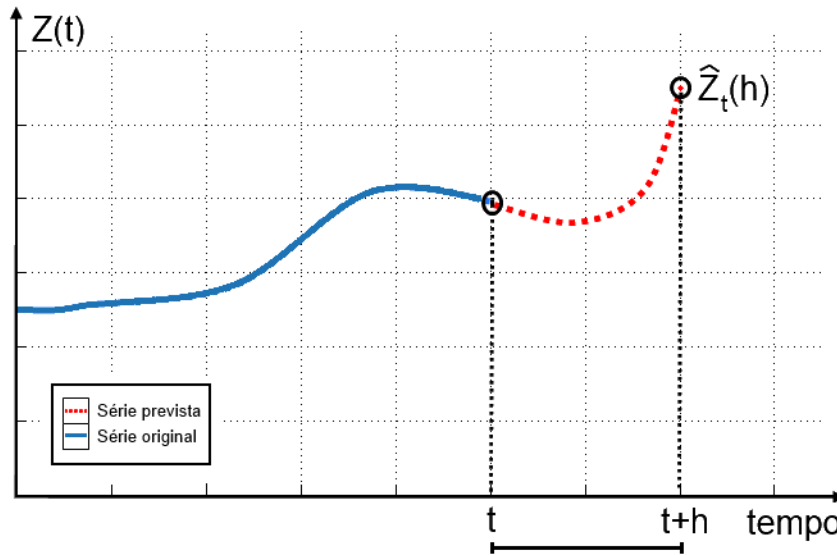


Figura 2.10: Observações de uma série temporal qualquer, com previsões de origem t e horizonte h .

2.4.2 Diagnóstico dos Erros Residuais da Predição

Como mostrado na Equação (2.14), o residual em previsão é a diferença entre um valor observado e o valor previsto. Para séries temporais, o erro residual é baseado em

um passo da previsão, ou seja, $\hat{y}_t$ é a previsão de y_t baseada nas observações $y_1, y_2, \dots, y_{t-1}$ (Hyndman e Athanasopoulos, 2013).

Um bom método de previsão produz residuais com as seguintes propriedades (Hyndman e Athanasopoulos, 2013):

- Os residuais devem ser não correlacionados ou com correlação a curto prazo. Caso haja uma correlação forte entre os residuais, então houve informações deixadas nos residuais que poderiam ser usadas no cálculo da previsão.
- Os residuais devem ter média zero, ou média constante e próxima de zero. Caso a média não obedeça esse padrão, então a previsão foi tendenciosa.

Qualquer método de previsão que não atenda essas características pode ser melhorado. Essas propriedades são importantes para saber se um método usa a informação de modo correto, no entanto, atender essas propriedades não garante que o método é o melhor (Hyndman e Athanasopoulos, 2013).

2.5 Modelos de Predição

Um modelo de predição é uma descrição matemática de uma série temporal. Desse modo, modelos de predição não são baseados em métodos ou procedimentos de predição (Morettin e de Castro Toloí, 2006). O modelo de predição é a descrição probabilística, estatística, ou com alguma base matemática, que tenta definir a série temporal, tentando aproximar ao máximo o modelo gerado à série temporal original. O modelo ARIMA e redes neurais artificiais são exemplos de modelos de predição.

Diferentemente do modelo de predição, o método de predição é o que fornece a base para os estudos teóricos e a criação dos modelos, ou seja, métodos de predição são simplesmente procedimentos computacionais para calcular as combinações dos valores passados que levam a valores futuros (Morettin e de Castro Toloí, 2006). Um exemplo de método de predição é o método dos mínimos quadrados, que é o método que a maioria dos modelos estatísticos, como o ARIMA, segue para fazer a predição.

Como já foi citado, existem dois grupos possíveis para os modelos de predição. No primeiro grupo, tem-se os modelos paramétricos, que usam um número finito de parâmetros, para representar a distribuição da série temporal. Os principais exemplos de modelos paramétricos são: suavização exponencial, média móvel, autorregressão, Holt-Winters e ARIMA (Box *et al.*, 2008; Hyndman e Athanasopoulos, 2013; Kalekar, 2004; Morettin e de Castro Toloí, 2006). Já no segundo grupo de modelos de predição, estão os modelos não paramétricos. Estes, mesmo sem o uso de parâmetros, se adaptam a diversos comportamentos ocorridos nas séries temporais, como por exemplo, SVM, Algoritmos Genéticos, RNAs tradicionais ou RNAP e combinações híbridas de RNAs com lógica nebulosa (Chao

et al., 2011; Crone *et al.*, 2011; Hornik e Leisch, 2000; Hyndman e Athanasopoulos, 2013; Sahin *et al.*, 2012; Stepnicka *et al.*, 2011).

2.5.1 Métodos Simples de Predição

Existem alguns métodos simples de predição que podem ser usados como comparativos (*benchmark*), para saber se a predição é aceitável ou não. Os métodos simples de predição são:

- Método da média (*average method*): os valores futuros são iguais a média do histórico dos dados;
- Método ingênuo (*naive method*): todas as previsões são simplesmente o valor da última observação;
- Método ingênuo sazonal (*seasonal naive method*): cada valor da previsão será igual ao valor observado na mesma época sazonal anterior, por exemplo, os valores previstos de um mês serão os mesmos valores desse mês no ano anterior (considerando o período sazonal de um ano);
- Método com *drift* (*drift method*): é uma variação do método ingênuo, que permite o valor previsto aumentar ou diminuir com o passar do tempo, onde a quantidade dessa mudança, que é chamada de *drift*, é a mudança média vista no histórico dos dados;

A Figura 2.11 mostra várias previsões de uma série temporal qualquer, cada previsão utiliza um método simples diferente. A série original possui períodos sazonais, por isso é possível usar o método ingênuo sazonal, que dos métodos mais simples, geralmente é o melhor. Esses métodos, na maioria dos casos, são usados somente como efeito comparativo. Caso um modelo de predição seja pior que os métodos simples, então, ele deve ser descartado (Hyndman e Athanasopoulos, 2013).

2.5.2 Suavização Exponencial e Holt-Winters

A Suavização Exponencial, também chamada de Alisamento Exponencial, é um tipo de modelo que trata as séries temporais, levando em consideração suas flutuações com o passar do tempo (Morettin e de Castro Tolo, 2006). O foco da suavização exponencial é dado pelas observações mais recentes, de modo que as observações passadas recebam um peso exponencialmente menor, na influência dos novos valores. Em outras palavras, os valores mais recentes recebem um peso maior na previsão da série temporal (Kalekar, 2004). Os modelos de predição mais novos e bem sucedidos, como o ARIMA, foram baseados na suavização exponencial (Hyndman e Athanasopoulos, 2013).

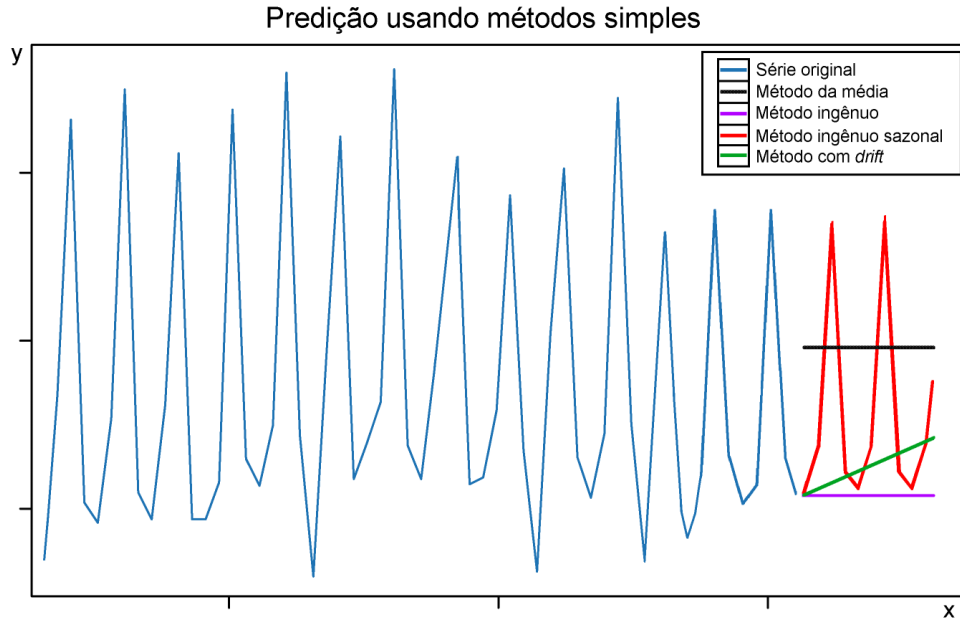


Figura 2.11: Exemplo de como são as previsões, de uma série temporal qualquer, usando os métodos mais simples (média, ingênuo, ingênuo sazonal e *drift*). Fonte da série temporal: Hyndman, *Time Series Data Library*, datamarket.com.

O método mais simples de suavização exponencial é a Suavização Exponencial Simples (SES), no qual as previsões produzidas são compostas por médias ponderadas de observações passadas, com os pesos de cada uma decaindo exponencialmente, quanto mais velhas forem as observações. Esse método gera previsões rápidas e confiáveis, além de que ele funciona para uma grande variedade de séries temporais. Entretanto, a SES é mais adequada para prever séries que não possuem tendência e sazonalidade (Hyndman e Athanasopoulos, 2013).

A SES foi influenciada por dois métodos simples de predição, sendo eles, o método ingênuo e o método da média. No método ingênuo, todas as previsões são iguais ao último valor observado da série, ou seja, a observação atual é a única importante e todas as observações anteriores não fornecem informação útil para a previsão. No método da média, todas as previsões são iguais a média dos dados observados, isto é, todas as observações tem a mesma importância e tem o mesmo peso na previsão. Desse modo, o método SES é uma combinação dos métodos ingênuos e da média, fazendo com que as previsões sejam calculadas a partir das médias ponderadas, nas quais as observações mais novas recebem um peso maior na previsão (Hyndman e Athanasopoulos, 2013).

A Equação (2.15) mostra a equação de previsão, de um passo a frente, usando o método de SES:

$$\hat{y}_{t+1} = \alpha \hat{y}_t + \alpha(1 - \alpha) \hat{y}_{t-1} + \alpha(1 - \alpha)^2 \hat{y}_{t-2} + \dots, \quad (2.15)$$

onde $0 \leq \alpha \leq 1$ é o parâmetro de suavização, t é o instante de tempo atual, e a previsão de um passo adiante no futuro, $\hat{y}_{t+1}$, é dada pela média ponderada de todas as observações da série $y_1, \dots, y_t$. A taxa de diminuição do peso é controlada pelo parâmetro α . Se

α for pequeno (próximo de 0), as observações mais antigas receberam um peso alto, ou seja, elas influenciarão mais nas predições. Caso α for grande (próximo de 1), um peso maior é dado somente às observações mais recentes, fazendo com que as informações mais antigas não tenham uma influência significativa na predição da série temporal (Hyndman e Athanasopoulos, 2013; Morettin e de Castro Toloi, 2006).

A Suavização Exponencial de Holt (SEH), é uma extensão da SES para prever séries temporais com componentes de tendência. Para isso, foi adicionado um parâmetro β extra, que é usado como o parâmetro de suavização para a tendência da série (Hyndman e Athanasopoulos, 2013; Morettin e de Castro Toloi, 2006). Também existe uma extensão do SEH, chamado de Suavização Exponencial Sazonal de Holt-Winters (HW), que além da tendência, também captura a sazonalidade da série temporal.

O método HW possui duas variações que se diferem na natureza da componente sazonal. Um deles é o método HW Aditivo, que é usado quando a variação sazonal é constante com o passar do tempo. O outro, é o método HW Multiplicativo, que é optado quando a variação sazonal muda proporcionalmente no tempo, isto é, a componente sazonal varia em porcentagens (Hyndman e Athanasopoulos, 2013; Kalekar, 2004; Morettin e de Castro Toloi, 2006).

2.5.3 Modelagem ARIMA

A modelagem ARIMA fornece outra abordagem para a previsão de séries temporais. Enquanto modelos de suavização exponencial são baseados na descrição de tendência e sazonalidade nos dados da série, os modelos ARIMA seguem o lado da autocorrelação desses dados (Hyndman e Athanasopoulos, 2013). O modelo ARIMA é um modelo que se adapta muito bem a diversas séries temporais e é amplamente usado na predição delas. Além disso, o ARIMA é gerado a partir da combinação de outros modelos. Antes de especificar o modelo ARIMA, é necessário detalhar as definições dos modelos que o compõem.

2.5.3.1 Modelo Autorregressivo

No modelo Autorregressivo (*Autoregressive* — AR), a previsão da variável de interesse é feita usando uma combinação linear dos valores passados da série temporal, ou seja, os valores passados da variável aleatória são usados para explicá-la. O termo “autor-regressão” indica que é feita uma regressão da variável contra si mesma. Assim, melhores resultados serão obtidos para séries que possuem autocorrelação entre os valores de suas observações. Esse modelo possui uma ordem (p), que é o parâmetro que indica a quantidade de valores passados que são usados para realizar a predição do valor futuro (Box *et al.*, 2008; Hyndman e Athanasopoulos, 2013).

Um modelo AR é uma representação de um tipo de processo estocástico, ou seja, uma

coleção de variáveis aleatórias, que representam a evolução de um sistema no passar do tempo. Uma série temporal, é dita um processo autorregressivo de ordem p , ou AR(p), se ela for uma soma linear ponderada dos p valores passados mais um choque aleatório (Chatfield, 2000; Hyndman e Athanasopoulos, 2013), de tal forma que:

$$y_t = \phi_1 y_{t-1} + \phi_2 y_{t-2} + \cdots + \phi_p y_{t-p} + e_t, \quad (2.16)$$

onde y_t é a observação no instante t ; ϕ_i é o parâmetro do modelo, com $i = 1, 2, \dots, p$; e_t representa um processo aleatório (*white noise*), no instante t , com média zero e variância σ_e^2 , isto é, média e variância constantes.

2.5.3.2 Modelo de Média Móvel

O modelo de Média Móvel (*Moving Average* — MA) não usa os valores passados da variável de previsão em uma regressão, na verdade, ele usa os erros de previsões passadas em um modelo similar a regressão, isto é, a previsão é feita a partir da média móvel ponderada dos valores dos erros de previsões anteriores. As médias móveis são interpretadas como mudanças de valores não previstas, decorrente de eventos externos ou choques aleatórios na série temporal (Box *et al.*, 2008; Hyndman e Athanasopoulos, 2013; Morettin e de Castro Toloi, 2006).

A ordem (q) deste modelo indica o número de períodos passados que são usados para realizar a predição do valor futuro. Uma série temporal, é dita um processo de médias móveis de ordem q , ou MA(q), se ela for uma soma linear ponderada dos últimos q valores de choques aleatórios (Chatfield, 2000; Hyndman e Athanasopoulos, 2013), de tal forma que:

$$y_t = e_t + \theta_1 e_{t-1} + \theta_2 e_{t-2} + \cdots + \theta_q e_{t-q}, \quad (2.17)$$

onde y_t é a observação no instante t ; θ_j é o parâmetro do modelo, com $j = 1, 2, \dots, q$; e_t representa um processo aleatório, no instante t , com média zero e variância σ_e^2 .

2.5.3.3 Modelo Autorregressivo de Média Móvel

Caso seja feita uma junção com p termos autorregressivos mais q termos de médias móveis, é gerado o modelo Autorregressivo de Média Móvel (*Autoregressive Moving Average* — ARMA). A importância do processo ARMA é que muitos conjunto de dados reais são aproximados de uma maneira mais fiel do que um processo puramente AR ou MA. Um modelo ARMA(p, q) indica a ordem p de um processo AR e a ordem q de um processo MA (Chatfield, 2000; Morettin e de Castro Toloi, 2006), de tal forma que:

$$y_t = \phi_1 y_{t-1} + \phi_2 y_{t-2} + \cdots + \phi_p y_{t-p} + e_t + \theta_1 e_{t-1} + \theta_2 e_{t-2} + \cdots + \theta_q e_{t-q}, \quad (2.18)$$

onde y_t é a observação no instante t ; ϕ_i é o parâmetro do modelo de autorregressão, com $i = 1, 2, \dots, p$; θ_j é o parâmetro do modelo de média móvel, com $j = 1, 2, \dots, q$; e_t representa um processo aleatório, no instante t , com média zero e variância σ_e^2 .

2.5.3.4 Modelo Autorregressivo Integrado de Média Móvel

Na prática, muitas séries temporais são não estacionárias, por isso, não é recomendável aplicar processos estacionários diretamente (Chatfield, 2000), ou seja, os processos AR, MA e ARMA precisam trabalhar com séries não estacionárias. Para remover a estacionariedade de uma série temporal, uma diferenciação sucessiva ou única deve ser aplicada (veja a Seção 2.3.2). Ao combinar as diferenciações com a autorregressão e a média móvel, é obtido o modelo Autorregressivo Integrado de Média Móvel (*Autoregressive Integrated Moving Average* — ARIMA) não sazonal, no qual o termo “Integrado” se refere ao reverso da diferenciação. Em outras palavras, caso a série temporal seja diferenciada d vezes e ajustada ao modelo ARMA(p,q), então o modelo para a série temporal original não diferenciada é dito como modelo ARIMA(p,d,q) (Hyndman e Athanasopoulos, 2013).

A modelagem ARIMA é largamente utilizada na predição de séries temporais, uma vez que ela possui várias propriedades e parâmetros que a torna capaz de lidar com a maioria dos tipos diferentes de séries temporais encontradas (Morettin e de Castro Toloi, 2006). O modelo ARIMA(p,d,q) é composto pela ordem p da parte autorregressiva, pelo grau d de diferenciações realizadas e pela ordem q da parte da média móvel. O parâmetro d representa o número de diferenciações necessárias para deixar a série temporal não estacionária (Chatfield, 2000; Hyndman e Athanasopoulos, 2013). Ao combinar o modelo ARMA(p,q) com as d diferenciações, obtemos o modelo ARIMA(p,d,q), que pode ser escrito como:

$$y_t = \phi_1 y'_{t-1} + \phi_2 y'_{t-2} + \dots + \phi_p y'_{t-p} + e_t + \theta_1 e_{t-1} + \theta_2 e_{t-2} + \dots + \theta_q e_{t-q}, \quad (2.19)$$

onde y'_t é a observação, no instante t , da série já diferenciada; ϕ_i é o parâmetro do modelo de autorregressão, com $i = 1, 2, \dots, p$; θ_j é o parâmetro do modelo de média móvel, com $j = 1, 2, \dots, q$; e_t representa um processo aleatório, no instante t , com média zero e variância σ_e^2 .

Encontrar os melhores parâmetros, que se ajustam melhor aos dados de uma série temporal, para o modelo ARIMA, pode ser difícil. Para facilitar esse problema, foi desenvolvida a metodologia Box-Jenkins (Box *et al.*, 2008). Essa metodologia é usada para encontrar o modelo mais adequado que será utilizado para a predição da série temporal, ou seja, ela consiste em identificar os parâmetros do modelo ARIMA(p,d,q) que melhor representem a série temporal desejada.

A metodologia Box-Jenkins é baseada em um ciclo iterativo, que se repete até ser encontrado um modelo satisfatório (não necessariamente o melhor) para uma série Z ,

por exemplo. Esse ciclo é executado em três etapas sequenciais: (1) identificação; (2) estimação; e (3) verificação (Box *et al.*, 2008; Morettin e de Castro Toloi, 2006).

A identificação (1) é a fase mais crítica do método, seu objetivo é determinar os valores dos parâmetros do modelo ARIMA. Inicialmente são feitas as diferenciações da série temporal Z , tantas vezes quantas necessárias, a fim de se obter uma série estacionária. O número de diferenças, d , necessárias para que o processo se torne estacionário, é alcançado quando a função de autocorrelação decresce rapidamente para zero (veja a Seção 2.3.3). Com a série estacionária, só resta identificar o processo ARMA(p,q), através da análise das autocorrelações e autocorrelações parciais estimadas.

A segunda fase, de estimação (2), é feita após identificado um modelo provisório para a série temporal. A estimação é a fase em que os parâmetros das variáveis preditoras são estimados, ou seja, o parâmetro ϕ da autorregressão e o parâmetro θ da média móvel. Também é estimado a variância do processo aleatório e_t , por meio do método de máxima verossimilhança.

A etapa final, do método Box-Jenkins, é a verificação (3), na qual verifica se o modelo estimado realmente representa adequadamente os dados. Geralmente são estimados parâmetros extras, para verificar entre eles qual é que melhor se ajusta à série temporal. A verificação é feita analisando os resíduos das predições, com testes de autocorrelação residual, tentando minimizar o *Aikake's Information Criterion* (AIC), ou algum outro critério de escolha (Hyndman e Athanasopoulos, 2013; Morettin e de Castro Toloi, 2006). Finalmente, após realizadas as três etapas anteriores, é analisado o resultado da verificação e, caso o modelo final escolhido seja satisfatório, então ele é escolhido, caso contrário, as três etapas são executadas novamente, usando um modelo diferente, até encontrar um modelo satisfatório.

2.6 Redes Neurais Artificiais

As Redes Neurais Artificiais (RNAs), usualmente conhecidas somente como Redes Neurais, são modelos matemáticos inspirados em modelos biológicos e no cérebro humano, tentando assimilar ao máximo suas funcionalidades e estrutura, para resolver problemas simples e complexos (Hornik e Leisch, 2000). O cérebro humano é um sistema de processamento de informação (computador) altamente complexo, ele processa informações de uma forma completamente diferente e muito mais rápida do que o computador digital convencional. Sua organização é constituída por neurônios, que trabalham de forma paralela e não linear para realizar certos processamentos, como por exemplo, reconhecimento de padrões, percepção e controle motor (Haykin, 1998).

As redes neurais oferecem as mesmas funcionalidades dos neurônios do cérebro humano para resolver problemas complexos, tais como a não linearidade, alto paralelismo, robustez, tolerância a erros e ruídos, adaptabilidade, aprendizado e generalização (Basheer e

Hajmeer, 2000; Cortez *et al.*, 2012; Haykin, 1998). As aplicações mais usuais das RNAs são em reconhecimento e classificação de padrões (reconhecimento de caracteres, de imagens e de voz), análise de imagens, agrupamento (*clustering*) de dados, previsão de séries temporais, otimização, aproximação de funções, processamento de sinais, e controle de processos.

As RNAs foram criadas baseadas na estrutura e comportamento do cérebro humano, no qual é composto por neurônios (nodos ou nós), que são estruturas de processamento simples, separadas em unidades fortemente conectadas e tem a capacidade de armazenar conhecimento experimental. A RNA se assemelha ao cérebro em dois aspectos, que são a parte computacional mais importante das redes neurais (Haykin, 1998):

- O conhecimento da rede neural é adquirido através de um processo de aprendizagem, também conhecido como algoritmo de aprendizagem ou algoritmo de treinamento, que é um processo adaptativo usado para aprender um conjunto de dados qualquer; e
- O conhecimento adquirido é armazenado pelas forças de conexão entre neurônios, conhecidas como pesos sinápticos.

Os neurônios artificiais são as unidades básicas das RNAs, cada conexão entre neurônios possui um peso sináptico, no qual armazena o conhecimento adquirido no processo de aprendizagem. Durante este processo, os pesos sinápticos são modificados de forma ordenada até que a rede neural reconheça e diferencie os dados de um conjunto finito. Após o aprendizado, a RNA está pronta para reconhecer os padrões dos dados passados no treinamento. Um ponto importante no processo de aprendizagem é a capacidade de generalização, ou seja, a rede neural é capaz de produzir uma saída desejada para valores de entrada que não foram utilizados durante o treinamento.

A abstração de um neurônio artificial em relação a um neurônio biológico é feita da seguinte forma conforme visto na Figura 2.12. A estrutura dos neurônios permite um conjunto de valores como entradas $(x_1, x_2, \dots, x_m)$ que produzem uma saída única (y_k) . As conexões do neurônio biológico são feitas pelos axônios e dendritos e as sinapses nervosas do neurônio k são representadas pelos pesos sinápticos $(w_{k0}, w_{k1}, \dots, w_{km})$. As entradas são ponderadas por seus respectivos pesos sinápticos $(w_{k1}, \dots, w_{km})$ e somadas ao valor de um bias b_k aplicado externamente, isso faz parte da junção aditiva.

Após a junção aditiva, a função de ativação φ (ou função restritiva) é usada para limitar o intervalo do sinal de saída. Normalmente, o intervalo normalizado da saída de um neurônio é descrito como um intervalo fechado, sendo $[0, 1]$ e $[-1, 1]$ os mais comuns (Basheer e Hajmeer, 2000; Haykin, 1998). Esse comportamento é apresentado de forma mais simplificada, em um diagrama de blocos, na Figura 2.13.

Em termos matemáticos, um neurônio k é descrito como o seguinte par de equações

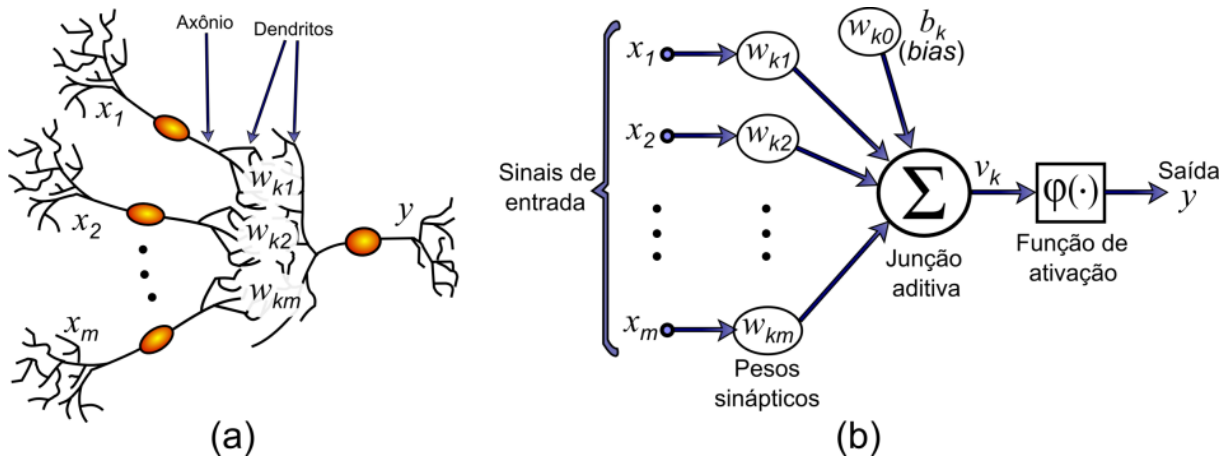


Figura 2.12: Analogia entre o sinal propagado em neurônios biológicos (a) e neurônios artificiais (b). Fonte: Adaptado de Haykin (1998).

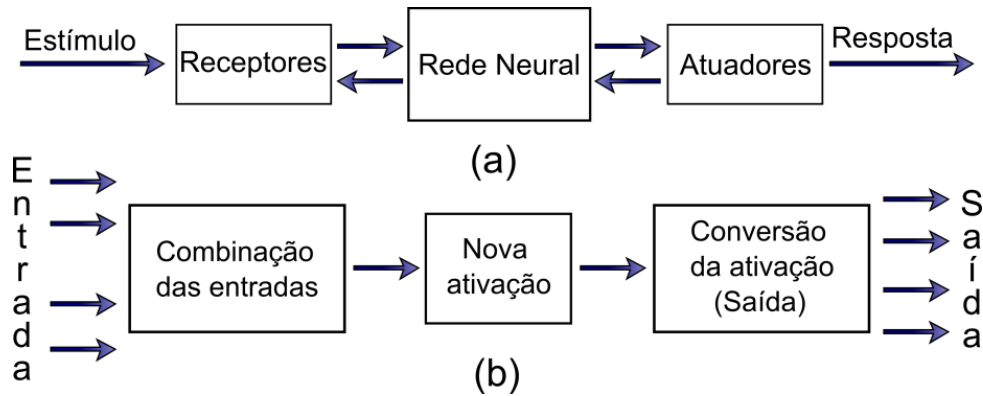


Figura 2.13: Comparação, usando um diagrama em blocos, do sistema nervoso humano (a) com o funcionamento de uma rede neural artificial (b). Fonte: Adaptado de Haykin (1998).

(Haykin, 1998):

$$v_k = \sum_{i=0}^m w_{ki} x_i \quad (2.20)$$

$$y_k = \varphi(v_k + b_k) \quad (2.21)$$

onde v_k , ou potencial de ativação, é a saída da junção aditiva; x_i são os sinais de entrada; w_{ki} são os pesos sinápticos do neurônio k , sendo que i é o número de conexões que este neurônio faz com outros; b_k é o bias; sigma é a função de ativação; e y_k é o sinal de saída do neurônio. O bias b_k é um parâmetro externo do neurônio artificial k , ele tem o efeito de aumentar ou diminuir a entrada líquida da função de ativação, dependendo se ele é positivo ou negativo.

A função de ativação, representada por φ , pode representar características lineares ou não lineares, determinando a saída de um neurônio a partir do seu potencial de ativação. Dentre as funções lineares, destacam-se a função linear por partes e a função de limiar (ou

função de Heaviside). As funções não lineares são as que mais se assemelham ao neurônio biológico, sendo as mais comuns a função sigmoide e a função tangente hiperbólica (*tanh*). A Figura 2.14 ilustra as funções de ativação, juntamente com os tipos de gráficos que cada uma representa.

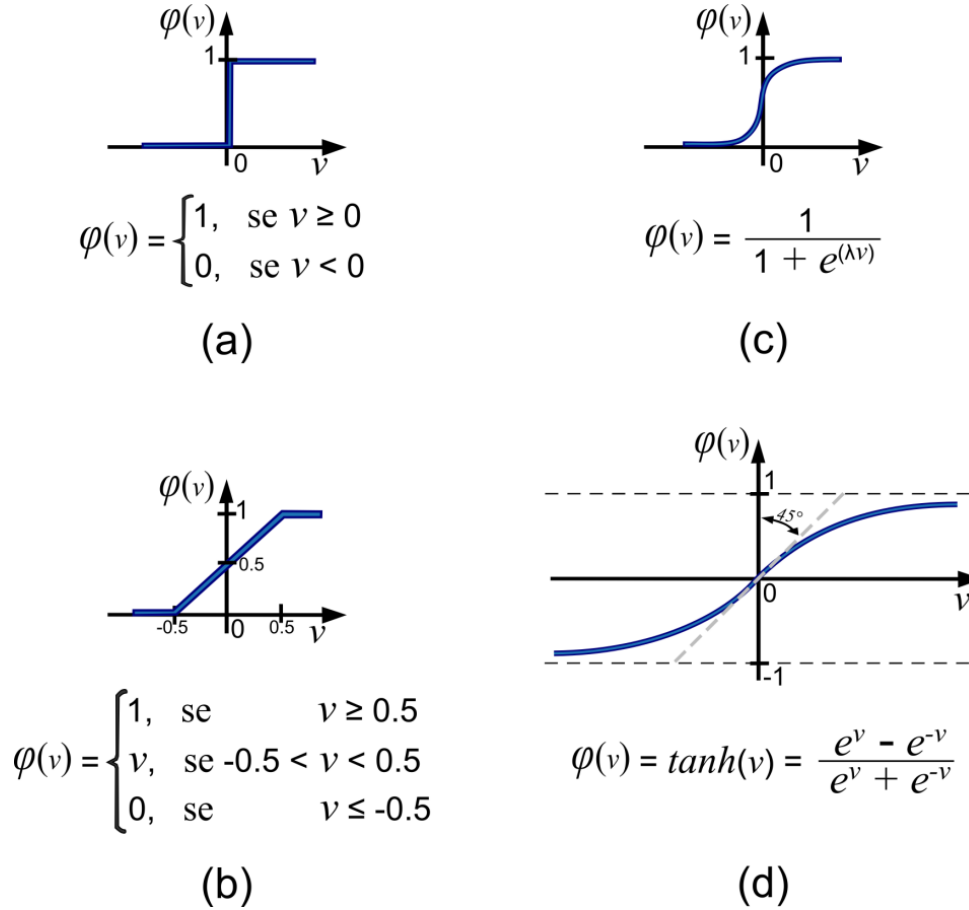


Figura 2.14: Tipos de função de ativação, com (a) função de limiar; (b) função linear por partes; (c) função sigmoide; e (d) função tangente hiperbólica. Fonte: Adaptado de Haykin (1998).

Como já citado as RNAs são bastante utilizadas em estudos de previsão de séries temporais, isso também gerou vários estudos de predição de tráfego usando diferentes tipos e arquiteturas de redes neurais. Existem várias pesquisas com redes neurais diretas, tais como as MLPs (Cortez *et al.*, 2012; Hosseini *et al.*, 2012). Porém, muitos estudos visam as RNNs (Hallas e Dorffner, 1998), pois elas possuem um ciclo interno de memória que facilita o aprendizado de comportamentos dinâmicos, temporais e sequenciais. Consequentemente, a RNN é um bom modelo para o aprendizado de séries temporais.

Uma vantagem de qualquer tipo de RNA é no tempo de resposta, isto é, o quão rápido é feito a predição dos valores futuros. O processo de aprendizagem é a etapa mais lenta no uso das RNAs. Contudo, após a aprendizagem a rede neural está pronta para ser usada, calculando os resultados da previsão de modo muito rápido, comparado a outros modelos de predição, como o ARIMA (Cortez *et al.*, 2012; Feng e Shu, 2005). Portanto,

as RNAs são boas para predição em tempo-real e curto prazo, conseguindo um resultado satisfatório em relação a precisão da predição e em um pequeno tempo de resposta (Cortez *et al.*, 2012).

2.6.1 Arquiteturas e Topologias das Redes Neurais Artificiais

As RNAs podem ser vistas como grafos orientados. Os neurônios são representados como vértices e as arestas representam as sinapses. A direção das arestas indica a orientação da alimentação dos pesos sinápticos de um neurônio para outro.

A topologia da rede neural define o grafo dela, ou seja, a estrutura do arranjo dos neurônios e como eles estão conectados entre si. A arquitetura de uma RNA é algo mais genérico, além da topologia, a arquitetura também define o padrão das conexões sinápticas, o algoritmo de aprendizagem, a função de ativação e outros parâmetros que influenciam no funcionamento da RNA. As diferentes arquiteturas das RNAs são formas abstratas de diferentes organizações de neurônios biológicos (Haykin, 1998).

A estrutura da rede neural é composta pela interconexão dos neurônios e pode variar em relação ao número de camadas, número de neurônios em cada camada, função de ativação dos neurônios em uma camada e a forma como as camadas são conectadas. Alguns exemplos mais comuns de arquiteturas de redes neurais são: Redes Neurais Diretas, de Camada Simples ou de Múltiplas Camadas; Redes Neurais Recorrentes; Redes Neurais Reticuladas (Haykin, 1998; Hornik e Leisch, 2000).

2.6.1.1 Redes Neurais Diretas de Camada Simples

As Redes Neurais Diretas de Camada Simples (*Feed-forward* ou Alimentadas Adiante) são um tipo de arquitetura que possui somente a camada de entrada e a camada de saída. A primeira é relativa aos neurônios sensoriais, que recebem os estímulos dos exemplos de entrada, e a segunda é relacionada aos neurônios processadores, que processam e transmitem o estímulo da entrada para a saída. Esse arranjo de camadas é o mais simples, a camada de entrada recebe os sinais de entrada e a camada de saída obtém as saídas. O número de neurônios de cada camada é independente e não precisa ser o mesmo para cada uma delas, esse número depende da aplicação da rede neural, variando para cada tipo de problema (de Villiers e Barnard, 1993; Hornik *et al.*, 1989).

Neste tipo de arquitetura, o fluxo de informações (*feed* ou alimento) da RNA, flui da camada de entrada para a camada de saída, como mostra a Figura 2.15. Por este motivo, elas são chamadas de Redes Diretas, *Feed-forward Networks* ou Redes Alimentadas Adiante. Esse tipo de RNA é bastante usada em classificação de padrões e filtragem linear. Os principais exemplos de uso são das redes Perceptrons e Adalines. Os algoritmos de aprendizagem principais usados nesse tipo de rede são a Regra de Hebb (para Perceptrons) e Regra Delta (para Adalines) (Haykin, 1998).

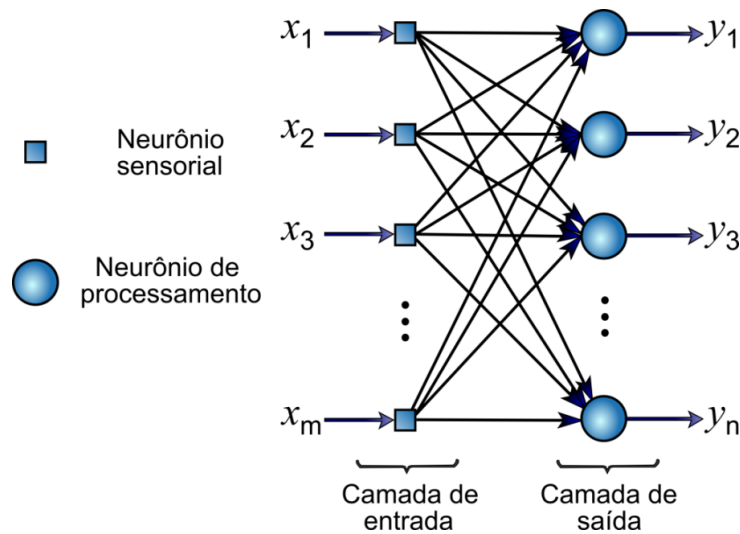


Figura 2.15: Exemplo da arquitetura de uma Rede Direta de Camada Simples, com m neurônios da camada de entrada e n neurônios na camada de saída, mostrando o fluxo de informações a partir da camada de entrada para a camada de saída. Fonte: Adaptado de Haykin (1998).

2.6.1.2 Redes Neurais Diretas de Múltiplas Camadas

As Redes Neurais Diretas de Múltiplas Camadas são uma evolução das RNAs com camada simples. Esse tipo de arquitetura, além das camadas de entrada e de saída, possuem camadas ocultas (intermediárias ou escondidas) de neurônios processadores. Como ela também é direta, o fluxo de informações flui no sentido da entrada para a saída. Entretanto, o número de neurônios e de camadas intermediárias depende do tipo e complexidade do problema, podendo ser somente uma camada intermediária (de Villiers e Barnard, 1993; Hornik *et al.*, 1989), ou várias (Hinton *et al.*, 2006).

Assim como nas Redes Neurais Diretas de Camada Simples, a informação provém da camada de entrada e é transmitida para as camadas seguintes, até então, chegar na camada de saída. A conexão dos neurônios de uma camada é feita somente com os neurônios da camada seguinte. Os neurônios, portanto, não fazem conexão com outros neurônios da mesma camada. Além disso, cada neurônio de uma camada é conectado a todos os neurônios da camada seguinte, como é ilustrado na Figura 2.16.

A aplicação dessa rede neural é mais utilizada na aproximação de funções não lineares, classificação de padrões, identificação de sistemas, otimização, robótica, controle de processos e filtragem não linear. Os principais exemplos são as Perceptrons de Múltiplas Camadas (*Multilayer Perceptrons* — MLP) e Redes de Base Radial (*Radial Basis Function* — RBF). Os algoritmos de aprendizagem geralmente seguem a Regra Delta Generalizada ou Backpropagation (para as MLP) e Regra Delta Competitiva (para as RBF) (Haykin, 1998).

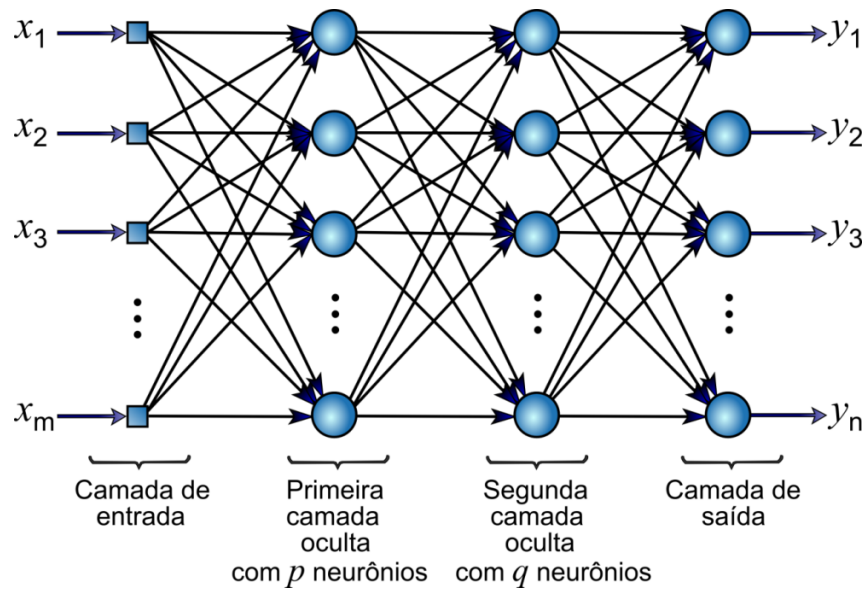


Figura 2.16: Exemplo da arquitetura de uma Rede Direta de Múltiplas Camadas, cujo fluxo de informações passa da camada de entrada, para as camadas ocultas e, posteriormente, para a camada de saída. A rede neural apresentada contém m e n neurônios nas camadas de entrada e de saída, respectivamente, além de p e q neurônios nas camadas ocultas. Fonte: Adaptado de Haykin (1998).

2.6.1.3 Redes Neurais Recorrentes

As Redes Neurais Recorrentes (*Recurrent Neural Networks* — RNN), também conhecidas como Redes Neurais Realimentadas (*Feed-backward*), são RNAs que possuem uma ou mais conexões entre neurônios que formam um ciclo. Esta é a principal diferença entre esse tipo de rede com a Rede Direta, que não pode haver laços de realimentação. A realimentação é quando a saída de um neurônio é aplicada na entrada de outros neurônios de camadas anteriores. Quando a saída de um nó é aplicada na entrada do mesmo nó é chamado de autorrealimentação (Haykin, 1998).

Os ciclos da RNN são responsáveis por armazenar e transmitir as experiências de um neurônio para outro, criando uma memória interna que facilita o aprendizado de dados sequenciais (Hallas e Dorffner, 1998; Haykin, 1998). Esses ciclos podem ser usados em qualquer lugar da rede neural e em qualquer direção e sentido, e.g., ligando a camada de saída com a de entrada, ligando uma camada oculta com outra camada qualquer, ou qualquer combinação de ligações cíclicas (Haykin, 1998). Dois exemplos com RNN são apresentados na Figura 2.17.

A retroalimentação possibilita um processamento dinâmico de informações, o que a torna recomendável em aplicações de problemas variantes com o tempo (como previsão de series temporais), controle de processos e entre outros. Os principais exemplos desse tipo de rede é a Rede de Hopfield e a MLP com retroalimentação. Os algoritmos de aprendizagem são a minimização de funções de energia (para a Rede de Hopfield) e Regra Delta com *Backpropagation* (para a MLP com retroalimentação) (Haykin, 1998).

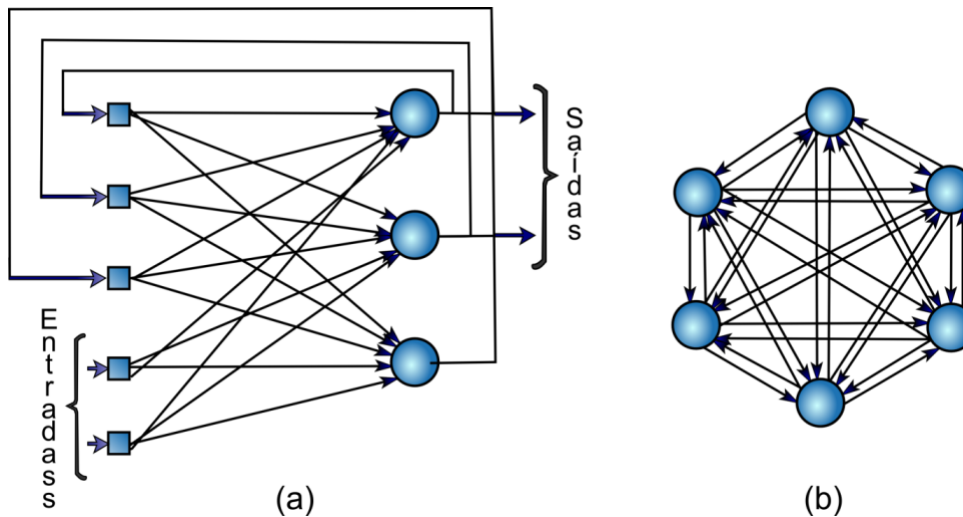


Figura 2.17: Exemplos de arquitetura de Redes Neurais Recorrentes: (a) Rede Neural Recorrente com ciclos da saída para a entrada; e (b) Rede de Hopfield. Fonte: Adaptado de Haykin (1998).

2.6.1.4 Redes Neurais Reticuladas

As Redes Neurais Reticuladas (ou Competitivas) são redes que fazem as unidades de saídas competirem entre si, onde somente uma unidade de saída pode estar ativa a cada momento. Os neurônios deste tipo de rede recebem informações idênticas na entrada, mas competem entre si para definir qual será ativo. Neste tipo de rede a localização dos neurônios está relacionada com o processo de ajuste de pesos, ou seja, a organização e posicionamento dos neurônios influencia nos ajustes dos pesos.

As estruturas reticuladas consideram a disposição espacial dos neurônio, visando propósitos de extração de características. Esse tipo de rede é usado especialmente em problemas de agrupamentos (*clustering*). O principal exemplo, exibido na Figura 2.18, é a Rede de Kohonen, que usa algoritmo de aprendizagem competitivo (Haykin, 1998).

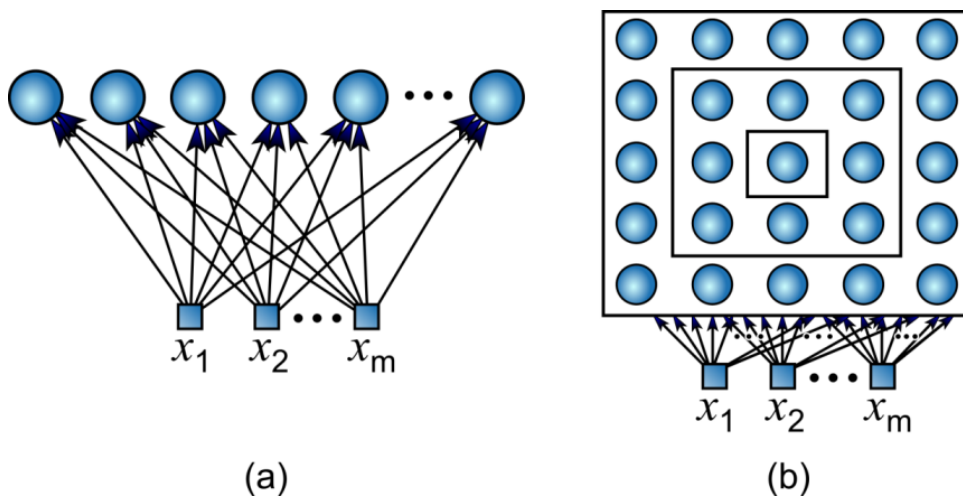


Figura 2.18: Exemplos de Redes de Kohonem com (a) Arranjo Unidimensional; e (b) Arranjo Bidimensional. Fonte: Adaptado de Haykin (1998).

2.6.2 Processos de Aprendizagem de Redes Neurais Artificiais

Um fator importante de uma rede neural é a habilidade de aprender conforme o ambiente em que ela está inserida e, assim, melhorar o seu desempenho. Antes de ser utilizada, uma RNA deve passar por uma fase de aprendizado, na qual a rede neural irá receber estímulos para conhecer o ambiente em que esteja inserida, isto através de um processo de ajustes de pesos sinápticos e de níveis de viés (*bias*) (Haykin, 1998).

Durante o processo de aprendizagem, também conhecido como treinamento, devem ser passados conjuntos de exemplos de entrada para a RNA. Os exemplos devem ser passados de forma sequencial e iterativa, e são nessas iterações que os pesos sinápticos são ajustados de forma coerente com o exemplo de entrada. Esse processo iterativo se repete até que a rede neural alcance um resultado satisfatório, isto é, até obter-se um erro final mínimo aceitável para a solução do problema. Uma iteração desse processo é chamada de época (*epoch*) ou período de treinamento. Outra condição de parada, para o processo de aprendizagem, aciona-se quando é atingido um número limite de épocas de treinamento, ou seja, é dado um limite máximo de épocas iterativas de treinamento, quando o número de iterações passa esse limite máximo, então o processo de aprendizagem para, mesmo se ainda não atingiu o erro mínimo esperado.

Existem dois pontos importantes em relação ao processo de aprendizagem: o algoritmo de aprendizagem e o paradigma de aprendizagem (Haykin, 1998). Existem vários tipos de algoritmos de aprendizagem e são eles que definem as regras para o processo de aprendizagem, decidindo a forma que os pesos sinápticos serão tratados, a propagação do sinal do estímulo, o ajuste dos pesos sinápticos, como os neurônios se comportam e entre outros. O algoritmo de aprendizagem também varia de acordo com o tipo de aplicação da rede neural, por exemplo, existem algoritmos que são próprios ou mais adequados para associação e reconhecimento de padrões, aproximação de funções, controle e filtragem, otimização, agrupamento e outros. O paradigma de aprendizagem define como a RNA se adapta às informações recebidas. Os paradigmas de aprendizagem são: aprendizado supervisionado e aprendizado não supervisionado.

2.6.2.1 Aprendizado Supervisionado

Aprendizado supervisionado é quando utiliza-se um agente externo como “professor” (supervisor), que indica à rede neural a resposta desejada para o padrão de entrada. O professor detém o conhecimento do ambiente, porém a rede neural desconhece. O aprendizado é baseado em um conjunto de exemplos de estímulos e suas respostas, que representam o comportamento que deve ser apresentado pela rede neural (Haykin, 1998).

No treinamento supervisionado, a RNA é exposta a um vetor de treino extraído do ambiente e o professor indica para a rede a resposta desejada para este vetor de treino (Figura 2.19). Os pesos da rede são ajustados iterativamente através da influência do

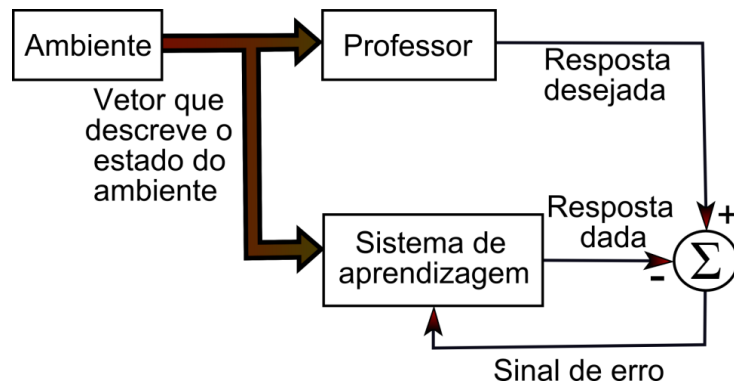


Figura 2.19: Diagrama representado o funcionamento do Aprendizado Supervisionado, que recebe, do professor a resposta desejada referente ao valor de entrada passado para o sistema de aprendizagem. Fonte: Adaptado de Haykin (1998).

vetor de treino e do sinal de erro. O sinal de erro é definido como a diferença entre a resposta desejada e a resposta efetivamente obtida da rede. Esse ajuste é executado de forma iterativa, de forma que o conhecimento do professor é transferido para a rede neural através do treinamento. Quando a rede não receber mais conhecimento, ou seja os pesos sinápticos permanecem o mesmos depois de um passo iterativo, o professor é dispensado e a rede estará treinada e pronta para lidar com o ambiente inteiramente por si mesma (Haykin, 1998).

No aprendizado supervisionado o professor é responsável por rotular os dados de entrada da RNA. Dados rotulados são dados que recebem um rótulo (marcação, nome ou classe) específico, que de alguma forma represente o significado desse dado. De forma resumida, os dados rotulados fazem parte de um vetor bidimensional de características e rótulos, que armazena as informações desses dados. Transpondo para o plano cartesiano, os dados rotulados são representados pelo par ordenado (característica, rótulo), no qual o eixo das abscissas fica responsável por armazenar as características (ou valores numéricos que as representem) e o eixo das ordenadas armazena os rótulos das características (ou um valor numérico que identifique a característica). No caso dos dados não rotulados, o eixo das ordenadas é inexistente.

Um exemplo de dados rotulados é o conjunto dos pares (*Animal*, *Classe*), no qual a característica é o *Animal* e o rótulo é a *Classe*, um exemplo disso seria o conjunto {(Galinha, Ave), (Cavalo, Mamífero), (Sapo, Anfíbio), (Leão, Mamífero), (Avestruz, Ave), (Jacaré, Réptil), (Urso, Mamífero)}, que também pode ser representado na forma numérica {(1, 1), (2, 2), (3, 3), (4, 2), (5, 1), (6, 4), (7, 2)}, onde cada característica e rótulo recebem um valor para representá-los de forma independente. Um exemplo de dados não rotulados são fotografias, nas quais não possuem um rótulo específico, ou seja, seus significados podem variar bastante, independentemente da aplicação.

2.6.2.2 Aprendizado Não Supervisionado

No aprendizado não supervisionado (ou auto-organizado) não existe um agente externo indicando a resposta desejada para os padrões de entrada, ou seja, não há um professor para supervisionar o processo de aprendizagem, este é autônomo. Por isso, não são necessários exemplos de dados rotulados para treinar a rede neural (Haykin, 1998). Nesse caso, a RNA organiza os valores de entrada com base em propriedades estatísticas, estabelecendo uma classificação dos dados de entrada.

Nessa forma de treinamento, somente padrões de entrada são apresentados a rede até que ela se torne sensibilizada às regularidades estatísticas e correlações dos dados de entrada. Daí, a rede desenvolve a habilidade de formar representações internas para codificar características da entrada e, por meio disto, agrupar os padrões de entrada em grupos com características similares ou criar novos grupos automaticamente. Nesse tipo de aprendizagem, o sistema de aprendizagem utiliza somente de suas funções internas para encontrar um padrão nos dados de entrada, sem estímulos de agentes externos, como mostra a Figura 2.20.

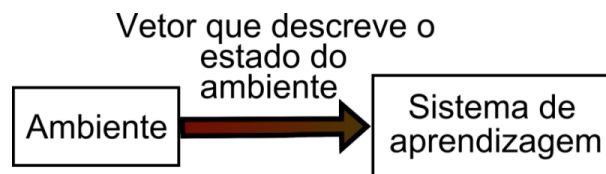


Figura 2.20: Diagrama representando a Aprendizagem Não Supervisionada, que somente recebe os exemplos de entrada do ambiente, mas não possui agentes externos que influenciam no aprendizado. Fonte: Adaptado de Haykin (1998).

2.6.2.3 Aprendizado por Reforço

Aprendizado por reforço pode ser considerado um paradigma intermediário entre o aprendizado supervisionado e o não supervisionado. No aprendizado por reforço não existe um professor para supervisionar o treinamento, porém, há um crítico externo que avalia a resposta fornecida pela RNA. O comportamento da rede é avaliado com base em algum critério numérico, que é fornecido em instantes espaçados de tempo, reforçando positivamente ou negativamente o aprendizado até o momento.

Na aprendizagem por reforço, exibido na Figura 2.21, o crítico converte um sinal de reforço primário, recebido do ambiente, em um sinal de reforço heurístico (de melhor qualidade) (Haykin, 1998). O sistema aprende por um reforço atrasado, isto é, o sistema observa uma sequência temporal de estímulos recebidos pelo ambiente, que eventualmente resultam na geração do sinal de reforço heurístico. Nessa aprendizagem, ações são tomadas ao longo de uma sequência de passos, essas ações determinam o comportamento global do sistema. A função do sistema de aprendizagem é descobrir estas ações e realimentá-las para o ambiente. Esse tipo de aprendizado é o mais complicado de se realizar, entretanto,

ele provê as bases para o sistema interagir com o ambiente e desenvolver a habilidade de aprender a desempenhar uma tarefa, somente com base nas saídas da sua própria experiência, resultante da interação com o ambiente (Haykin, 1998).

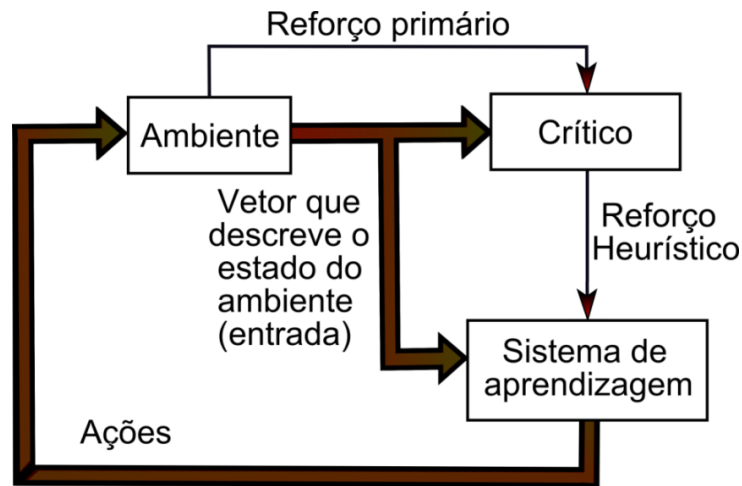


Figura 2.21: Diagrama representando o Aprendizado por Reforço. Fonte: Adaptado de Haykin (1998).

2.6.3 Redes Neurais Artificiais com Aprendizagem Profunda

Uma Rede Neural Artificial com Aprendizagem Profunda (RNAP) é uma RNA que possui muitas camadas (pelo menos 4 camadas de neurônios, em uma MLP, por exemplo) (Bengio, 2009; de Villiers e Barnard, 1993) e por isso, ela necessita de um algoritmo de aprendizagem que leve em conta essa profundidade. A Aprendizagem Profunda (*Deep Learning*) se refere a um método de aprendizado de máquina que usa uma RNA para aprender características hierárquicas, separadas em vários níveis de representação dos dados, ou seja, cada camada de uma rede neural aprende um nível de representação das características dos dados.

Os dados de diferentes níveis da RNA são hierarquizados pelas suas próprias características, fatores ou conceitos. Os níveis mais altos (*high-levels*) da representação, são definidos pelos níveis mais baixos (*low-levels*), onde as representações dos níveis mais baixos podem definir vários conceitos diferentes nos níveis mais altos. Quanto mais alto o nível, mais abstrata e não linear será a representação dos dados (Bengio, 2009; Hinton *et al.*, 2006).

Os seres humanos, quando vão reconhecer uma imagem, por exemplo, utilizam sua intuição sobre como decompor o problema em subproblemas (múltiplos níveis de representação). Então eles transferem conhecimentos de aprendizados anteriores para generalizar e aprender novos exemplos, ou seja, descobrir o que a imagem representa (Bengio, 2009). A aprendizagem profunda de uma RNA é baseada nessa ideia de decomposição do problema em subproblemas, assim como o cérebro humano faz para evoluir seu aprendizado.

Desse modo, os algoritmos de aprendizagem profunda, baseiam-se em representações distribuídas para cada dado que deve ser aprendido pela RNA. Esses algoritmos assumem que a distribuição das características é feita pela interação entre vários fatores, que são disseminados em vários níveis. Cada nível corresponde a diferentes graus de abstração, sendo que os níveis mais altos compreendem características mais complexas e são obtidos pela generalização dos níveis mais baixos.

O algoritmo de aprendizagem profunda deve descobrir as abstrações dos dados de entrada de forma automática. As abstrações são separadas entre as camadas da RNA, que representam os níveis hierárquicos das características dos dados. O número de níveis e a estrutura de relacionamento deve ser descoberta pelo algoritmo de aprendizagem profunda, através dos exemplos analisados. De forma ideal, os algoritmos de aprendizagem devem aprender com o mínimo possível de ajuda humana, ou seja, sem ter que definir as abstrações manualmente e criar um grande conjunto de exemplos para o algoritmo (Bengio, 2009; Hinton *et al.*, 2006).

Para um algoritmo baseado em aprendizagem profunda reconhecer, por exemplo, imagens de rostos humanos, ele começa analisando os pixels da imagem para extrair as características mais básicas. Em seguida, o algoritmo transfere o conhecimento obtido para um nível acima, para extrair novos dados mais genéricos e complexos (Bengio, 2009). A Figura 2.22 mostra um exemplo de níveis de características que podem ser encontrados em um algoritmo baseado em aprendizagem profunda, sendo eles:

- **1º Nível:** primeiro o algoritmo extrai características (nível mais baixo) que são invariantes a pequenas diferenças geométricas. Com isso ele detecta bordas, pequenas retas, pequenas curvas e pontos;
- **2º Nível:** em seguida, ele transforma os dados encontrados no nível anterior e os generalizam. Assim, ele encontrará formas mais complexas que reúnem várias características encontradas no nível anterior. Com isso, ele detecta a combinação dessas formas, como olhos, boca, nariz, orelha e assim por diante;
- **3º Nível:** abstraindo os dados da camada anterior, o algoritmo encontra as características invariantes ao posicionamento dos dados do nível anterior. Portanto, ele encontra os rostos, posicionamento das orelhas ao lado do rosto, nariz ao centro, olhos em cima do nariz e boca em baixo do nariz.

2.6.3.1 Algoritmos de Aprendizagem Profunda

A maioria dos algoritmos de aprendizado das RNAs são próprios para arquiteturas rasas (*shallow architectures*), com 1, 2 ou, no máximo, 3 níveis no total (incluindo camadas de entrada e de saída) (Bengio, 2009; Erhan *et al.*, 2009). Esses algoritmos também funcionam com arquiteturas mais profundas, porém, o tempo de treinamento dos algoritmos

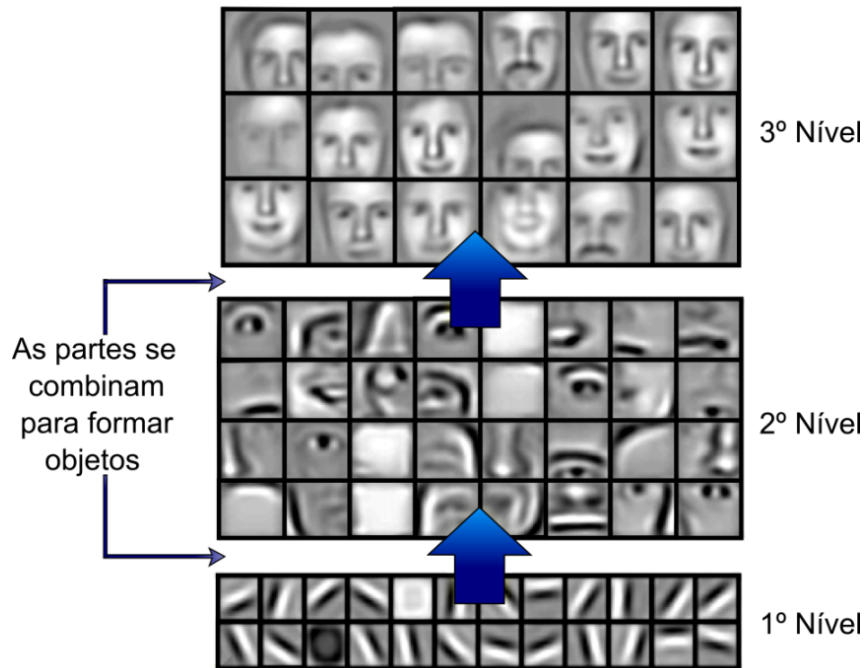


Figura 2.22: Exemplo dos níveis de representação aprendidos ao usar a aprendizagem profunda em um problema de reconhecimento de imagens com rostos humanos. Fonte: Adaptado de Bengio (2009).

convencionais, usados em RNAPs, acaba sendo muito grande e ainda produz resultados ruins, comparados às RNAs rasas.

A profundidade da arquitetura se refere ao número de camadas da rede neural, e teoricamente, quanto mais camadas maior será o nível de composições de operações não lineares aprendidas (Bengio, 2009; Hinton *et al.*, 2006). Inspirado pela arquitetura de um cérebro humano, os pesquisadores que usavam redes neurais sempre tentaram treinar redes profundas de múltiplos níveis. Entretanto, até o ano de 2006, poucas tentativas obtiveram sucesso efetivo ao treinar arquiteturas profundas. Até então, os pesquisadores conseguiam bons resultados principalmente com arquiteturas de dois ou três níveis hierárquicos.

Em 2006, Hinton e outros pesquisadores da Universidade de Toronto introduziram as *Deep Belief Networks* (DBNs) (Hinton *et al.*, 2006), que possui um algoritmo guloso de aprendizagem, no qual treina uma camada da rede neural de cada vez, explorando um algoritmo de aprendizado não supervisionado para cada camada. As DBNs, são redes neurais diretas compostas de várias camadas ocultas. Ela contém uma camada de entrada, com neurônios de entrada (chamados de unidades visíveis), um número N de camadas ocultas e, por fim, uma camada de saída. Os pesos das conexões são dados por w_j , que é a matriz de pesos entre os neurônios da camada $j - 1$ e j , e b_j é o bias da camada j (Figura 2.23). A camada de unidades visíveis representam os dados de entrada, uma camada oculta aprende a representar as características que correlacionam os dados das camadas hierárquicas anteriores.

O processo de aprendizagem das DBNs é realizado através de um algoritmo de trei-

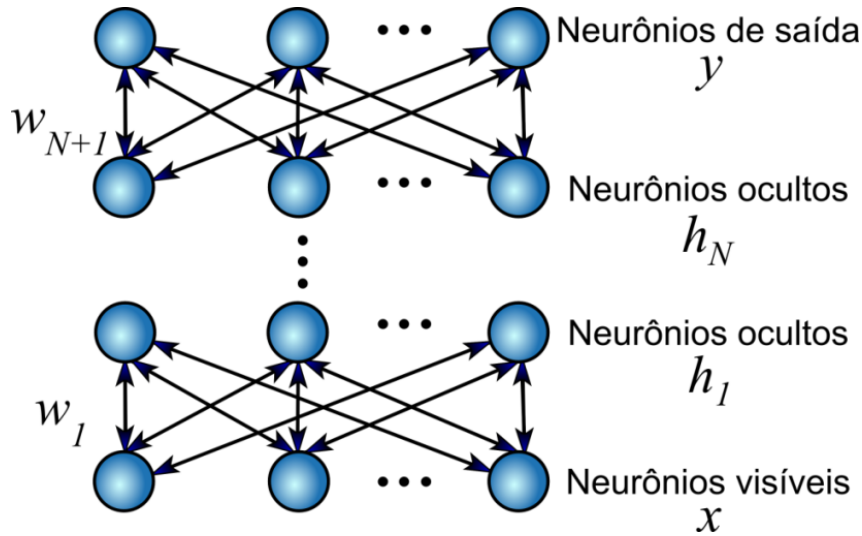


Figura 2.23: Arquitetura de uma DBN com N camadas e seus neurônios ocultos, uma camada de entrada (neurônios visíveis) e uma camada de saída (neurônios de saída). Fonte: Adaptado de Bengio (2009).

namento guloso e não supervisionado, que opera em uma camada por vez (Hinton *et al.*, 2006). Esse algoritmo é baseado no treinamento de uma sequência de *Restricted Boltzmann Machines* (RBMs). Uma RBM é uma rede neural com duas camadas, na qual neurônios de entrada, que são estocásticos e binários, são conectados com neurônios de saída, também estocásticos e binários, tudo isso com conexões simetricamente ponderadas (Chao *et al.*, 2011).

A camada de entrada da RBM é feita de unidades visíveis, já a segunda é feita de unidades ocultas. Não há conexões entre unidades da mesma camada. Com isso, as unidades ocultas são condicionalmente independentes. Um ponto importante da RBM é que seus neurônios possuem conexões bidirecionais (Figura 2.24) (Hinton *et al.*, 2006).

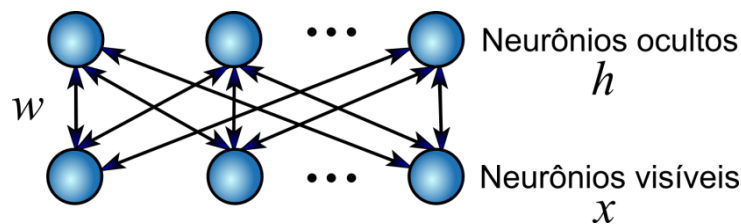


Figura 2.24: Arquitetura de uma RBM, com uma camada de entrada (neurônios visíveis) e uma camada oculta (neurônios ocultos). Fonte: Adaptado de Bengio (2009).

A DBN é construída durante o seu treinamento, no qual se baseia em treinar várias RBMs e combiná-las para formar uma DBN. O algoritmo de aprendizagem é aplicado sequencialmente, treinando uma RBM por vez, na qual aprende os dados em uma camada oculta. Após aprender uma camada oculta, os pesos dos neurônios das camadas ocultas podem ser tratados como novos dados para treinar outra RBM, ou seja, a nova camada depende do aprendizado da camada anterior. E isso pode ser repetido para aprender quantas camadas ocultas forem necessárias, sendo essa a base do algoritmo guloso, que

prioriza uma camada por vez (Bengio, 2009; Hinton *et al.*, 2006).

Após o aprendizado da DBN, os valores das variáveis em cada camada podem ser inferidos por um algoritmo, começando com os dados da camada mais baixa (camada de entrada) e usando os pesos gerados para obter um resultado final na camada mais alta (camada de saída). Usando esse método que treina uma rede neural através de várias camadas inicialmente independentes, o resultado pode ser visto como uma única rede de múltiplas camadas. Aprender uma camada oculta por vez é um modo muito efetivo de aprendizagem para redes neurais profundas, com várias camadas ocultas e milhões de pesos (Bengio, 2009).

O algoritmo guloso é eficiente, pois ele combina um conjunto de modelos mais simples, que aprendem de forma sequencial, para finalmente, aprender um modelo mais complexo. Para forçar cada camada da sequência a aprender algo diferente das camadas anteriores, os dados são modificados após a aprendizagem de cada camada. A ideia do algoritmo guloso é permitir que cada camada na sequência receba uma representação diferente dos dados. A camada executa uma transformação não linear nos vetores de entrada e produz como saída os vetores que serão usados como entrada para a próxima camada na sequência (Bengio, 2009).

O algoritmo guloso de treinamento de uma DBN é separado em três passos sequenciais:

1. Construir e treinar uma RBM com a camada de entrada x e uma camada oculta h ;
2. Empilhe outra camada oculta em cima da RBM, criada anteriormente, para criar uma nova RBM, formada a partir da camada oculta da RBM anterior mais a nova camada oculta da RBM atual.
3. Continue empilhando camadas no topo da DBN, treinando assim como no passo anterior. O resultado final será uma DBN como mostra a Figura 2.23.

2.6.4 Modelos de Redes Neurais Artificiais

Diferentes modelos de redes neurais utilizam arquiteturas diferentes para a RNA, i.e., modelos diferentes possuem organizações diferentes das camadas da rede neural e dos neurônios de cada camada. Além disso, os modelos de RNAs utilizam diferentes processos de aprendizagem. A seguir, são apresentados os modelos de redes neurais artificiais mais comuns nas pesquisas científicas. São descritos tanto os modelos mais simples quanto os modelos mais complexos, que são os modelos de redes neurais com aprendizagem profunda.

2.6.4.1 *Multilayer Perceptron e Treinamento Backpropagation*

As redes neurais MLP são uma das arquiteturas mais comuns na literatura. Esse tipo de rede neural possui uma camada de entrada, uma ou mais camadas ocultas e uma camada de saída. Boas práticas de uso sugerem somente uma ou duas camadas ocultas

(de Villiers e Barnard, 1993). Isso se dá pelo fato de que os mesmos resultados podem ser obtidos aumentando o número de neurônios na camada oculta, em vez de aumentar o número de camadas escondidas (Hornik *et al.*, 1989).

As redes MLPs são redes neurais diretas, nas quais todos os neurônios de uma camada são conectados com todos os neurônios da camada seguinte, mas não há conexão entre os neurônios da mesma camada. Esse tipo de rede neural é chamada de direta (ou alimentada adiante) devido ao fluxo de informação, que percorre da camada de entrada para a camada de saída. O algoritmo de treinamento mais comum da MLP é o *Backpropagation*, que é um algoritmo de treinamento supervisionado, no qual a MLP aprende a saída desejada através de vários dados de entrada (Basheer e Hajmeer, 2000).

As etapas do algoritmo *Backpropagation* são apresentadas no Algoritmo 1 (Basheer e Hajmeer, 2000). O processo de atribuição de valores dos pesos (**linha 2**) é comumente feito com valores aleatórios pertencentes no intervalo $[0,1]$. Como ele é um algoritmo de treinamento supervisionado, as entradas da rede devem ser apresentadas juntamente com a saída desejada (**linha 4**). Nas **linhas 5 e 6**, o cálculo dos valores de saída são realizados com a função de ativação. O erro calculado na **linha 7** é a diferença da saída desejada pela resposta encontrada pela rede nesta iteração. O ajuste dos pesos (**linha 8**) utiliza a taxa de aprendizagem como atualização dos erros. Finalmente, na **linha 9**, é verificado a condição de parada, que pode variar dependendo da aplicação.

Algoritmo 1 Treinamento Backpropagation

```
1: procedure BACKPROPAGATION
2:   Inicialização aleatória dos pesos sinápticos;
3:   repeat
4:     Atribuição dos padrões de entrada e suas saídas esperadas;
5:     Cálculo dos valores de saída dos neurônios ocultos;
6:     Cálculo da resposta da rede (valores dos neurônios de saída);
7:     Cálculo do erro (diferença da resposta da rede pela saída esperada);
8:     Ajuste dos pesos sinápticos do neurônios da rede neural;
9:   until Condição de parada não satisfeita;
10: end procedure
```

Geralmente o *Backpropagation* sofre um problema na magnitude da derivativa parcial, tornando-se muito alta ou muito pequena. Isso é um problema onde o processo de aprendizagem sofre muitas flutuações e instabilidades, deixando a convergência do treinamento mais lenta ou inexistente, fazendo com que a rede neural fique presa em um mínimo local. Para ajudar a evitar esse problema, foi criado o algoritmo *Resilient Backpropagation* (RPROP), que possui uma taxa de aprendizagem dinâmica. Esta, é atualizada para cada conexão dos neurônios, para diminuir o erro de forma independente para cada neurônio.

2.6.4.2 *Jordan Neural Network*

A JNN é uma rede neural recorrente (RNN) simples. As RNNs são chamadas de recorrentes pois possuem, pelo menos, uma conexão entre neurônios que formam um ciclo. Os ciclos da RNN são responsáveis de armazenar e transmitir experiências passadas para aprendizados recentes.

Devido aos ciclos, é criada uma memória interna que facilita a aprendizagem de dados sequenciais (Hallas e Dorffner, 1998; Haykin, 1998), tais como as séries temporais e o tráfego de Internet. Os ciclos, que também são chamados de laços de realimentação ou retroalimentação, podem ser usados em qualquer camada e neurônio da rede neural. A retroalimentação das RNNs possibilitam um processamento temporal e dinâmico de informações, o que as tornam eficientes na resolução de problemas que variam no tempo (Haykin, 1998).

Um dos objetivos deste trabalho é comparar modelos de redes neurais mais simples e clássicos com os mais complexos e novos. Para esse propósito foi escolhido a Rede Neural de Jordan (*Jordan Neural Network* — JNN), já que ela é uma RNN simples. A JNN contém uma camada de contexto que guarda a saída anterior, da camada de saída. Assim, a camada de contexto recebe as respostas, conhecidas como *feedback*, das iterações anteriores e as transmite para a camada oculta, permitindo uma memória de curto prazo simplificada (Hallas e Dorffner, 1998). Os algoritmos de treinamento para a JNN podem ser tanto o *Backpropagation* comum quanto o RPROP.

2.6.4.3 *Stacked Autoencoder e Aprendizagem Profunda*

O SAE é um tipo de rede neural profunda que é construída a partir de camadas de *Autoencoders* esparsos, no qual a saída de cada camada é conectada com a entrada da camada seguinte (Chen *et al.*, 2012a). O SAE usa todos os benefícios de uma rede neural profunda e possui um alto poder de classificação. Por causa disso, ele pode aprender dados úteis sobre agrupamento hierárquico e decomposição da entrada em relação parte-todo (Ufdl, 2014).

A principal ideia das redes neurais profundas está no fato delas terem múltiplas camadas, formando representações utilizando os vários níveis de abstração dos dados. Consequentemente, redes mais profundas possuem um poder de aprendizagem superior em comparação com redes mais rasas (Bengio, 2009).

O algoritmo de treinamento do SAE é baseado em um treinamento não supervisionado, camada por camada, através de um algoritmo guloso. Cada camada é treinada de forma independente. Desse modo, os *Autoencoders* treinados são empilhados, e consequentemente, produzindo uma rede neural profunda pré-treinada (inicializada) (Bengio *et al.*, 2007; Chen *et al.*, 2012a; Ufdl, 2014).

O algoritmo de treinamento guloso é suficiente para treinar dados não rotulados, i.e.,

ele não treina os dados considerando a saída esperada. Por outro lado, para dados rotulados, tais como as séries temporais, o treinamento guloso não é suficiente e é usado como um pré-treinamento não supervisionado, em vez da inicialização aleatória dos neurônios. Depois disso, a etapa de refinamento (*fine-tuning*) é iniciada. Muitos trabalhos dão ênfase nos benefícios do treinamento não supervisionado e guloso usado para a inicialização de uma nova rede neural (Hinton *et al.*, 2006; Larochelle *et al.*, 2009; Ranzato *et al.*, 2007).

As etapas do algoritmo de treinamento do SAE são apresentadas no Algoritmo 2 (Bengio, 2009). O treinamento do SAE inicia-se treinando um *Autoencoder* para formar a primeira camada da rede neural (**linha 2**). Em seguida, na **linha 4**, as saídas da camada oculta do *Autoencoder* recém treinado são usadas como as entradas para a próxima camada, que também é treinada como um *Autoencoder* independente. Os treinamentos das **linhas 2 e 4** são puramente não supervisionados e usam dados não rotulados. Cada novo *Autoencoder* é empilhado no *Autoencoder* treinado anteriormente (**linha 5**). Esse processo é repetido até o SAE chegar ao número desejado de camadas ocultas (**linha 6**). Após formadas as camadas, na **linha 7**, as saídas da camada oculta do último *Autoencoder* treinado são usadas como entradas para a camada final (camada de saída). Por fim, na **linha 8**, um algoritmo de treinamento supervisionado é aplicado como *fine-tuning* para esta rede neural profunda.

Algoritmo 2 Treinamento guloso de um Stacked Autoencoders

```

1: procedure SAE-TRAINING
2:   Treine um Autoencoder para formar a primeira camada do SAE;
3:   repeat
4:     Use as saídas da camada oculta do Autoencoder anterior como entradas para
       treinar um novo Autoencoder;
5:     Empilhe o novo Autoencoder no topo do anterior;
6:   until Atingir o número desejado de camadas ocultas;
7:   Use as saídas do último Autoencoder treinado como entradas para a camada de
       saída do SAE;
8:   Aplique o fine-tuning para o SAE reconhecer dados rotulados;
9: end procedure
  
```

A ideia desse algoritmo de treinamento é que o pré-treinamento não supervisionado aproxime os pesos das camadas da rede neural para um bom mínimo local. Caso isso aconteça, a etapa de treinamento supervisionado será mais eficiente e encontrará melhores representações dos dados de entrada (Bengio, 2009).

2.7 Avaliação da Precisão da Predição

Para a predição do tráfego de Internet é usado uma série temporal dos bytes transmitidos nos dispositivos de rede. Na previsão de séries temporais, deve-se estimar como a sequência se comportará no futuro. Para isso são usadas as informações passadas da va-

riável que será prevista para descobrir os fatores que afetam seu comportamento no passar do tempo. Desse modo a predição será uma continuação da série temporal, baseada nos padrões e tendências identificadas na série observada.

A série predita não será exatamente a mesma da série original futura, pois as predições não são perfeitas, ou seja, ocorrem erros de predição. Por isso, é necessário uma medida da precisão da predição feita, ou seja, uma medida da taxa de erro cometida durante a predição da série temporal. Essa taxa de erro, é um dos fatores mais importantes na escolha de qual método de predição que será utilizado para o sistema de gerenciamento de largura de banda, proposto neste trabalho.

Para avaliar o resultado da predição existem alguns métodos estatísticos, sendo que a maioria deles é baseado nos mínimos quadrados (*least squares*) e na Soma dos Erros Quadrados (*Sum Squared Error* — SSE), dado por:

$$e_t = y_t - \hat{y}_t \quad (2.22a)$$

$$SSE = \sum_{i=1}^n e_i^2 \quad (2.22b)$$

onde e_t é o erro de predição no tempo t ; y_t é o valor real da variável no tempo t ; $\hat{y}_t$ é o valor previsto da variável no período t ; n é o número de predições realizadas.

Além do SSE os métodos mais usados são *Mean Squared Error* (MSE), *Root Mean Square Error* (RMSE), *Normalized Root Mean Square Error* (NRMSE) e o *Mean Absolute Percentage Error* (MAPE). Quanto menor for o valor dos erros, melhor será o método de previsão, os valores ideais são os mais próximos de 0 (zero). Com o erro calculado da predição de tráfego, pode-se compará-lo com os resultados dos erros de outros modelos de predição.

Neste trabalho, o MSE e o NRMSE foram utilizados como comparativo dos resultados dos modelos preditores. O MSE é simplesmente a média do SSE, já o NRMSE é a normalização da raiz quadrada do MSE:

$$MSE = \frac{SSE}{n}, \quad (2.23a)$$

$$NRMSE = \frac{\sqrt{MSE}}{y_{max} - y_{min}}, \quad (2.23b)$$

onde n é o número de predições realizadas, y_{max} é o valor real observado máximo e y_{min} é o valor real observado mínimo.

Gerenciamento de Redes de Computadores

O grande crescimento das redes de computadores traz uma preocupação envolvendo seu gerenciamento. Este gerenciamento é crítico para qualquer organização, principalmente para as que precisam suportar mais usuários e aplicações. Esse aumento na escalabilidade e complexidade das redes de computadores pode trazer, cada vez mais, problemas de desempenho, incapacitando totalmente ou parcialmente a rede de computadores (Stallings, 1999).

Uma rede de computadores grande não pode ser gerenciada somente com o esforço humano. A grande complexidade desse tipo de rede requer o uso de ferramentas automatizadas de gerenciamento de redes. O objetivo do gerenciamento de redes é auxiliar os engenheiros de rede a entender e controlar uma rede de computadores, de modo que os dados trafeguem com máxima eficiência e produtividade para os usuários da rede.

Neste capítulo são abordadas as ideias principais referentes ao gerenciamento de redes de computadores. As Seções 3.1 e 3.2 descrevem a base para o gerenciamento de redes, apresentando as áreas do gerenciamento de redes e as arquiteturas dos sistemas de gerenciamento de redes. Sequencialmente, a Seção 3.3 detalha como ocorre os congestionamentos nas redes de computadores e como esse congestionamento pode ser evitado e controlado. Na Seção 3.4 é descrito o protocolo SNMP, que é um protocolo de gerenciamento de redes. Por fim, a Seção 3.5 explana os mecanismos para o controle da largura de banda.

3.1 Áreas do Gerenciamento de Redes

As principais áreas funcionais do gerenciamento de redes, definidas pelo ISO, são: Gerenciamento de Falta (*Fault Management*); Gerenciamento de Contabilidade (*Accounting Management*); Gerenciamento de Configuração (*Configuration Management*); Gerenciamento de Desempenho (*Performance Management*); e Gerenciamento de Segurança (*Security Management*). Estas áreas funcionais foram classificadas considerando o modelo de referência OSI, no entanto essa classificação tem uma aceitação ampla pelos fornecedores dos dois tipos de sistemas de gerenciamento de redes, os padronizados e os proprietários (Stallings, 1999).

3.1.1 Gerenciamento de Falta

Para manter uma operação correta de uma rede de computadores cada componente específico é essencial e deve funcionar corretamente. Qualquer tipo de falta em um ou mais desses componentes de rede pode incapacitar a rede de forma total ou parcial. O conceito de falta é diferente do conceito de erro. Uma falta é uma condição anormal que precisa de uma ação da gerência para concertar o dispositivo ou processo problemático. Já um erro é um evento único ocasional. A falta geralmente ocorre devido uma falha constante em uma operação ou por erros excessivos, como por exemplo, em uma linha de comunicação fisicamente cortada, onde o sinal não poderá ser propagado (Stallings, 1999).

O Gerenciamento de Falta é importante para o funcionamento contínuo da rede. Quando ocorrer alguma falta, ela deve ser corrigida o mais rápido possível, para minimizar os problemas e prejuízos. Quando a falta é identificada os seguintes passos devem ser tomados de forma rápida (Leinwand e Fang, 1996; Stallings, 1999):

1. Determinar exatamente onde a falta ocorreu;
2. Isolar a rede a partir da falta para que ela possa funcionar sem interferência;
3. Reconfigurar ou modificar a rede de um modo que minimize os impactos da operação da rede, mesmo sem o componente ou componentes que falharam;
4. Consertar ou substituir os componentes falhos para restaurar a rede para o estado anterior à falta.

Os usuários esperam uma solução rápida e confiável dos problemas, para tal, são necessários diagnósticos e gerenciamento rápido, para minimizar o impacto e a duração da falta. Componentes redundantes também podem ser usados para alterar as rotas de comunicação e aumentar a tolerância a falhas e a confiabilidade. De forma resumida, o Gerenciamento de Falta ocorre através da detecção, isolamento e correção de operações anormais do ambiente de rede (Stallings, 1999).

3.1.2 Gerenciamento de Contabilidade

O Gerenciamento de Contabilidade faz a contabilização e verificação da utilização de recursos da rede por parte de cada conta de um usuário ou grupo de usuários. Os recursos para cada usuário devem ser assegurados de acordo com as permissões de cada um deles no acesso aos dados ou recursos da rede e na utilização do tráfego (Leinwand e Fang, 1996; Stallings, 1999). Cada usuário possui um limite de uso, o gerenciamento de contabilidade atribui as cotas dos usuários e controla se o limite foi atingido.

No gerenciamento da rede é importante contabilizar o uso dos seus recursos pelos usuários finais e por tipos diferentes de usuários. Isso é necessário para verificar se os usuários estão abusando de seus privilégios de acesso de modo a prejudicar a rede ou prejudicar os outros usuários. O engenheiro de rede pode usar esses dados detalhados da contabilidade da rede para melhorar o acesso dos usuários finais e planejar de forma mais fácil o crescimento da rede de computadores, limitando o acesso e manipulação de certas informações para certos usuários, por exemplo (Stallings, 1999).

3.1.3 Gerenciamento de Configuração

Redes de comunicações modernas são compostas de componentes individuais e subsistemas lógicos, nos quais podem ser configurados para trabalhar em muitas aplicações diferentes. Desse modo, o mesmo dispositivo pode ser configurado para trabalhar tanto quanto um roteador (ou um *intermediate system* — IS) quanto um dispositivo do usuário (ou um *end system* — ES). Após definir como o dispositivo será usado, o gerenciador de configuração pode escolher os *softwares* ou atributos apropriados para esse dispositivo (Stallings, 1999).

O Gerenciamento de Configuração se preocupa em inicializar a rede e desligar uma parte ou toda a rede. Esse gerenciamento tem a função de recuperar as informações da rede e usar essas informações na instalação dos dispositivos dela. Também é necessário manter, adicionar e atualizar as relações e configurações entre componentes e seus status durante as operações da rede (Leinwand e Fang, 1996; Stallings, 1999).

Para realizar o Gerenciamento de Configuração é necessário manter relatórios com os inventários dos dispositivos da rede de computadores. Por isso é importante recuperar informações sobre os dispositivos, para que futuramente seja possível modificar a configuração, caso necessário. E além disso, é necessário produzir novos relatórios, para obter um inventário atualizado do componentes da rede.

3.1.4 Gerenciamento de Segurança

O Gerenciamento de Segurança cuida da proteção da informação e o controle de acesso das instalações. Isso inclui gerar, distribuir e armazenar chaves criptografadas. Todas as

senhas e outros acessos de informações restritas devem ser mantidos e distribuídos. O Gerenciamento de Segurança também monitora e controla os acessos às redes de computadores e os pontos de acessos às informações das partes ou nós da rede (Stallings, 1999). Esse tipo de gerenciamento também foca em disponibilizar certos dados somente para usuários autorizados.

Os relatórios de eventos (*logs* de dados) também são uma ferramenta importante para a segurança. A gerência da segurança é interligada à coleta, armazenagem e análise dos registros e relatórios de segurança. Além das senhas de segurança, uma organização também precisa armazenar de forma segura outras informações, como por exemplo: informações bancárias, folha de pagamento, registros sigilosos e outros. As seguintes tarefas são importantes para o Gerenciamento de Segurança (Leinwand e Fang, 1996; Stallings, 1999):

- identificar quais informações devem ser protegidas;
- identificar os pontos de acesso da rede de computadores;
- garantir e manter seguros os pontos de acesso.

3.1.5 Gerenciamento de Desempenho

As redes de computadores atuais são compostas de muitos componentes diferentes, que devem ser interligados e compartilham dados e recursos. Em muitos casos, é crítico que o desempenho de uma rede esteja em um certo limite mínimo aceitável, de modo que os dispositivos dela não fiquem sobrecarregados. Relatórios do desempenho dos componentes da rede são muito importantes para o planejamento da rede.

O Gerenciamento de Desempenho envolve medidas de desempenho do *hardware* e *software* utilizando parâmetros de QoS. Existem vários fatores que podem alterar o desempenho e devem ser levados em consideração, como por exemplo as taxas de erro, tempo de resposta, tráfego excessivo, vazão (*throughput*), utilização de recursos e largura de banda (*bandwidth*) (Leinwand e Fang, 1996; Stallings, 1999).

O Gerenciamento de Desempenho de uma rede de computadores envolve duas categorias diferentes, o monitoramento e o controle. Monitoramento é a função que observa e coleta dados das atividades na rede. O monitoramento é importante para obter as estatísticas do desempenho e pode ser usado para encontrar algum gargalo antes mesmo dos usuários finais perceberem. O controle faz ajustes na rede para melhorar seu desempenho. No controle, ações corretivas podem ser tomadas para mudar rotas ou recursos de rede durante horários de pico quando o gargalo é encontrado (Stallings, 1999).

3.2 Sistemas de Gerenciamento de Redes

Um sistema de gerenciamento de redes é uma coleção de ferramentas que faz o monitoramento e o controle da rede de computadores. Um bom sistema de gerenciamento precisa de dois elementos, uma plataforma de gerenciamento de rede e *hardwares* incorporados a essa plataforma (Leinwand e Fang, 1996; Stallings, 1999). A plataforma de gerenciamento deve operar em uma interface única e poderosa, mantendo comandos não somente amigáveis ao usuário, mas também que execute a maioria das tarefas de gerenciamento de rede. Tanto os *hardwares* quanto os *softwares*, que fazem parte do gerenciamento da rede, devem ser incorporados à ela para que exista uma comunicação simples durante o monitoramento e controle da rede de computadores.

A arquitetura do sistema de gerenciamento é separada em duas, arquitetura de *software* e arquitetura geral. O *software* de gerenciamento pode ser dividido em três partes: (i) a interface com o usuário; (ii) o *software* do gerenciamento de rede; (iii) e o *software* com suporte a comunicação e base de dados. A arquitetura geral dos sistemas de gerenciamento pode ser dividida entre (a) Centralizada, (b) Hierárquica e (c) Distribuída (Leinwand e Fang, 1996; Stallings, 1999).

A arquitetura Centralizada de um sistema de gerenciamento utiliza somente um computador como centro de controle, no qual é responsável de realizar todas as obrigações do gerenciamento. Essa é a arquitetura mais tradicional, principalmente em ambientes com um *mainframe* dominante, onde os recursos principais estão concentrados.

A arquitetura Hierárquica substitui o centro de controle único por vários sistemas, onde um atua como servidor central e outros trabalham como clientes, onde cada sistema possui um conjunto de funcionalidades específicas. Esse tipo de arquitetura usa uma base de dados única, porém há uma distribuição das tarefas do gerenciamento.

Finalmente, a arquitetura Distribuída é uma combinação das arquiteturas anteriores, utilizando múltiplas plataformas espalhadas pela rede, com um sistema central líder e vários outros subsistemas independentes. O sistema central e os subsistemas são interoperáveis, eles ficam localizados em LANs distribuídas por toda a organização, por exemplo. Cada subsistema tem acesso limitado em relação ao monitoramento e controle e, geralmente, são divididos em departamentos. O sistema central líder possui acesso global e pode gerenciar todos os recursos da rede.

O trabalho realizado e descrito nesta dissertação propôs um sistema de gerenciamento centralizado que atue no Gerenciamento de Desempenho, em ambos, monitoramento e controle. Este monitoramento foca na coleta de dados, através do SNMP, do tráfego transmitido em um enlace de comunicação. A partir do tráfego coletado, uma rede neural foi treinada para reconhecer os padrões e tendências dele.

A etapa de controle de rede de computadores usa a rede neural para prever o tráfego futuro, a partir daí, dependendo do tráfego futuro, é tomada a decisão de alterar a largura

de banda das interfaces de uma forma mais justa. As interfaces que necessitarem mais tráfego receberam uma largura de banda maior e mais justa. Tudo isso é feito com a finalidade de diminuir o atraso da transferência de pacotes para diminuir o congestionamento nessas interfaces.

3.3 Congestionamento nas Redes de Computadores

A transmissão de um dado em uma rede de computadores não é feita de forma contínua. Na verdade, antes de transmitir um certo dado, ele é dividido em blocos pequenos, chamados de pacotes, que são enviados individualmente. Por isso, as redes de computadores são comumente chamadas de redes de pacotes ou redes de comutação de pacotes.

O uso de pacotes é feito por dois motivos. O primeiro é a coordenação da transmissão para garantir que os dados chegaram intactos, sem alterações ou erros. A divisão dos dados em pacotes pequenos ajuda o receptor a determinar quais blocos chegaram sem erros e quais não chegaram. O segundo motivo é para garantir o acesso justo a uma instalação de rede compartilhada, um sistema de rede não pode impedir o acesso de computadores em prol de um outro computador. O uso dos pacotes pequenos ajuda a melhorar a justiça no acesso, onde cada pacote, de diferentes origens, tem seu tempo de uso em instalações compartilhadas (Comer, 2007).

Durante a transmissão dos dados em uma rede de computadores pode ocorrer um aglomerado muito grande de pacotes em uma parte desta rede. Quando isso acontece, há uma alta probabilidade de haver atraso ou perda de pacotes, degradando o desempenho da rede. Essa situação é chamada de congestionamento.

O congestionamento acontece nos roteadores, então ele pode ser detectado na camada de rede. Entretanto, o congestionamento ocorre pelo tráfego gerado pela camada de transporte, e este tráfego pode se intensificar ainda mais devido a perda e retransmissão dos pacotes. Logo, as camadas de rede e transporte, do modelo de referência OSI, compartilham a responsabilidade para tratar o congestionamento (Tanenbaum e Wetherall, 2010). A retransmissão dos pacotes perdidos, portanto, trata um dos sintomas do congestionamento nas redes, que é a perda dos pacotes, porém não trata a causa do congestionamento (Kurose e Ross, 2012).

Um problema típico da rede de computadores é alocar os recursos da rede de forma justa e eficaz, para todos os usuários que usam a rede de forma concorrente. Esses recursos que são compartilhados incluem a largura de banda dos enlaces e os *buffers* de memória nos roteadores ou *switches* onde os pacotes estão enfileirados, aguardando sua transmissão (Peterson e Davie, 2011). Quando há muitos pacotes em um roteador, eles ficam em espera, aguardando sua vez de ser transmitido no enlace. Se a frequência de chegada de novos pacotes for maior do que a frequência de saída, então ocorrerá um congestionamento, o que faz com que os pacotes que estão chegando sejam descartados, pois não há espaço

suficiente na fila de espera de transmissão. Para tratar a causa do congestionamento de rede, são necessários mecanismos e algoritmos de controle de congestionamento, que podem ser bem variados.

A Figura 3.1 esboça um exemplo de um gargalo de velocidades, no qual o roteador recebe dados de forma mais rápida do que ele consegue enviar esses dados. O roteador que atua como *gateway*, recebendo dados de duas fontes diferentes ($f1$ e $f2$) e enviando esses dados para o mesmo destino (d). O roteador recebe os pacotes nas interfaces de entrada e os coloca na interface de saída, em seguida, envia um pacote por vez para o destino. Nesse exemplo, ao passar do tempo, provavelmente irá ocorrer um acúmulo muito grande de pacotes na fila de espera do roteador, que dará início ao congestionamento.

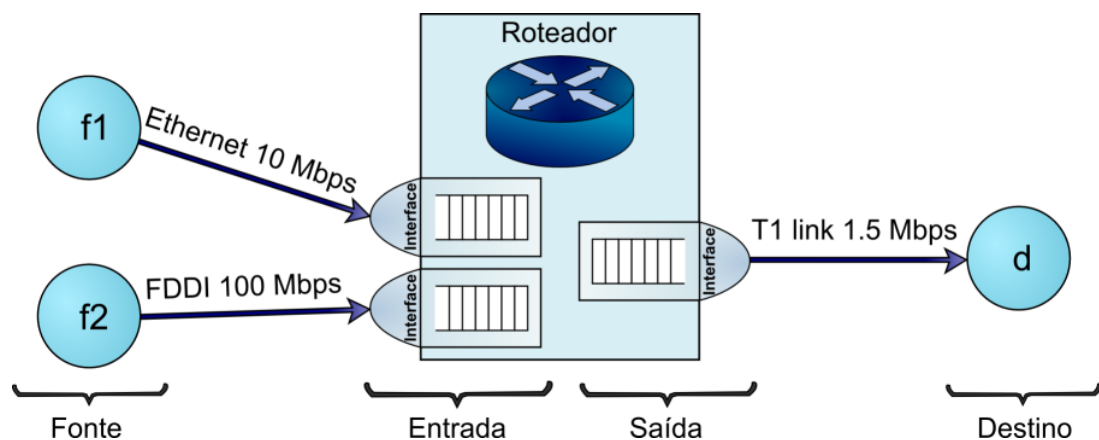


Figura 3.1: Um exemplo de um gargalo que pode gerar congestionamento. Os enlaces das fontes que enviam dados tem uma largura de banda de 10 Mbps (*Ethernet*) e 100 Mbps (*Fiber Distributed Data Interface* — FDDI). O enlace de saída do roteador possui uma largura de banda de 1.5 Mbps (*T1 link*). Fonte: Adaptado de Peterson e Davie (2011).

3.3.1 Controle de Congestionamento e Alocação de Recursos

A Figura 3.2 retrata o início de um congestionamento. Na área sem congestionamento, caso sejam ignorados os possíveis erros e faltas na rede, o número de pacotes úteis e (não repetidos) enviados pelos *hosts* são todos recebidos em seu destino final. Contudo, quando a carga do tráfego se aproxima da capacidade de transmissão da rede, algumas rajadas de tráfego podem encher os *buffers* de memória de um roteador, por exemplo, fazendo com que alguns pacotes sejam perdidos. Desse modo, haverá retransmissões de pacotes o número de pacotes entregues caí abaixo do número ideal. A partir desse ponto, a rede já está congestionada.

No exemplo anterior, Figura 3.2, a rede pode sofrer de um colapso de congestionamento. Neste colapso, os pacotes não serão entregues, haverá um atraso muito grande no seu envio, os *hosts* poderão reenviá-los, gerando pacotes que podem não ser mais úteis e o desempenho da rede, ou parte dela, cairá drasticamente. Assim, a rede de computadores

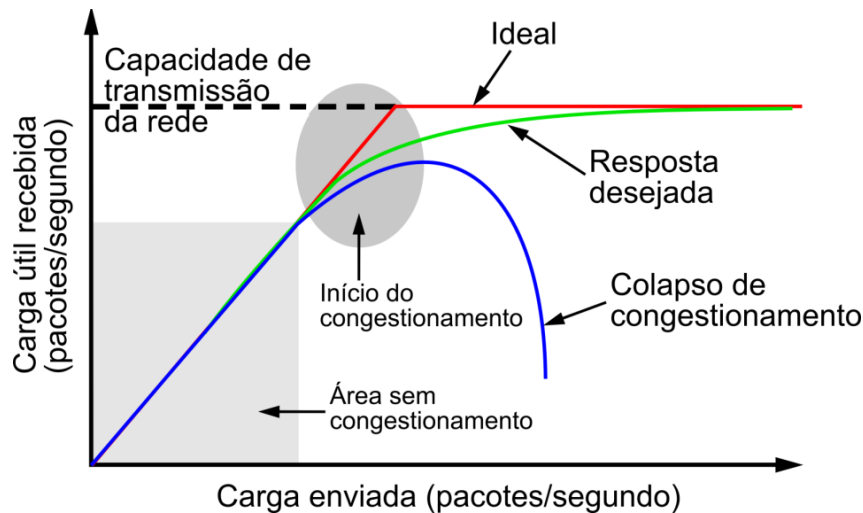


Figura 3.2: Ilustração gráfica do congestionamento. Quando o tráfego ocorre em excesso, o desempenho cai drasticamente. Fonte: Adaptado de Tanenbaum e Wetherall (2010).

deve ser projetada para lidar com essa situação, usando o controle de congestionamento e a alocação de recursos da rede de computadores (Tanenbaum e Wetherall, 2010).

A alocação de recursos e o controle de congestionamento são problemas complexos e são áreas de pesquisa bastante estudadas. Um dos fatores que deixam esses problemas mais complexos é que eles não estão isolados em somente uma camada do modelo OSI. A alocação de recursos é parcialmente implementada nos roteadores ou *switches*, dentro da camada de rede. Ela também é parcialmente implementada na camada de transporte (Peterson e Davie, 2011).

A alocação de recursos é um processo pelo qual os elementos da rede tentam atender às demandas, que são concorrentes, das aplicações que precisam dos recursos (físicos ou lógicos) da rede, por exemplo, a largura de banda e espaços na fila de espera em roteadores ou *switches*. Isso nem sempre acontece, fazendo com o que algumas aplicações e usuários recebam menos recursos de rede do que eles precisam. O controle de congestionamento são os esforços feito pelos componentes de rede para prevenir ou diminuir o congestionamento (Peterson e Davie, 2011).

No controle de congestionamento, inicialmente, é feita uma prevenção a fim de evitar o congestionamento, porém, caso este ocorra, então são tomadas atitudes para diminuí-lo ou dissipá-lo. O controle de malha fechada é o que trata o congestionamento após seu acontecimento. Já o controle de malha aberta, aplica políticas para impedir que o congestionamento ocorra.

O controle de congestionamento e a alocação de recursos estão inteiramente ligados. Caso a rede de computadores atue ativamente na alocação de recursos, então o congestionamento pode ser evitado, desse modo, não seria necessário o controle de congestionamento (Peterson e Davie, 2011). O controle de congestionamento pode atuar de diversas formas, como:

- Controle de quais pacotes devem ser descartados primeiro, como o algoritmo de descarte *Random Early Detection* (RED), no qual utiliza o tamanho médio das filas, descartando pacotes de acordo com cálculos probabilísticos (Rajput *et al.*, 2014);
- Alocação dinâmica de recursos, para diminuir o atraso e a variância do atraso durante as transferências dos pacotes (An *et al.*, 2014);
- Disciplinas de filas para controlar a ordem de transmissão dos pacotes, como por exemplo, a técnica *Virtual Output Queue* (VOQ), que segrega as filas de entrada em diversas classes diferentes, ao invés de manter somente uma fila, para melhorar a vazão do tráfego (Rajput *et al.*, 2014);
- Controle de admissão, para recusar conexões e acessos que poderiam congestionar a rede (Yan *et al.*, 2014);
- Controle para evitar o congestionamento, onde a rota do tráfego é escolhida de tal forma que não ocorra o congestionamento, como o método de rota mais rápida evitando áreas que estejam próximas de um congestionamento (Huang *et al.*, 2014).
- Notificação de congestionamento, que identifica que o congestionamento já ocorreu e avisa outros componentes de rede para que evitem trafegar dados na área congestionada (Zhang e Ansari, 2013);
- Métodos de segregação de tráfego, onde certos pacotes não afetam outros pacotes (Kurose e Ross, 2012; Tanenbaum e Wetherall, 2010);
- Controle da velocidade que um *host* envia os pacotes; e várias outras (Peterson e Davie, 2011).

Uma alocação de recursos eficaz, além de evitar o congestionamento, também ajuda a melhorar o desempenho das entidades de transporte, uma vez que toda a largura de banda disponível seria usada em sua capacidade máxima, sem congestionamentos. Ainda que esse seja o cenário ideal (como mostra a Figura 3.2), é algo extremamente difícil de ser feito, pois o tráfego geralmente tem características de rajadas. Devido a essas rajadas, em um enlace de 100 Mbps para cinco entidades de transporte, por exemplo, cada uma das entidades deve receber menos do que 20 Mbps (100 Mbps/5 entidades), mesmo que o ideal seja 20Mbps (Peterson e Davie, 2011; Tanenbaum e Wetherall, 2010).

O controle de congestionamento pode ser feito de maneiras simples, como simplesmente enviar uma mensagem para algum *host* para que ele pare de enviar dados. Entretanto, essa não é uma abordagem aceitável, os mecanismos de controle de congestionamento devem tratar os *hosts* com justiça. A distribuição dos recursos deve ser de forma justa para todos os *hosts*, mesmo se os recursos forem poucos, para não gerar um grande dano ou perda para somente um *host* (Peterson e Davie, 2011).

3.3.2 Congestionamento em Roteadores

A Figura 3.3 retrata a arquitetura de um roteador que encaminha os pacotes que chegam nas portas (interfaces) de entrada, passando pela matriz de comutação (*switching fabric*), para uma das portas de saída. A porta de saída é decidida pelo processamento de roteamento, através dos protocolos de roteamento e das tabelas de endereçamento (ou roteamento), que indicam o caminho que o pacote deve tomar. Além disso, o processamento de roteamento também pode executar funções de gerenciamento de rede, como alterar as tabelas de roteamento para evitar instabilidades na rede. As portas de entrada e de saída fazem parte da camada física e fazem a ligação física de um enlace com o roteador. A matriz de comutação conecta as portas de entrada com as portas de saída, é nesse componente que os pacotes são de fato encaminhados de uma interface de entrada para uma interface de saída (Kurose e Ross, 2012).

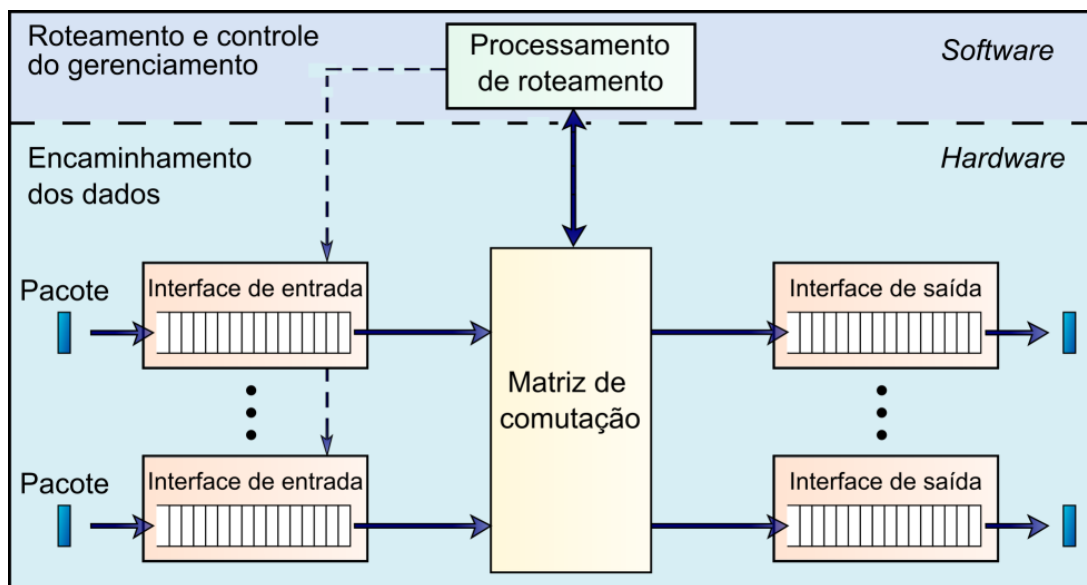


Figura 3.3: Arquitetura de um roteador. Fonte: Adaptado de Kurose e Ross (2012); Peterson e Davie (2011).

As portas de entrada, além das funções da camada de rede, realizam funções da camada de enlace necessárias para interoperar com a camada de enlace de origem do pacote. A partir dos pacotes que estão na fila de entrada, o processamento de roteamento consulta as tabelas de roteamento para decidir para qual porta de saída o pacote deve ser encaminhado. O encaminhamento da entrada para saída é feito pela matriz de comutação e pode ser efetuado de vários modos, os mais comuns são: encaminhamento por memória; encaminhamento por barramento; encaminhamento por uma rede de interconexões. As portas de saída armazenam os pacotes recebidos pela matriz de comutação e transmitem esses pacotes para o enlace físico de saída, atuando nas funções necessárias da camada de enlace de física. Quando a conexão é bidirecional, isto é, o tráfego pode fluir em ambas direções, uma porta de saída é pareada com uma porta de entrada, formando uma conexão

única (Kurose e Ross, 2012; Tanenbaum e Wetherall, 2010).

Além do tempo de transferência de um pacote em um enlace de conexão física, também há o tempo que esse pacote permanece dentro do roteador, ambos causam atraso na comunicação. Esse tempo é determinado pelo tempo de espera e processamento que o pacote sofre em um roteador. Então, cada processo explicado anteriormente, toma um certo tempo. Dentre esses tempos, estão:

1. tempo de espera na fila de entrada;
2. tempo de processamento da decisão da saída apropriada para um certo pacote;
3. tempo para a troca de filas, feita pela matriz de comutação;
4. tempo de espera na fila de saída, antes do pacote ser de fato enviado.

Todos esses tempos geram um atraso na transferência, caso ocorra um aumento de fluxo de pacotes muito grande no roteador, esse atraso será cada vez maior, podendo acarretar em um congestionamento no roteador.

3.3.3 Critérios Básicos para o Controle de Congestionamento

O controle de congestionamento possui duas medidas, referentes ao desempenho da rede, que devem ser consideradas na escolha de um bom método. Este, deve levar em conta tanto a eficácia quanto a justiça, para a alocação de recursos de rede. A eficácia da alocação de recursos está relacionada a atingir o objetivo com eficiência, ou seja, alocar os recursos para todos os componentes que precisarem. A justiça, na alocação de recursos, é uma área mais nebulosa, pois se relaciona a todos elementos terem os recursos que merecem, isto é, dar mais recursos para quem precisa de mais recursos, porém, de modo que não prejudique os outros. Isto, é algo mais difícil de ser feito (Peterson e Davie, 2011).

A vazão, ou *throughput*, em uma rede de computadores tem uma forte influência no desempenho da mesma. A largura de banda, ou *bandwidth*, é a medida da capacidade total de bits por segundo que pode ser transmitida em um meio. Já a vazão, é o número de bits por segundo que de fato é transmitido na rede, ou seja, a vazão poderá ser no máximo igual a largura de banda.

O atraso, latência ou *delay*, em uma rede de computadores, também é uma medida de desempenho. O atraso está relacionado ao tempo de transferência de um pacote entre enlaces físicos (e.g., como um cabo de cobre) mais o tempo de espera em um componente da rede (e.g., como um roteador) (Kurose e Ross, 2012; Peterson e Davie, 2011). O atraso é, basicamente, o tempo de transmissão no meio físico mais o tempo de comutação, processamento e espera do pacote nos dispositivos de rede.

Uma medida comum, usada pelos projetistas, para avaliar a eficácia da alocação de recursos de rede é a razão da vazão pelo atraso (Peterson e Davie, 2011). O ideal é maximizar essa razão, para ter o máximo de vazão, com o menor atraso possível. Porém,

existe o problema de que se aumentar muito a quantidade de pacotes transmitidos na rede, para aumentar a vazão, também aumenta o tempo de espera nas filas de cada roteador que os pacotes transitam, aumentando também o atraso. A carga ideal de dados na rede deve ser a máxima, todavia, não pode ser alta o suficiente para gerar atrasos maiores e congestionamentos (Peterson e Davie, 2011).

A Figura 3.4 contém um gráfico que mostra a curva da razão entre vazão e atraso. O ponto máximo da curva é a carga ótima da rede. Na área à esquerda da carga ótima, o controle de congestionamento pode estar sendo muito conservativo, evitando que pacotes sejam enviados para não ocorrer congestionamentos. Já na área à direita, uma quantidade muito grande de pacotes estão sendo enviados na rede, o que gera o aumento do atraso, devido às esperas nas filas aumentarem. Além disso, os pacotes podem ser descartados quando as filas encherem, originando retransmissões de pacotes, agravando o congestionamento na rede de computadores.

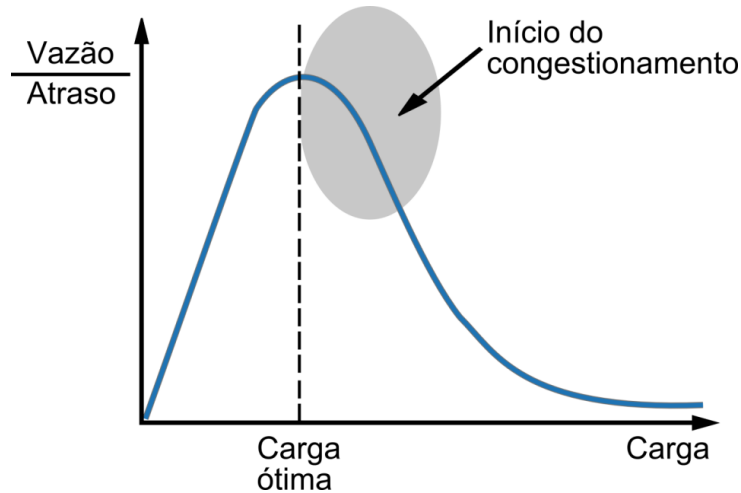


Figura 3.4: Razão da vazão (bits/segundo) pelo atraso (segundos) em função da carga (pacotes/segundo). Fonte: Adaptado de Peterson e Davie (2011).

O efeito dos atrasos pode ser visualizado na Figura 3.5. Quando a carga do tráfego de rede é muito grande e se aproxima da capacidade máxima do roteador, o tempo de espera na fila aumenta para cada pacote, assim, o atraso de envio dos pacotes cresce exponencialmente, causando o congestionamento. As retransmissões dos pacotes agravam ainda mais esse problema (Kurose e Ross, 2012; Peterson e Davie, 2011).

A Equação (3.1) mostra a relação entre o atraso e a vazão em um roteador:

$$D = \frac{D_0}{1 - U}, \quad (3.1)$$

onde D representa o atraso real, D_0 é o atraso quando a rede está ociosa e U é um valor pertencente no intervalo $[0, 1]$, que indica a porcentagem da utilização atual da rede. Desse modo, se a rede não estiver em uso, U será igual a zero e o atraso real será igual a D_0 . Quando o tráfego chega próximo da capacidade da rede, o atraso tende ao infinito

(Comer, 2007; Tanenbaum e Wetherall, 2010).

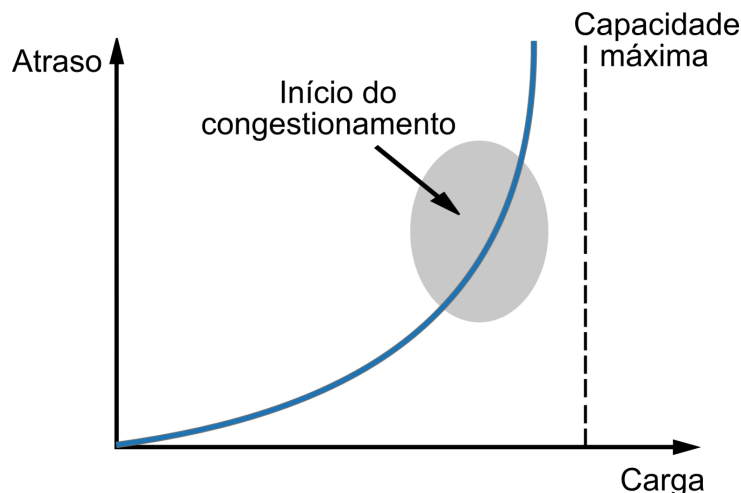


Figura 3.5: Atraso (segundos) em função da carga (pacotes/segundo). Fonte: Adaptado de Peterson e Davie (2011); Tanenbaum e Wetherall (2010).

3.4 Protocolo SNMP

As redes de computadores estão se tornando cada vez maiores, mais heterogêneas e, por consequência, mais complexas. Devido a esse aumento da complexidade o custo do gerenciamento das redes também fica maior. Para controlar esses custos, ferramentas padronizadas de gerência são necessárias, nas quais possam ser usadas em vários tipos diferentes de produtos, incluindo *end systems*, *bridges*, roteadores e equipamentos gerais de telecomunicação. As ferramentas de gerência também devem poder ser usadas em um ambiente misto com produtos de fornecedores diferentes (Stallings, 1999).

O *Simple Network Management Protocol* (SNMP) foi concebido em resposta a essa necessidade de se ter ferramentas que possam ser usadas em um amplo espectro de produtos e com custo baixo. O protocolo SNMP foi desenvolvido para providenciar uma ferramenta para vários fornecedores, com alta portabilidade e escalabilidade (Leinwand e Fang, 1996; Stallings, 1999). Ele é mais do que um protocolo, pois ele se refere há um conjunto de padronizações para o gerenciamento de redes, incluindo tanto o protocolo quanto a definição de uma base de dados e conceitos associados à gerência. A partir de 1989, o SNMP foi adotado como o padrão para redes de computadores baseadas no *Transmission Control Protocol/Internet Protocol* (TCP/IP) e sua popularidade somente aumentou (Case *et al.*, 1990).

O modelo de gerenciamento de rede que o SNMP provém inclui quatro elementos-chaves: estação de gerenciamento (gerente), agente de gerenciamento (agente), base de informação do gerenciamento (*Management Information Base* — MIB) e os protocolos de gerenciamento de rede. Assim, o gerente envia requisições de gerenciamento para os agentes, cujas funções são obter informações sobre os objetos gerenciados e executar a

ações necessárias para o gerenciamento. A MIB é a base de dados que contém informações sobre os elementos (objetos) a serem gerenciados, cada recurso a ser gerenciado é representado por um objeto e a coleção de todos os objetos é armazenada na MIB (Karris, 2009; Stallings, 1999).

O protocolo SNMP foi projetado como atuador na camada de aplicação, como padrão para arquiteturas baseadas no conjunto de protocolos TCP/IP. Ele opera sobre o *User Datagram Protocol* (UDP) (Leinwand e Fang, 1996; Stallings, 1999). A Figura 3.6 mostra uma configuração típica dos protocolos que usam o SNMP. Os blocos na cor cinza indicam o ambiente operacional que pode ser gerenciado. Os blocos na cor branca são fornecedores de funções que serão utilizadas no gerenciamento de rede.

A configuração da Figura 3.6, apresenta o modelo de gerenciamento com uma estação de gerenciamento única, onde um processo gerente controla o acesso a uma MIB central e providencia uma interface para o gerenciador da rede. O processo gerente (ou gerenciador) é capaz de conduzir o gerenciamento usando o SNMP (uma camada abaixo), que é implementado acima do UDP, IP e outros protocolos dependentes à rede (por exemplo, *Ethernet*, *Fiber Distributed Data Interface* — FDDI, e X.25). A partir da versão 2 do SNMP (SNMPv2) é possível a interação entre dois gerentes, possibilitando uma arquitetura de gerenciamento distribuída (Case *et al.*, 1990; McCloghrie e Rose, 1991; SNMPv2 Working Group *et al.*, 1996).

A Figura 3.7 exhibe o papel do protocolo SNMP, mostrando todo o contexto que envolve as trocas de mensagens entre agente e gerente. Uma estação de gerenciamento comunica-se com um agente SNMP através de uma rede qualquer, isso é feito pela aplicação de gerenciamento que pode enviar três tipos de mensagens SNMP, que são: **GetRequest**, **GetNextRequest** e **SetRequest**. Destas, as duas primeiras são variações da função **get**, usada para solicitar valores.

A função **set** do protocolo SNMP, é usada para atribuir (ou atualizar) valores. O agente reconhece todas essas três mensagens, passando para a aplicação de gerenciamento uma mensagem **GetResponse**. Além disso, um agente pode enviar uma mensagem **trap** (armadilha) em resposta a um evento que afeta a MIB e aos recursos gerenciáveis subjacentes (Leinwand e Fang, 1996; Stallings, 1999).

O SNMP é um protocolo sem conexão (*connectionless*), pois se baseia no UDP, que também é um protocolo sem conexão. Por isso, não ocorre manutenção de conexões entre uma estação de gerenciamento e seus agentes. Ao invés disso, cada troca é uma transação separada entre uma estação de gerenciamento e um agente (Stallings, 1999). A Figura 3.8 mostra um diagrama de sequências com as cinco mensagens SNMP mencionadas anteriormente. As primitivas formam a comunicação entre Gerente (estação de gerenciamento) e Agente (agente SNMP).

As estações de gerenciamento do SNMP são responsáveis por executar aplicações de monitoramento e controle nos elementos de rede. Os elementos de rede são equipamentos

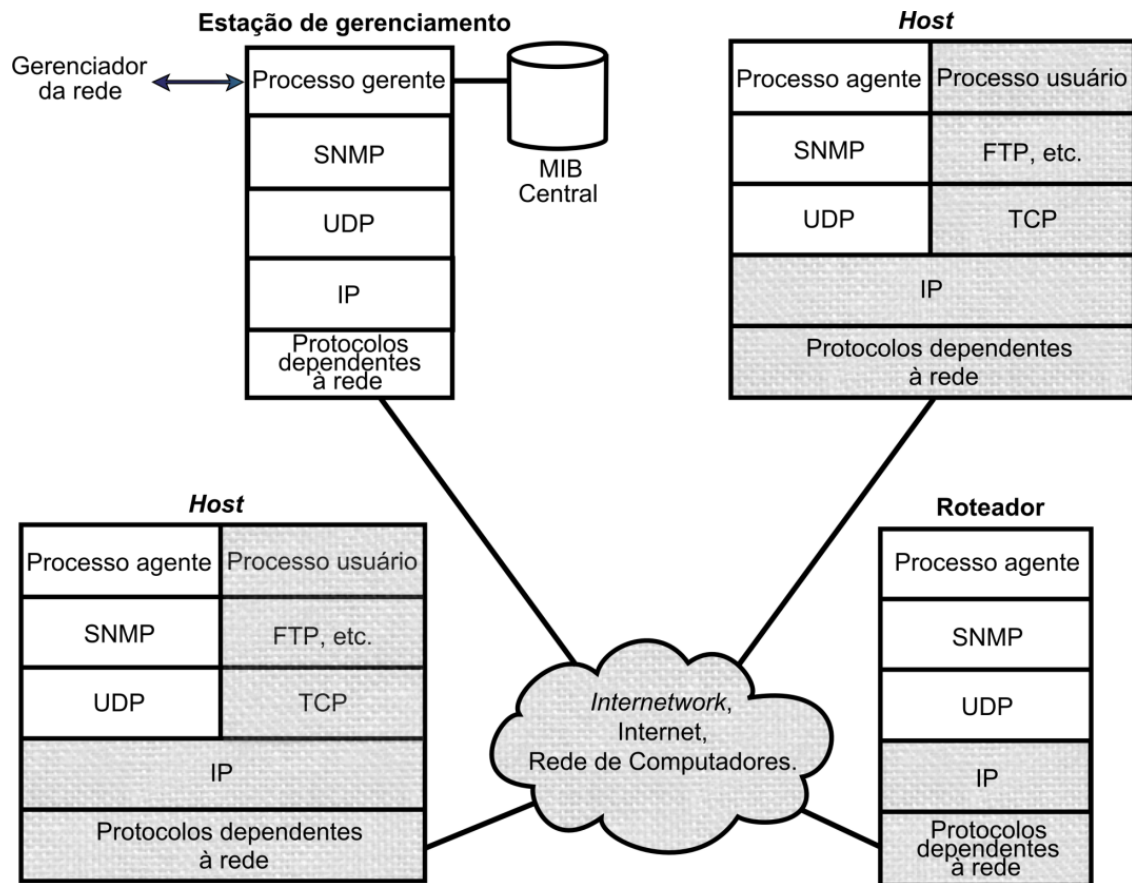


Figura 3.6: Configuração típica dos protocolos para o SNMP. Fonte: Stallings (1999).

como roteadores, *hosts*, *gateways*, *bridges*, entre outros, por exemplo. Esses elementos possuem agentes de gerenciamento responsáveis pela execução das funções de gerência de rede requisitadas pelos gerentes. Os objetos gerenciados ficam armazenados na MIB, que é a base de dados responsável pelo armazenamento de informações sobre os recursos gerenciados (Stallings, 1999).

Para apenas uma estação de gerenciamento, um processo gerenciador controla o acesso a uma MIB central na estação de gerenciamento e fornece uma interface para o gerenciador de rede. Os objetos contidos na MIB podem ser lidos e modificados pela aplicação de gerenciamento, através de um agente SNMP. O agente é um elemento transparente no processo de gerenciamento. O processo gerenciador realiza o gerenciamento de rede usando o SNMP, que está implementado sobre o UDP, IP, e protocolos dependentes de rede. Cada agente deve também implementar SNMP, UDP, e IP. Em suma, um processo agente interpreta as mensagens SNMP e controla os agentes da MIB.

O SNMP permite a gerência de qualquer dispositivo em uma rede, desde que esse dispositivo proporcione a recuperação e o envio das informações de gerenciamento pelo SNMP. Desse modo, é possível também obter as informações dos dispositivos físicos da rede, tais como, impressoras, modem, fontes de energia, banco de dados, sistemas operacionais como Linux, Windows e outros (Stallings, 1999). Geralmente, são os fornecedores

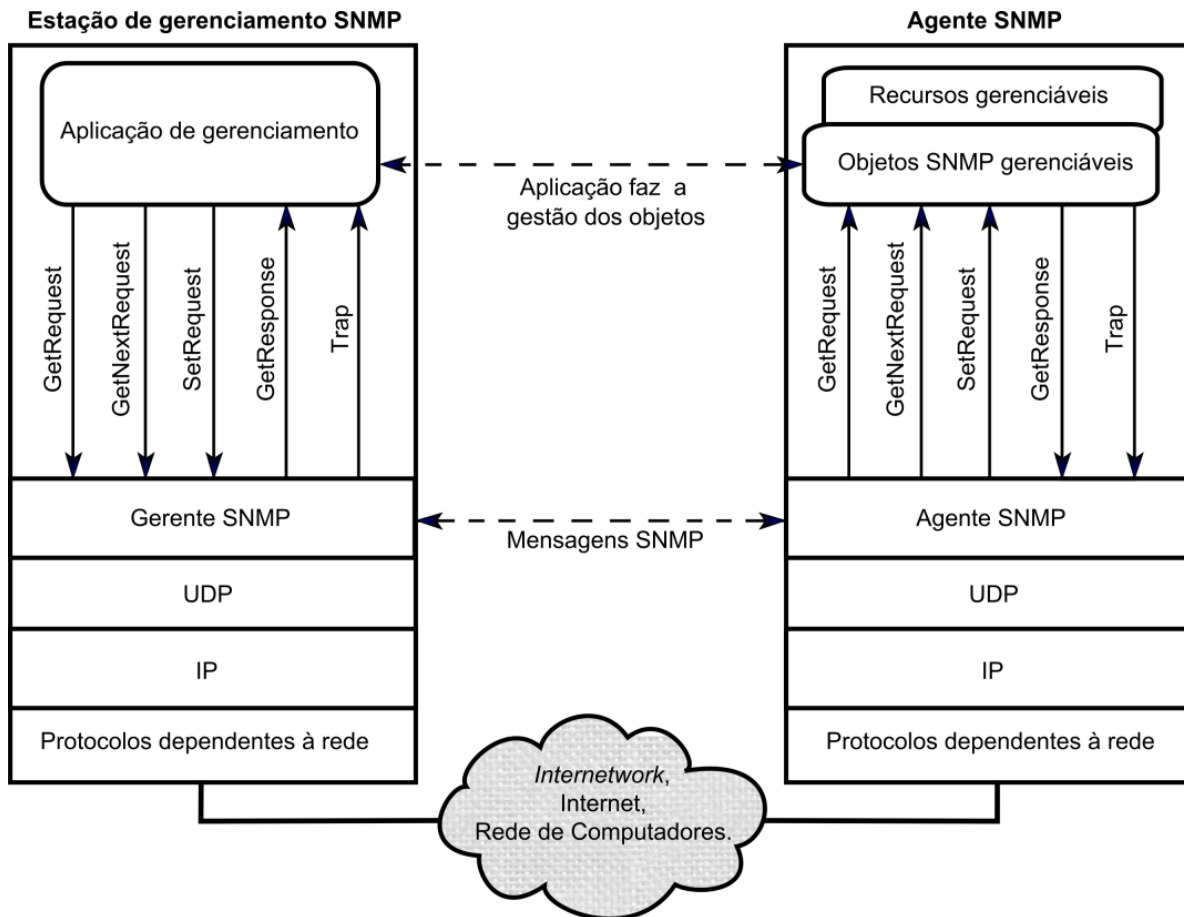


Figura 3.7: O papel do SNMP e suas mensagens principais entre agente e gerente. Fonte: Stallings (1999).

de equipamentos que implementam os agentes em seus produtos, oferecendo uma portabilidade grande no uso do SNMP.

3.4.1 Estrutura da MIB

Para qualquer sistema de gerenciamento de rede, as informações sobre os elementos que são gerenciáveis deve ser armazenadas. Para ambos ambientes, OSI ou TCP/IP, essas informações são armazenadas na base de informações do gerenciamento, chamada MIB (Stallings, 1999). Uma MIB é uma base de dados de objetos gerenciáveis que são acessados pelos protocolos de gerenciamento de rede. Uma MIB do SNMP é um conjunto de parâmetros nos quais uma estação de gerenciamento SNMP pode consultar, definir ou alterar no agente SNMP de um dispositivo de rede, como um roteador (Karris, 2009). Cada recurso gerenciável é representado por um objeto. A MIB é uma coleção estruturada desses objetos.

O SNMP possui um conjunto de especificações com diversas bases de dados e funções para o gerenciamento de rede. As especificações são adotadas pelo *Internet Engineering Task Force* (IETF) e descritas nos documentos *Request for Comments* (RFC). Esse

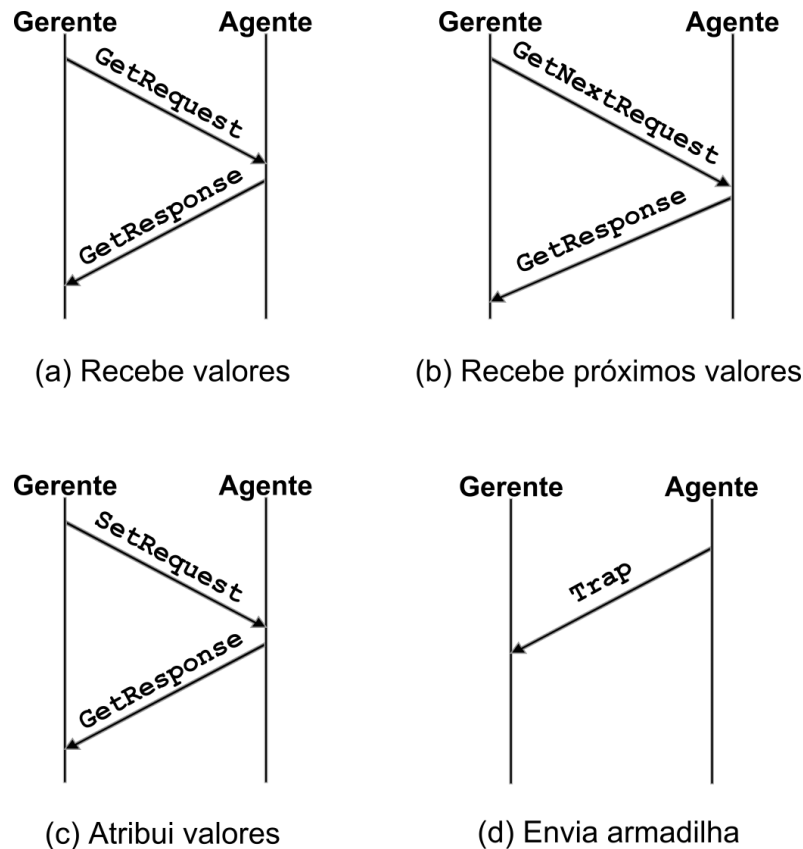


Figura 3.8: Sequencias das mensagens de comunicação SNMP. Fonte: Stallings (1999).

conjunto de especificações é amplo e continua crescendo e evoluindo, as especificações principais e mais básicas são (Karris, 2009):

- Estrutura da MIB para o gerenciamento de redes baseadas em TCP/IP — RFC 1155, que descreve como são definidos os objetos gerenciáveis contidos na MIB (Rose e McCloghrie, 1990);
- Base de informações para o gerenciamento de redes baseadas em TCP/IP — RFC 1213, que descreve os objetos gerenciáveis contidos na MIB-II (McCloghrie e Rose, 1991);
- Definição dos objetos gerenciados na MIB e do protocolo usado para a gerenciamento dos objetos — RFC 1157 (Case *et al.*, 1990).

Cada sistema em uma rede (estação de trabalho, servidor, roteador e outros equipamentos) mantém uma MIB que reflete os estados dos recursos gerenciáveis do sistema. Uma entidade de gerenciamento de rede pode monitorar os recursos de um sistema específico, através da leitura dos valores dos objetos na MIB, e controlar os recursos deste sistema, modificando esses valores. Um agente de gerenciamento, tem uma lista de objetos que ele gerencia, como por exemplo, o estado de uma interface de um roteador.

Os objetos da MIB são organizados em uma árvore, apresentada na Figura 3.9. A hierarquia a partir de uma árvore é a base para a nomeação dos objetos usados pelo

SNMP. O SNMP segue a forma da estrutura da informação do gerenciamento (*Structure of Management Information* — SMI), especificada no RFC 1155 (Rose e McCloghrie, 1990). O SMI identifica os tipos de dados que podem ser usados na MIB e especifica como os recursos de uma MIB são representados e nomeados (Karris, 2009). O propósito do SMI é simplificar a MIB, permitindo escalabilidade (Stallings, 1999), ou seja, identificar os objetos de maneira fácil e permitir que novos objetos sejam adicionados.

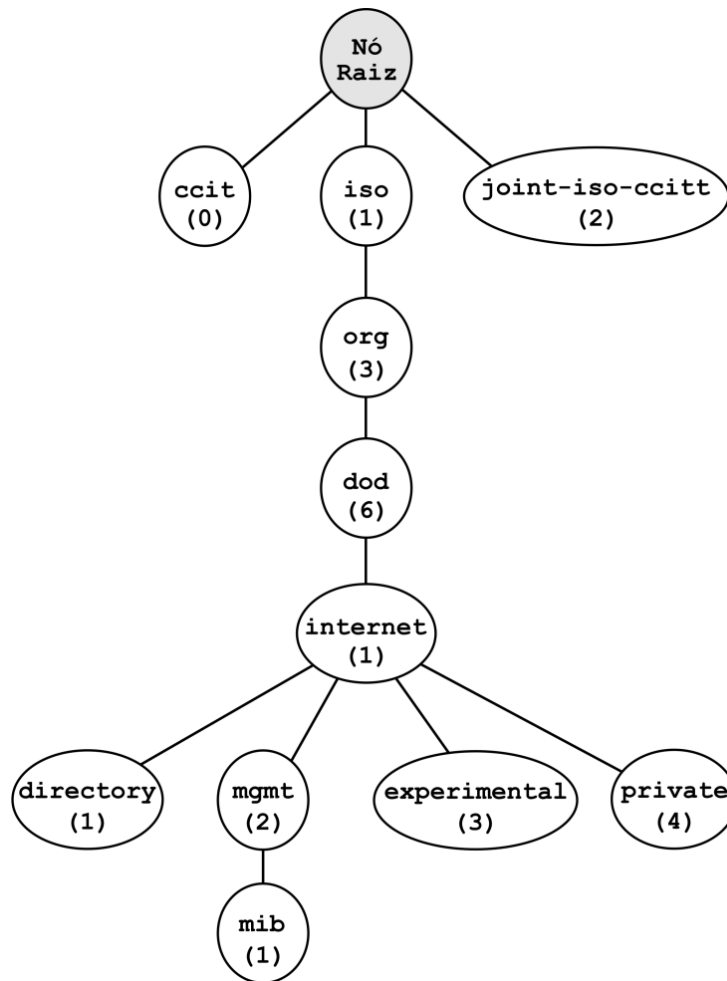


Figura 3.9: Árvore de objetos SMI. Fonte: Adaptado de Stallings (1999).

O nó no topo da árvore de objetos SMI é a raiz, o restante abaixo do nó raiz, mostrado na Figura 3.9, é uma subárvore, contendo alguns objetos da MIB. As folhas da árvore são os objetos que de fato são gerenciados, onde cada um deles representa um recurso, atividade, ou qualquer informação relacionada que será gerenciada (Karris, 2009; Stallings, 1999). Cada objeto na MIB está associado a um identificador do tipo *Abstract Syntax Notation One* (ASN.1), que é chamado de *Object Identifier* (OID).

O identificador OID serve para nomear os objetos da árvore. Além disso, ele também é usado para identificar a estrutura hierárquica dos objetos, na forma de sequência de inteiros separados por pontos. Por exemplo, o OID que identifica o objeto internet é representado como `iso(1).org(3).dod(6).internet(1)`, ou somente `1.3.6.1` (forma

numérica) ou `iso.org.dod.internet` (forma textual). Cada objeto MIB, portanto, recebe um OID numérico e uma forma textual associada.

As subárvores `ccitt(0)` e `joint-iso-ccitt(2)`, da árvore de objetos SMI, não estão relacionadas ao protocolo SNMP. As subárvores do nó `1.3.6.1` (objeto `internet`) são detalhadas a seguir. A subárvore `directory(1)` não é usada atualmente. A subárvore `mgmt(2)` define um conjunto padrão de objetos de gerenciamento de Internet, os objetos da MIB estão listados dentro dessa subárvore.

Atualmente, existem duas versões de MIB aprovadas pelo *Internet Activities Board* (IAB), a `mib-1` e a `mib-2`, esta, sendo uma extensão da primeira. Ambas possuem o mesmo OID, mas somente uma está presente (ativa) em qualquer configuração. A subárvore `experimental(3)` é reservada para propósitos de pesquisas e testes. Os objetos sob a subárvore `private(4)`, são usados por indivíduos e organizações para definir seus próprios objetos (Karris, 2009; Stallings, 1999).

A MIB-II (RFC 1213 (McCloghrie e Rose, 1991)) define a segunda versão da MIB, que estende a MIB-I, com objetos e grupos adicionais. Para incluir objetos na MIB-II, eles devem passar por certos critérios, como por exemplo: um objeto deve ser essencial e sua utilidade deve ser evidenciada; ele não pode possuir variáveis redundantes ou repetidas; restrições de como a implementação deve ser feita; etc.

Como a MIB-II contém somente objetos que foram decididos essenciais, então nem um deles é opcional, ou seja, os fabricantes devem implementá-los em seus equipamentos. O grupo de objetos da subárvore `mib-2` (de OID igual a `1.3.6.1.2.1`), é subdividido em dez grupos, que no geral armazenam informações estatísticas relacionadas ao nome dado ao grupo (Karris, 2009; Stallings, 1999):

- `system(1)`: Informações gerais de sistema, como nome e localização, por exemplo;
- `interfaces(2)`: Informação sobre cada interface do sistema, como interfaces ativas, inativas, rastreamento de bytes enviados e recebidos, por exemplo;
- `at(3)`: Descrição da tabela de tradução de endereços;
- `ip(4)`: Informações relacionadas à implementação e execução do protocolo IP, como dados do roteamento;
- `icmp(5)`: Informações relacionadas à implementação e execução do protocolo *Internet Control Message Protocol* (ICMP);
- `tcp(6)`: Informações do protocolo TCP no sistema, como por exemplo, conexões e falhas;
- `udp(7)`: Informações estatísticas referentes ao uso do protocolo UDP;
- `egp(8)`: Informações dos estados do *Exterior Gateway Protocol* (EGP), como tabela de vizinhos;

- **transmission(9)**: Usado para providenciar um prefixo para o nome dos objetos definidos, com informações sobre as transmissões e protocolos de acesso;
- **snmp(10)**: Informações sobre o SNMP, como o número de pacotes enviados e recebidos;

A estrutura hierárquica que compõe a MIB não foi desenvolvida somente para o SNMP. Ela foi desenvolvida, de fato, pela ISO e faz parte da linguagem de definição de objetos ASN.1, sendo adotada pelo IETF como padrão para identificação dos objetos gerenciados pelo protocolo SNMP (Stallings, 1999).

3.5 Controle da Largura de Banda

Como já citado, o trabalho proposto nesta dissertação refere-se à criação de um sistema de gerenciamento centralizado, com foco no gerenciamento de desempenho, tanto para o monitoramento quanto para o controle. O controle da rede foca no controle de congestionamento baseado na alocação de recursos. A largura de banda, das interfaces do roteador, é o recurso que é alocado de forma dinâmica, dependendo do tráfego futuro previsto pela rede neural. As interfaces do roteador que necessitarem mais tráfego recebem uma largura de banda maior e mais justa.

A modificação da largura de banda, das interfaces de comunicação de um dispositivo de rede, pode ser realizada de muitas maneiras. Um mecanismo muito utilizado é a disciplina de escalonamento de filas por classes, conhecidas como *classful qdiscs*. Esse tipo de mecanismo é muito útil quando é necessário diferenciar tipos distintos de tráfego, de tal forma que tráfegos diferentes recebem tratamentos diferentes (Astuti, 2003; Hubert, 2012). Dentre essas disciplinas de escalonamento, duas se destacam: *Class-Based Queueing* (CBQ) e *Hierarchical Token Bucket* (HTB).

A disciplina de escalonamento CBQ é bastante utilizada e é a mais conhecida dentre as disciplinas de escalonamento baseadas em classes. Para qualquer disciplina de escalonamento de tráfego de rede baseada em classes, os dados do tráfego são classificados antes de serem enviados. Desse modo, cada pacote é classificado em uma classe, de acordo com suas características.

Cada classe CBQ possui uma fila diferente, com prioridades também diferentes. Durante o escalonamento, cada fila é atendida durante seu tempo predeterminado, que é definido pela sua prioridade. Isso faz com que todas as filas sejam atendidas, durante tempos diferentes, de acordo com a política de escalonamento (Astuti, 2003).

Além dos mecanismos básicos de classificação, o gerenciamento usando a disciplina CBQ fornece o compartilhamento da largura de banda para classes que utilizam o mesmo enlace físico (Risso e Gevros, 1999). Esse compartilhamento faz com que as classes que não estão utilizando sua capacidade total de largura de banda, compartilhem sua lar-

gura de banda livre (chamada de “excesso”) com as classes que a precisarem. Assim, a redistribuição da largura de banda melhora a utilização do enlace. No CBQ, as classes são definidas em níveis hierárquicos, ou seja, cada classe pode possuir outras subclasses e cada subclasse também pode ter suas próprias subclasses. Desse modo, a organização das classes no CBQ é feita com uma estrutura de dados em árvore.

O CBQ utiliza dois escalonadores, o escalonador geral e o escalonador de compartilhamento. O escalonador geral garante a largura de banda apropriada para cada classe nas folhas da árvore (Risso e Gevros, 1999). O escalonador de compartilhamento, distribui o excesso de largura de banda para a estrutura de compartilhamento.

O escalonador geral da disciplina CBQ pode variar dependendo da implementação, ele pode ser um escalonador simples, como o *Packet Round Robin* (PRR), ou um escalonador mais complexo, como o *Weighted Round Robin* (WRR). Já o escalonador de compartilhamento é mais complexo, primeiramente, ele estima a vazão do tráfego de uma classe. Em seguida, o escalonador coloca um marcador para as classes que estão subutilizando a largura de banda, i.e., ele marca as classes que possuem excesso de largura de banda. O mecanismo que decide qual classe que receberá a largura de banda extra envolve, basicamente, verificar o estado de uso de cada classe e atribuir o recurso extra para a classe que estiver mais próxima do limite, mas isso nem sempre é trivial (Astuti, 2003; Risso e Gevros, 1999).

A disciplina CBQ é muito usada e conhecida porque é a disciplina de escalonamento baseada em classe mais antiga, contudo, isso faz com que ela também seja a mais complexa. O uso do CBQ pode ser bastante complexo devido aos seus parâmetros de configuração, além disso, ele não é otimizado para muitas situações típicas (Hubert, 2012). O *Hierarchical Token Bucket* (HTB) é a técnica de controle de tráfego mais utilizada nos sistemas operacionais Linux (Astuti, 2003), ele trabalha do mesmo modo que o CBQ, entretanto, o HTB é mais robusto e utiliza menos parâmetros em sua configuração.

A disciplina de escalonamento HTB usa um filtro de *Token Bucket* como base para o escalonador geral, que deixa a vazão de dados mais estável e uniforme. O escalonador de compartilhamento é bem simples, cada classe que precisa de uma largura de banda extra, indica para sua classe pai essa necessidade (Brown, 2006). Assim, se a classe pai possuir esse recurso extra, então ela empresta o excesso para sua subclasse.

O controle da largura de banda com o HTB é feito a partir das classes criadas. Cada classe tem vários parâmetros, como: largura de banda fixa; largura de banda máxima que pode receber emprestado; prioridade da classe; classe pai; etc. A decisão de qual classe um pacote pertencerá é feita através de filtros.

Os filtros do HTB atribuem os pacotes para suas respectivas classes. Os pacotes são enfileirados de acordo com suas classes e o escalonador decide por quanto tempo cada classe é atendida, quais pacotes terão preferência e a vazão máxima que os pacotes serão transmitidos (Astuti, 2003; Hubert, 2012). A Figura 3.10 exemplifica a disciplina de

escalonamento HTB.

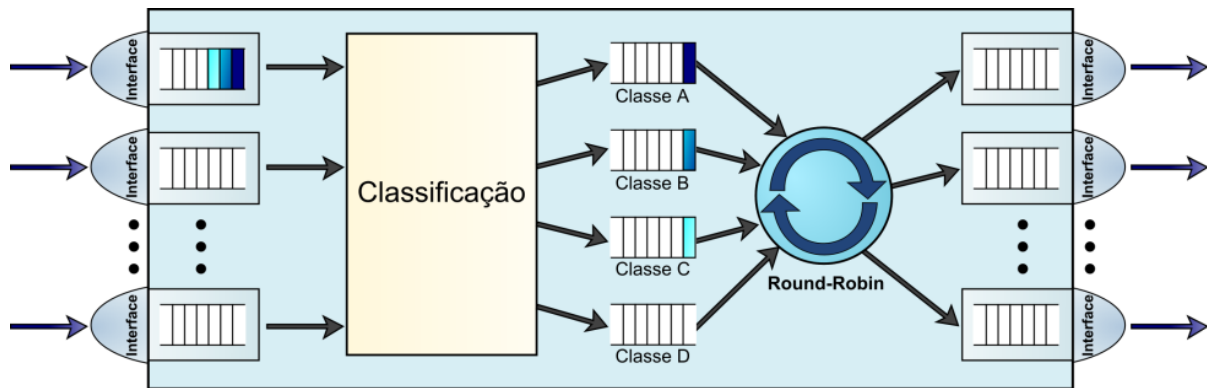


Figura 3.10: Disciplina de escalonamento HTB.

O controle da largura de banda proposto nesta monografia é realizado pela disciplina HTB. O principal motivo dessa escolha é que o HTB possibilita a distribuição seletiva da largura de banda de cada interface de comunicação do roteador, de forma mais simples e eficiente do que o CBQ.

Redes Neurais para a Predição de Tráfego

As redes neurais oferecem as mesmas funcionalidades dos neurônios do cérebro humano, o que as deixa capazes de aprender e resolver problemas complexos. Uma RNA é formada a partir de uma combinação de unidades simples de processamento, que são fortemente conectadas. Essas unidades simples são os neurônios da rede neural, eles são organizados em camadas, onde cada camada pode ter um ou mais neurônios. As RNAs são muito utilizadas para o aprendizado de máquina e reconhecimento de padrões. Os neurônios interconectados recebem os dados de entrada e armazenam esse conhecimento em seus pesos sinápticos.

O conhecimento da rede neural é adquirido de forma iterativa através do processo de aprendizagem, que envolve o algoritmo de treinamento (Haykin, 1998). Após o treinamento, a RNA já aprendeu os dados esperados e está pronta para generalizar seus conhecimentos a partir de dados que não foram usados no treinamento, ou seja, a RNA usa o conhecimento armazenado para inferir novos dados. Com isso, neste trabalho, a rede neural estará pronta para o uso, fazendo previsões do tráfego que passa no roteador de uma rede de computadores.

As RNAs são boas candidatas para a previsão de séries temporais, como o tráfego de Internet, graças às suas características inspiradas no modelo de aprendizagem do cérebro humano, como (Cortez *et al.*, 2012; Haykin, 1998):

- a capacidade de aprender problemas não lineares;
- trabalhar com alto paralelismo entre neurônios;

- robustez para se adaptar a dados diferentes, com tolerância a erros e ruídos não esperados;
- aprendizagem por experiência e habilidade de generalização dos dados, onde as características dos dados são aprendidas em vez de memorizadas.

Para aplicações adaptativas, o tempo de resposta é um fator muito importante. Além do tempo de resposta, o método de predição para esse tipo de aplicação, deve levar em consideração o horizonte de predição, o custo computacional e o erro de predição. A principal vantagem das RNAs para o sistema de gerenciamento adaptativo de largura de banda proposto é o tempo de resposta da predição da rede neural. Embora o processo de aprendizagem seja uma etapa lenta, após a aprendizagem a rede RNA está pronta para ser usada, calculando os resultados da previsão de modo muito rápido. As RNAs, portanto, são boas para predição em curto prazo e em tempo real, já que elas calculam um resultado satisfatório, em relação a precisão da predição, em um curto tempo de resposta (Cortez *et al.*, 2012; Feng e Shu, 2005).

O objetivo deste capítulo é apresentar o estudo comparativo dos modelos de predição (MLP, RNN e SAE) e dos algoritmos de aprendizagem (*Backpropagation*, RPROP e treinamento guloso). Esse estudo, envolve a análise de experimentos feitos com esses modelos de redes neurais, comparando a arquitetura da RNA, a complexidade do modelo, o tempo de treinamento, a precisão da predição e a capacidade de generalização dos dados. Por fim, os resultados de cada modelo de predição são avaliados, sendo que o melhor deles, foi escolhido como modelo de predição para o sistema de gerenciamento adaptativo de largura de banda.

A seleção dos modelos de predição foi feita com o objetivo avaliar tanto abordagens mais simples e tradicionais, como a MLP e RNN, quanto abordagens mais novas e complexas de aprendizagem profunda, como o SAE e as MLPs mais profundas. A análise enfatiza predições de curto prazo. Os testes utilizaram amostras de séries temporais de tráfego de Internet obtidas na base de dados DataMarket (Hyndman, 2014).

4.1 Predição com Redes Neurais

Neste trabalho, as redes neurais são utilizadas para a predição do tráfego de Internet. Para tal, o tráfego é coletado na forma de uma série temporal, que é passada para o treinamento da RNA. O processo de aprendizagem de uma rede neural é feito de forma iterativa, no qual em cada iteração, uma informação é passada para os neurônios de entrada da RNA, em seguida, a rede neural extrai as abstrações e características desses dados de entrada e obtém uma saída.

Uma iteração do processo de aprendizagem é chamada de época ou período de treinamento. No treinamento da RNA, as épocas são repetidas, passando vários dados de

entrada diferentes, até que o erro obtido pela rede neural seja o menor possível. Outro critério de parada do treinamento é o número máximo de épocas, ou seja, o treinamento da rede neural será concluído ao atingir o número máximo de épocas ou quando o erro final atingir um valor desejado.

Para a predição de séries temporais, como o tráfego de Internet, o aprendizado da RNA utiliza os valores dos históricos passados referentes ao tráfego. Para cada época de treinamento, um pequeno grupo de dados sequenciais da série temporal é passado como entrada da rede neural. Essa sequência de dados de entrada é chamada de janela de tempo.

A extração de conhecimentos (aprendizagem) da RNA é feita a partir das características e correlações entre os dados da janela de tempo, para aplicação na previsão do valor futuro. Este valor futuro é o alvo da predição de horizonte $h = 1$, que é o elemento imediatamente após a janela de tempo. Para predições de horizonte $h > 1$, o alvo da predição são os h elementos imediatamente após a janela de tempo.

Para treinar a série temporal completa é utilizado uma janela deslizante, de tal forma que para cada época de treinamento, a janela de tempo desloca-se uma unidade adiante no tempo. Então, o algoritmo de treinamento é aplicado a cada época para uma janela de tempo diferente. A Figura 4.1 mostra o exemplo de uma janela de tempo de uma rede neural com horizonte de predição igual a 1.

No exemplo da Figura 4.1, a janela de tempo é composta de 5 valores, pois a rede neural possui 5 neurônios de entrada. Já o valor previsto, ou seja, o valor esperado da saída da RNA é o próximo valor que está fora da janela de tempo, pois a rede neural possui 1 neurônio de saída. Os valores de entrada são os valores observados nos tempos $f(x_1), f(x_2), f(x_3), f(x_4), f(x_5)$ e o valor de saída esperado é $f(x_6)$.

Para avaliar os resultados do treinamento da RNA, a série temporal é comumente dividida em duas partes: conjunto de treinamento e conjunto de testes. O conjunto de treinamento é uma parte da série temporal que é utilizada unicamente para o treinamento da rede neural. Já o conjunto de testes, é utilizado para avaliar a precisão e eficácia da rede neural após o treinamento.

O resultado da execução da rede neural sobre o conjunto de testes é usado para garantir que a rede neural realmente aprendeu as características dos dados e não somente os memorizou (*overfitting*). Isso é garantido caso o conjunto de testes seja diferente do conjunto de treinamento. Caso a RNA treinada se adaptou bem à entradas que não foram treinadas, então pode se dizer que ela obteve um bom grau de generalização dos dados.

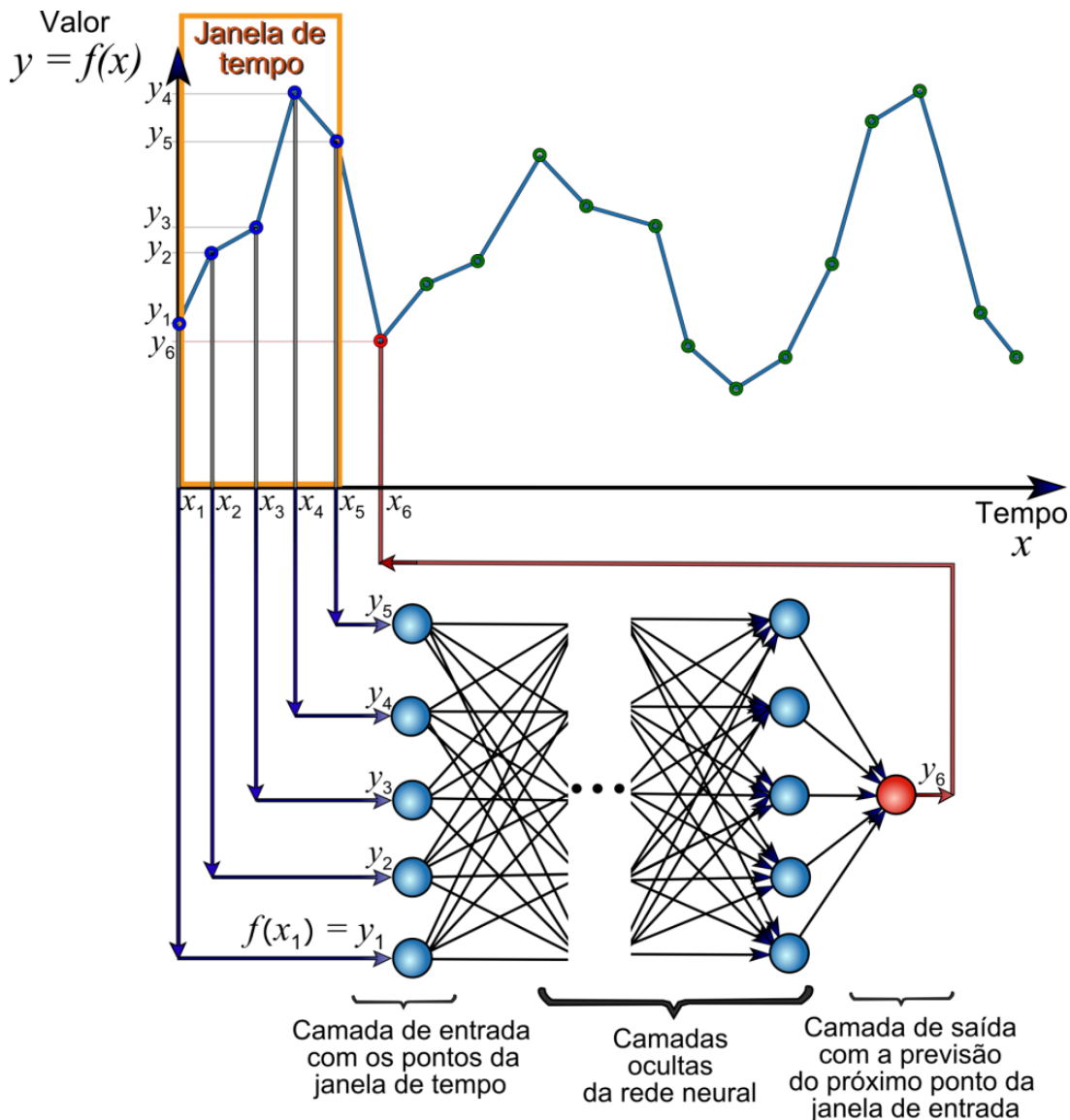


Figura 4.1: Exemplo de uma janela deslizante de previsão de uma série temporal, usando uma rede neural com cinco neurônios de entrada e um neurônio de saída.

4.2 Descrição dos Experimentos Realizados

Os experimentos realizados neste trabalho levaram em consideração três modelos diferentes de redes neurais artificiais. Dentre as RNAs para os experimentos, foram utilizadas:

- *Multilayer Perceptron* usando o *Backpropagation* para o treinamento (MLP-BP);
- *Multilayer Perceptron* usando o *Resilient Backpropagation* (RPROP) como treinamento (MLP-RP);
- *Jordan Neural Network* (JNN) ou Rede Neural de Jordan, usando o RPROP como treinamento;
- *Stacked Autoencoder* (SAE), usando um algoritmo guloso para a aprendizagem profunda.

Os experimentos realizados envolveram séries temporais obtidas no DataMarket e foram coletados por R. J. Hyndman (robjhyndman.com/TSDL/). Os experimentos foram realizados a partir de dados coletados diariamente, a cada hora e a cada intervalo de cinco minutos. Ao todo, seis séries temporais foram usadas, sendo elas: A-1d; A-1h; A-5m; B-1d; B-1h; B-5m. Para todas essas séries temporais foi feito um estudo comparativo da precisão da predição, do tempo de treinamento e da complexidade da RNA, usando as seguintes redes neurais: uma MLP com *Backpropagation*, uma MLP com RPROP; JNN; e SAE.

As séries temporais utilizadas são compostas pelo tráfego de Internet (em bits) de um Provedor de Serviços de Internet (*Internet Service Provider* — ISP) com centros em 11 cidades Europeias. Os dados correspondem a um enlace transatlântico e foram coletados a partir das 06:57 horas no dia 7 de Junho de 2005 até as 11:17 horas em 31 de Julho de 2005. Esta série foi coletada em intervalos diferentes, resultando em três séries temporais diferentes: A-1d é uma série temporal com dados diários; A-1h são dados coletados a cada hora; e A-5m contém dados coletados a cada cinco minutos.

As séries temporais restantes são compostas pelo tráfego de Internet de um ISP, coletado em uma rede acadêmica no Reino Unido. Os dados foram coletados entre 19 de Novembro de 2004, as 09:30 horas, até dia 27 de Janeiro de 2005, as 11:11 horas. Da mesma maneira, essa série foi dividida em três outras diferentes: B-1d são dados diários; B-1h são dados coletados de hora em hora; e B-5m, com dados coletados de cinco em cinco minutos. A Figura 4.2 apresenta as séries temporais utilizadas nos experimentos.

Os experimentos foram conduzidos usando o DeepLearn Toolbox (Palm, 2014) e Encog Machine Learning Framework (Encog, 2014). Ambos são de código aberto e possuem bibliotecas que abrangem várias técnicas de aprendizado de máquina e inteligência artificial. Eles foram escolhidos, pois são muito usados na comunidade científica, são de código aberto, possuem uma boa usabilidade e suprem nossas necessidades, tais como as redes neurais MLP, RNN e SAE.

DeepLearn Toolbox é um conjunto de bibliotecas escritas em MATLAB que cobrem várias técnicas tais como RNA, DBN, SAE, *Convolutional Autoencoders* (CAE) e *Convolutional Neural Networks* (CNN). Encog é um *framework* que suporta vários algoritmos de aprendizado de máquina, como algoritmos genéticos, redes Bayesianas, modelo oculto de Markov e RNA. Para o Encog, foi usado a versão Java 3.2.0, mas ele também está disponível para .Net, C e C++.

Para a predição e treinamento da MLP com *Backpropagation* (MLP-BP), da MLP com *Resilient Backpropagation* (MLP-RP) e da JNN foi usado o Encog; já para o SAE foi usado o DeepLearn Toolbox para MATLAB. Os códigos fonte utilizados para os experimentos de predição com as redes neurais artificiais podem ser vistos na Seção A.1 do Apêndice A.

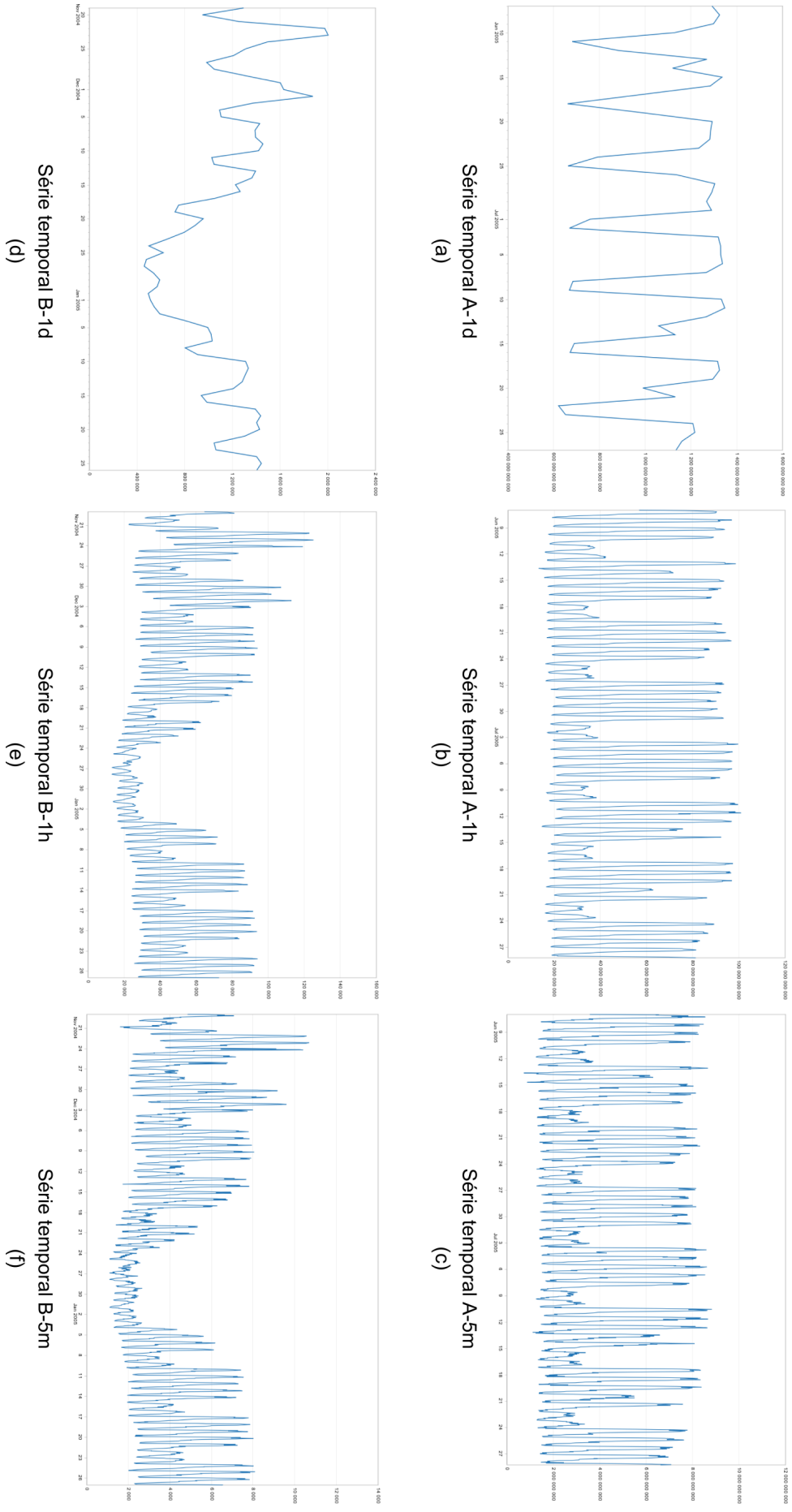


Figura 4.2: Séries temporais com o tráfego (em bits) utilizadas nos experimentos: (a) A-1d; (b) A-1h; (c) A-5m; (d) B-1d; (e) B-1h; (f) B-5m. Fonte: Hyndman, *Time Series Data Library*, datamarket.com.

As configurações do computador onde os experimentos foram realizados são: Notebook Dell, Vostro 3550; Processador Intel Core i5-2430M, 2.40GHz de *clock* e 3Mb de memória Cache); 6GB de memória RAM DDR3; Sistema Operacional Windows 7 Home Basic 64-Bit. A versão da Plataforma Java instalada é a Enterprise Edition com o JDK 7 e a versão do MATLAB é a R2013b.

4.2.1 Análise Prévia do Tráfego de Internet

As propriedades básicas do tráfego de Internet já são bem conhecidas. Estudos mostram os diferentes comportamentos que o tráfego de Internet contém, tais como a auto-similaridade e a dependência de longo prazo (*Long-Range Dependence* — LRD) (Bai e Shami, 2013; Feng e Shu, 2005). Além disso, outra característica são as estruturas não lineares dos tráfegos de Internet (Chen *et al.*, 2012b; Tong *et al.*, 2005). Apesar disso, diferentes tráfegos de Internet podem apresentar características predominantes diferentes, ou seja, um tráfego específico pode exibir alta não linearidade e pouca auto-similaridade, por exemplo. Por esse motivo, encontrar um único modelo de predição que trabalha bem com todos os tipos de tráfego é uma tarefa difícil.

A dependência de longo prazo indica que os valores atuais de uma série temporal recebem muita influência dos valores passados, ou seja, mesmo a longo prazo, os dados passados possuem uma alta correlação com dados mais novos. Esse conceito é muito parecido com o de auto-similaridade e, em muitos casos, os autores os consideram como termos sinônimos (Tong *et al.*, 2005), entretanto, o termo “auto-similaridade” é mais comum em trabalhos estatísticos. Matematicamente, a auto-similaridade se manifesta principalmente quando a autocorrelação da série temporal decai hiperbolicamente em vez de exponencialmente (Bai e Shami, 2013). Em outras palavras, a auto-similaridade indica alta autocorrelação entre as observações da série temporal, ou seja, a série temporal é estritamente não estacionária.

Devido às características de auto-similaridade e não linearidade do tráfego de Internet, as redes neurais são boas candidatas para sua predição, pois elas trabalham muito bem com dados não lineares (Haykin, 1998; Tong *et al.*, 2005) e tem uma alta capacidade de aprendizagem e generalização. Desse modo, as RNAs conseguem aprender as características intrínsecas de um conjunto de dados não linear e inferir novos dados a partir desse aprendizado. Além disso, a aprendizagem das redes neurais depende da autocorrelação dos dados de entrada (Haykin, 1998), por esse motivo, uma rede neural pode aprender mais facilmente séries temporais com auto-similaridade e dependência de longo prazo.

Antes de aplicar o treinamento das séries temporais deste experimento, foi feita uma análise da autocorrelação para cada série temporal, a fim de verificar se as séries tem as características de auto-similaridade, que ajudam no treinamento da RNA. A Figura 4.3 exibe a função de autocorrelação para as séries temporais usadas neste experimento

(A-1d; A-1h; A-5m; B-1d; B-1h; B-5m). Observa-se que a autocorrelação decresce muito lentamente para as séries A-5m e B-5m, indicando um alto grau de auto-similaridade. A autocorrelação das séries restantes incrementa ou decrementa de forma mais rápida que as séries A-5m e B-5m, porém essas mudanças são de características hiperbólicas ou lineares, o que também indica um alto grau de auto-similaridade. Por conseguinte, todas as séries do experimento são aptas para o aprendizado da rede neural.

Como já citado, as redes neurais são modelos de predição não paramétricos, ou seja, a RNA não necessita de parâmetros que descrevem a série temporal. A predição depende unicamente do aprendizado final obtido após o treinamento de cada RNA. A predição será realizada com dados univariados referentes aos bits transmitidos no enlace de comunicação da rede de computadores. As RNAs dos experimentos, portanto, usam séries temporais discretas com somente uma variável preditora, representada pelos bits trafegados na rede.

4.2.2 Normalização e Conjunto de Treinamento

Ajustar os dados do histórico pode levar a um modelo de predição mais simplificado, fazendo com que os padrões se tornem mais consistentes dentro do conjunto de dados. Padrões simples geralmente levam a previsões mais precisas (Feng e Shu, 2005; Hyndman e Athanasopoulos, 2013). Além disso, a normalização é essencial para prevenir que valores altos se sobressaiam mais que valores baixos e para evitar a saturação dos pesos da RNA (Basheer e Hajmeer, 2000). Por isso, antes de treinar a rede neural é importante normalizar os dados da série temporal.

Para diminuir a escala dos dados foi usado a normalização entre mínimo e máximo para limitar os dados no intervalo $[0.1; 0.9]$. Este intervalo foi escolhido para evitar possíveis saturações da função sigmoide, o que afetaria o desempenho do treinamento (Basheer e Hajmeer, 2000). Outro motivo para a escolha desse intervalo é relativa ao limite inferior igual a 0.1, isto é feito para evitar a divisão por zero durante os cálculos com os valores normalizados.

A função de normalização é aplicada sobre os dados originais, gerando uma série temporal normalizada, esta, é usada para o treinamento das RNAs. O conjunto de treinamento usado neste experimento é de 50% do tamanho da série original, como mostra a Figura 4.4. Desse modo, a rede neural é treinada com a primeira metade da série temporal e os testes para validar o treinamento são feitos com a segunda metade da série original. O valor de 50% foi escolhido para balancear o conjunto de treinamento e o conjunto de testes, evitando o *overfitting* e a falta de dados para o treinamento.

A Tabela 4.1 mostra o tamanho total de cada série temporal e o tamanho do conjunto de treinamento para cada uma delas. De todas as 6 séries temporais usadas, seus tamanhos variam entre 51 valores (para a série menor, com dados diários) e 19888 valores (para a série temporal maior, com dados coletados a cada cinco minutos). O tamanho do conjunto

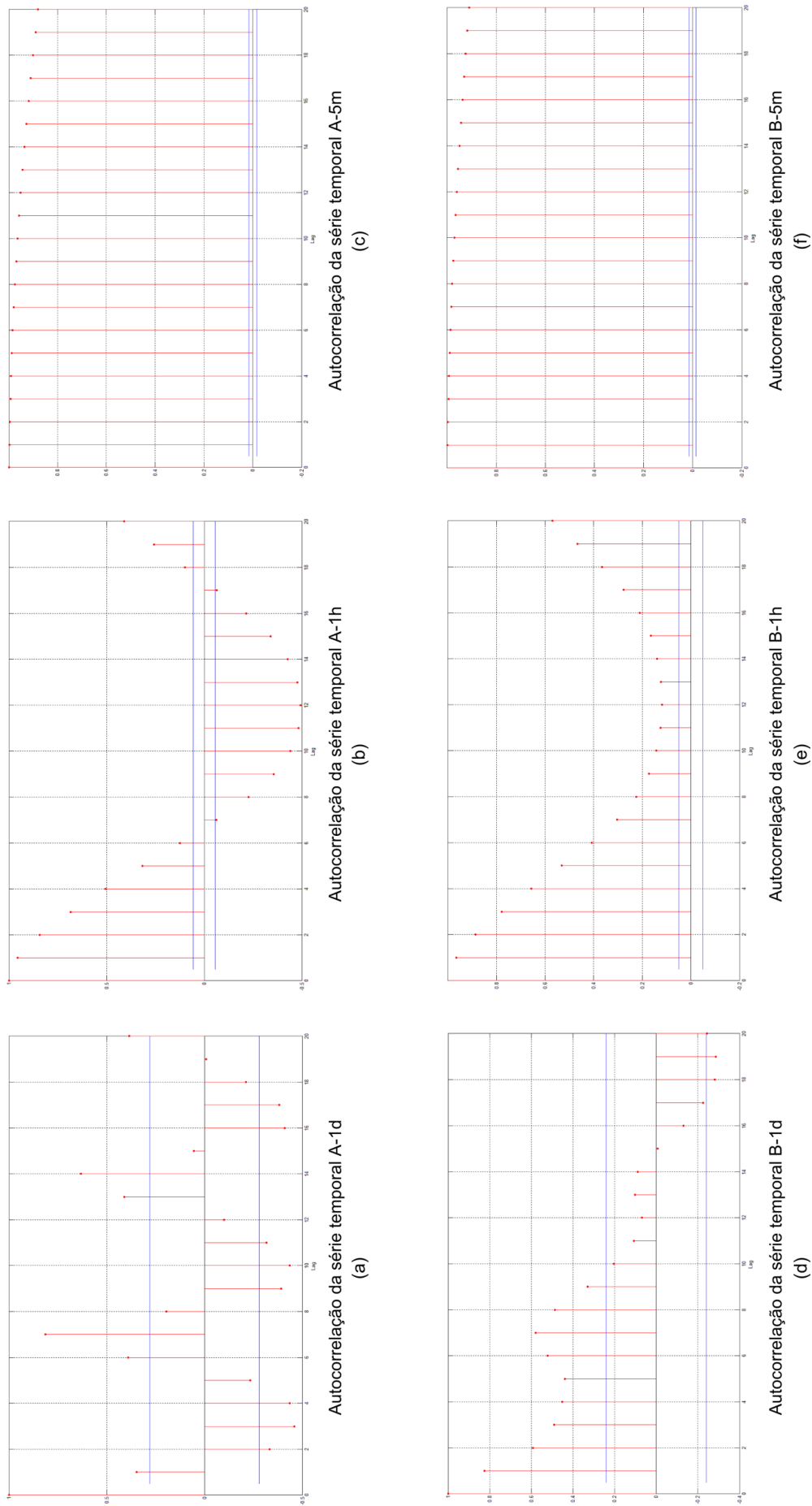


Figura 4.3: Autocorrelação das séries temporais dos experimentos: (a) A-1d; (b) A-1h; (c) A-5m; (d) B-1d; (e) B-1h; (f) B-5m. Fonte das séries temporais: Hyndman, *Time Series Data Library*, datamarket.com.

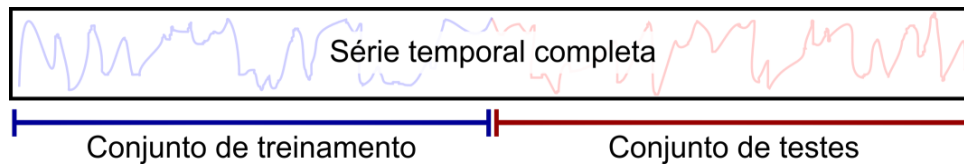


Figura 4.4: Série temporal original dividida em duas partes: conjunto de treinamento e conjunto de testes.

de treinamento também variou bastante, entre 25 e 9944 observações.

Tabela 4.1: O intervalo de tempo e o tamanho de cada série temporal.

Série Temporal	Intervalo de tempo	Tamanho total da série temporal	Tamanho do conjunto de treinamento
A-1d	1 dia	51	25
A-1h	1 h	1231	615
A-5m	5 min	14772	7386
B-1d	1 dia	69	34
B-1h	1 h	1657	828
B-5m	5 min	19888	9944

4.2.3 Execução dos Experimentos

O primeiro passo antes de executar os experimentos foi a análise prévia do tráfego das séries temporais utilizadas nos experimentos. O segundo passo foi a normalização entre $[0.1; 0.9]$ de cada série temporal. Em seguida, foi dividido o conjunto de treinamento e o conjunto de testes de cada série temporal. Finalmente, os experimentos foram realizados, nos quais as redes neurais foram treinadas com o conjunto de treinamento e avaliadas a partir do conjunto de testes.

Os experimentos foram divididos em dois cenários: (a) cenário pontual; (b) cenário exaustivo. O cenário pontual engloba alguns testes pontuais, de tentativa e erro (Basheer e Hajmeer, 2000), para verificar os valores ideais para os parâmetros das redes neurais. Neste cenário os parâmetros analisados foram a taxa de aprendizagem e o tamanho do lote treinado em cada período. O tamanho do lote define a quantidade de dados de entrada que serão treinados em cada época de treinamento.

A taxa de aprendizagem indica a quantidade ajustada no peso dos neurônios em relação ao erro de treinamento, isto é, quanto maior a taxa de aprendizagem, maior é o impacto do ajuste dos pesos em cada período de treinamento. Taxas de aprendizagem mais altas aceleram o treinamento, entretanto podem gerar muitas oscilações nele, dificultando a obtenção de um erro mais baixo. Por outro lado, uma taxa de aprendizagem mais baixa leva a um treinamento mais estável, porém é muito mais lento.

Os testes feitos no cenário pontual verificaram diferentes taxas de aprendizagem e diferentes tamanhos de lote. Os valores explorados neste cenário para a taxa de aprendi-

zagem foram: 0.01; 0.1; 0.25; e 0.5. Para o tamanho do lote de treinamento, foram feitos testes com um único lote treinado por período; lotes com metade do tamanho conjunto de treinamento; e lotes com 10% do tamanho do conjunto de treinamento.

O cenário exaustivo utilizou os melhores parâmetros obtidos no cenário pontual para realizar experimentos exaustivos. Estes experimentos consideraram a complexidade da RNA treinada e o erro de predição obtido nos testes. A complexidade refere-se ao número de neurônios da rede neural e o erro de predição é avaliado pelo erro quadrático médio (MSE) e este erro normalizado (NRMSE). Os códigos fonte do cenário exaustivo são apresentados na Seção A.1 do Apêndice A.

Os experimentos do cenário exaustivo avaliaram o resultado de diferentes complexidades para cada RNA, ou seja, um modelo de RNA foi treinado diversas vezes com diferentes números de neurônios. Como a predição é feita com somente um valor futuro, então foi utilizado somente um neurônio na camada de saída. Cada experimento exaustivo foi replicado 3 vezes para ampliar o espaço de busca dos resultados, pois o treinamento de uma rede neural não garante o resultado ótimo (Haykin, 1998).

A notação $[min : passo : max]$ foi usada para representar a lista com a variação do número de neurônios em uma camada da RNA, onde min e max representam o número mínimo e máximo de neurônios em uma camada e o passo indica o número de incrementos sucessivos dos valores da lista. Por exemplo, o intervalo $[2 : 4 : 10]$ representa a lista de 3 elementos $\{2, (2 + 4 = 6), (6 + 4 = 10)\}$.

4.2.3.1 Experimento Exaustivo 1 — JNN

Os experimentos com a JNN usaram 4 camadas no total, a camada de entrada, uma camada oculta, a camada de saída e a camada de contexto. Devido a camada de contexto e a camada de saída possuírem somente um neurônio elas foram desconsideradas no cálculo da complexidade. Ao todo foram treinadas 400 JNNs diferentes, cada uma delas com uma complexidade diferente. Cada rede neural foi obtida a partir da combinação entre os valores da camada de entrada e da camada oculta.

A distribuição do número de neurônios da camada de entrada e da camada oculta é a seguinte: $[5 : 5 : 100] = \{5, 10, 15, \dots, 90, 95, 100\}$. Ao todo são 20 valores diferentes para cada camada, portanto, a combinação desses valores nas camadas de entrada e oculta gera 400 redes neurais diferentes.

4.2.3.2 Experimento Exaustivo 2 — MLP

A única diferença entre MLP-BP e MLP-RP é o algoritmo de treinamento. Enquanto a MLP-BP utiliza o *Backpropagation* como treinamento, a MLP-RP usa o *Resilient Backpropagation*. Entretanto, ambas utilizaram a mesma quantidade de camadas e a mesma quantidade de neurônios por camada. Este experimento é dividido em 4 partes:

1. MLP-BP com três camadas;
2. MLP-RP com três camadas;
3. MLP-BP com quatro camadas;
4. MLP-RP com quatro camadas.

As redes com três camadas possuem a camada de entrada, uma camada oculta e a camada de saída. As MLPs com quatro camadas possuem duas camadas ocultas. Ao todo foram treinadas 886 MLPs, sendo 443 MLP-BP e 443 MLP-RP. Do mesmo modo que o experimento exaustivo anterior, essas MLPs foram obtidas a partir da combinação entre os valores da camada de entrada e das camadas ocultas. A distribuição dos neurônios para as MLPs foi:

1. MLP com três camadas: $[5 : 10 : 95] = \{5, 15, 25, \dots, 85, 95\}$. São 10 valores diferentes para cada camada. Combinando os valores da camada de entrada com a camada oculta são geradas 100 redes neurais diferentes.
2. MLP com quatro camadas: $[5 : 10 : 65] = \{5, 15, 25, \dots, 55, 65\}$. São 7 valores diferentes para cada camada. Combinando os valores da camada de entrada e das duas camadas ocultas são geradas 343 redes neurais diferentes.

4.2.3.3 Experimento Exaustivo 3 — SAE

Os experimentos com a rede neural SAE foram feitos com até seis camadas no total, ou seja, uma camada de entrada, uma camada de saída e quatro camadas ocultas. Além disso, para comparar a eficiência de redes mais profundas com redes mais rasas, também foram realizados experimentos com SAEs de quatro e cinco camadas. Ao todo foram testadas 637 SAEs diferentes. A distribuição dos neurônios em cada camada para este experimento foi:

1. SAE com quatro camadas: $[20 : 20 : 100] = \{20, 40, 60, 80, 100\}$. São 5 valores diferentes para cada camada. Combinando os valores entre as camadas de entrada e ocultas são geradas 125 SAEs diferentes.
2. SAE com cinco camadas: $[20 : 20 : 80] = \{20, 40, 60, 80\}$. São 4 valores diferentes para cada camada. Então, a combinação desses valores entre as camadas de entrada e ocultas gera 256 SAEs diferentes.
3. SAE com seis camadas: $[20 : 20 : 80] = \{20, 40, 60, 80\}$. São 4 valores diferentes para cada camada. Porém, nesse experimento a camada de entrada teve um valor fixo de 20 neurônios. Isso foi feito para diminuir a quantidade de testes que eram considerados desnecessários. Então, a combinação desses 4 valores diferentes entre as quatro camadas ocultas gera 256 SAEs diferentes.

4.2.4 Apresentação dos Resultados dos Experimentos

Após a execução dos experimentos relativos ao cenário pontual a taxa de aprendizagem e o tamanho do lote de treinamento foram escolhidos. Foram testados valores diferentes para a taxa de aprendizagem, como 0.5, 0.25, 0.1 e 0.01. Como já era esperado, os erros mais baixos foram obtidos usando 0.01 na taxa de aprendizagem, em contrapartida o treinamento é mais lento.

Todas as redes usaram a função de ativação sigmoide, na camada de entrada e nas camadas ocultas, e a função de ativação linear na camada de saída. As redes MLP-BP e SAE usaram uma taxa de aprendizagem igual a 0.01. As redes MLP-RP e JNN não possuem taxa de aprendizagem, pois usam o RPROP como treinamento. O *Backpropagation* foi usado como treinamento supervisionado para a MLP-BP e SAE, contudo, o SAE possui um pré-treinamento não supervisionado que é executado antes do treinamento supervisionado.

Os treinamentos para cada rede neural foram feitos em um único lote. Assim, todos os dados de entrada do conjunto de treinamento são treinados em um único período de treinamento que ajusta os pesos da rede neural para o lote inteiro. Também foram conduzidos testes com mais lotes (menos dados de entrada para cada período de treinamento), porém foram obtidas taxas de erros similares. De qualquer modo, para lotes menores, o treinamento dura mais tempo para convergir, pois produz uma quantidade maior de lotes, fazendo com que uma quantidade menor de dados sejam treinados em cada período de treinamento. Por isso, foi utilizado somente um lote, pois o treinamento tende a ser mais rápido.

4.2.4.1 Treinamento das Redes Neurais para a Predição

O treinamento das redes MLP-BP, MLP-RP e JNN duraram 1000 períodos (ou épocas). O treinamento do SAE é separado em duas etapas. A primeira delas é o pré-treinamento não supervisionado, que durou 900 épocas de treinamento. A segunda etapa é o refinamento, que usa um treinamento supervisionado e que durou 100 períodos de treinamento.

O tempo de treinamento é afetado principalmente pelo tamanho do conjunto de dados que serão treinados e pelo número de neurônios da RNA. Quanto maior o tamanho do conjunto de treinamento e maior o número de neurônios, maior será o tempo necessário para treinar a rede neural. A Tabela 4.2 mostra o tempo de treinamento médio obtido para cada série temporal e o modelo de predição.

A única diferença entre o MLP-BP e MLP-RP é no algoritmo de treinamento. A taxa de aprendizagem dinâmica do RPROP faz com que ele seja um dos algoritmos de treinamento mais rápidos para uma RNA e isso ficou claro nos resultados, o MLP-RP foi mais rápido do que o MLP-BP. A JNN com RPROP foi a rede neural que usou menos

Tabela 4.2: Comparação da média do tempo de treinamento.

Série Temporal	Tempo de treinamento (milissegundos)			
	MLP-BP	MLP-RP	JNN	SAE
A-1d	268	79	67	8874
A-1h	2373	1482	1473	585761
A-5m	86296	56078	33981	6724641
B-1d	558	326	108	17280
B-1h	7322	7181	2838	837567
B-5m	117652	78078	45968	8691876

neurônios e camadas, 3 com exceção da camada de contexto, ao invés de 4 camadas como os outros métodos. De todas as RNA utilizadas, o treinamento mais rápido foi da JNN, pois usou menos neurônios e um algoritmo de treinamento mais rápido.

As 50 primeiras épocas de treinamento e seus respectivos erros são mostrados na Figura 4.5. Nela é comparado o treinamento supervisionado do SAE com o treinamento da MLP-BP. Observa-se que devido ao pré-treinamento do SAE, o seu refinamento converge mais rapidamente do que o treinamento da MLP. Entretanto, mais períodos de treinamento são o suficiente para que elas alcancem taxas de erros mais similares.

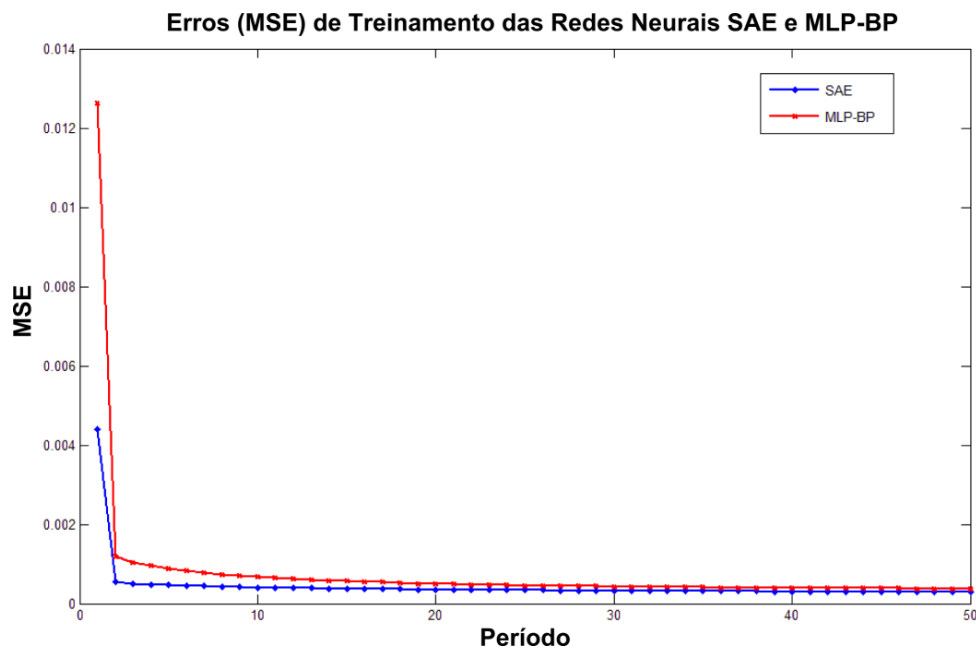


Figura 4.5: Comparação do MSE obtido pelo SAE e pela MLP-BP, durante os 50 primeiros períodos de treinamento, para a série temporal B-5m.

A Figura 4.6 mostra as 50 primeiras épocas de treinamento e seus respectivos erros para a MLP-RP e JNN, que usam o RPROP como algoritmo de treinamento. Uma característica deste tipo de treinamento é a mudança dos erros no início do treinamento, uma vez que não há uma taxa de aprendizagem fixa. Por isso, os erros continuam oscilando até que um valor melhor para a atualização dos pesos seja encontrado.

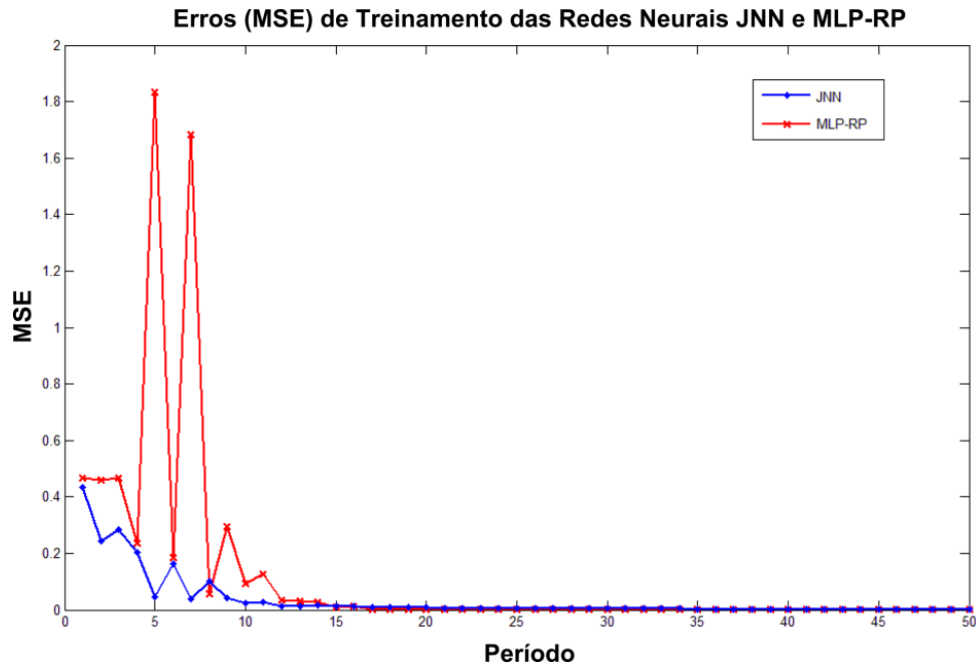


Figura 4.6: Comparação do MSE obtido pela JNN e pela MLP-RP, durante os 50 primeiros períodos de treinamento, para a série temporal B-5m.

As Figura 4.7 e Figura 4.8 mostram os resultados das previsões da JNN e SAE, respectivamente, para a série temporal B-5m. Os gráficos da MLP-BP e MLP-RP foram muito similares em relação ao gráfico da JNN, exibido na Figura 4.7. Devido a baixa escala da imagem é difícil de ver uma diferença entre elas, por isso é mostrado somente o gráfico da JNN. Nota-se que as JNN, MLP-BP, MLP-RP se ajustaram melhor aos dados reais, apesar disso, o SAE também se saiu bem na aprendizagem dos dados. Também é claro que a JNN se saiu muito bem na generalização dos dados, pois a rede foi treinada somente até o tempo 1×10^4 , contudo a JNN ainda conseguiu inferir que haveria um baixo tráfego no período de tempo de 1×10^4 até 1.3×10^4 .

Todos os modelos de previsão utilizados aprenderam as características das séries temporais e usaram essas características para prever os dados que, a priori, não são conhecidos. Um detalhe importante é que a JNN usou menos neurônios e somente uma camada oculta. Portanto, a fase de treinamento dela é mais rápida do que da MLP-BP, MLP-RP e SAE.

4.2.4.2 Arquiteturas e Topologias de cada Rede Neural

Os experimentos realizados utilizaram várias topologias diferentes, foram feitas variações tanto em número de neurônios por camada quanto em número de camadas da RNA. Para a MLP-BP e a MLP-RP, os melhores desempenhos foram obtidos com 4 camadas, por volta de 15 neurônios de entrada, um neurônio na camada de saída, 45 e 35 neurônios nas camadas ocultas, respectivamente, como ilustra a Figura 4.9(a). Verificou-se que aumentar o número de neurônios ou o número de camadas da RNA não resulta em um desempenho melhor, até certo ponto a média do erro normalizado foi parecido para as

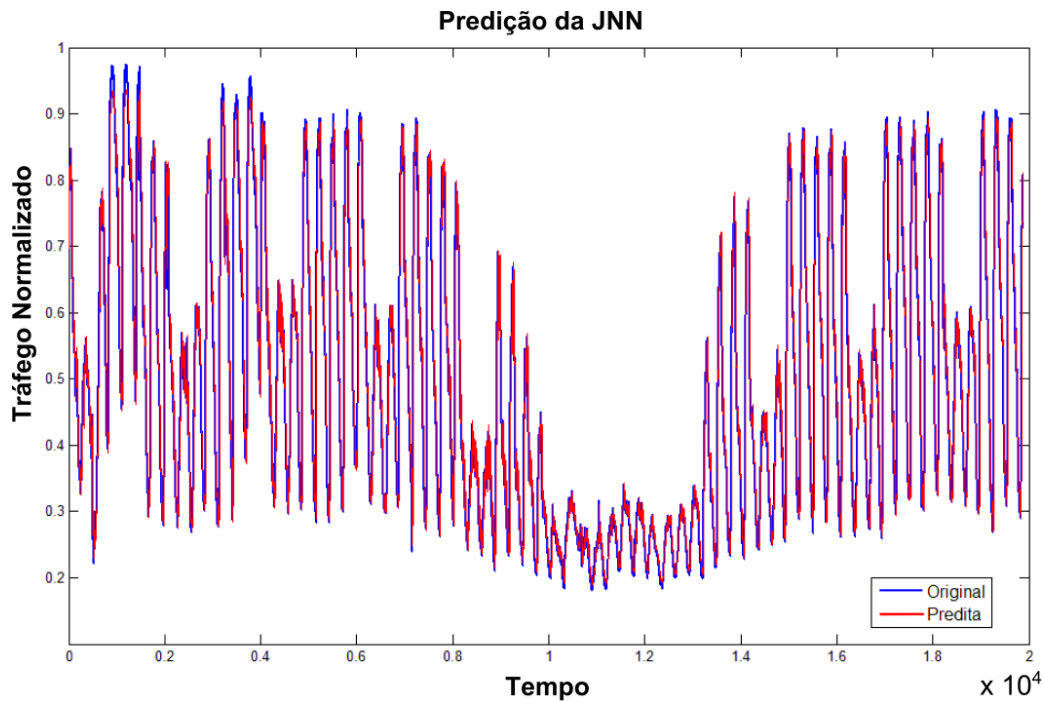


Figura 4.7: Comparação da série temporal Original e a série temporal Predita usando a JNN com uma camada oculta, para a série B-5m. Como o gráfico da série original e predita são muito similares e a escala do gráfico é muito baixa, torna-se difícil observar graficamente as diferenças de cada um. Contudo, nota-se que os valores preditos se encaixam bem com os valores originais.

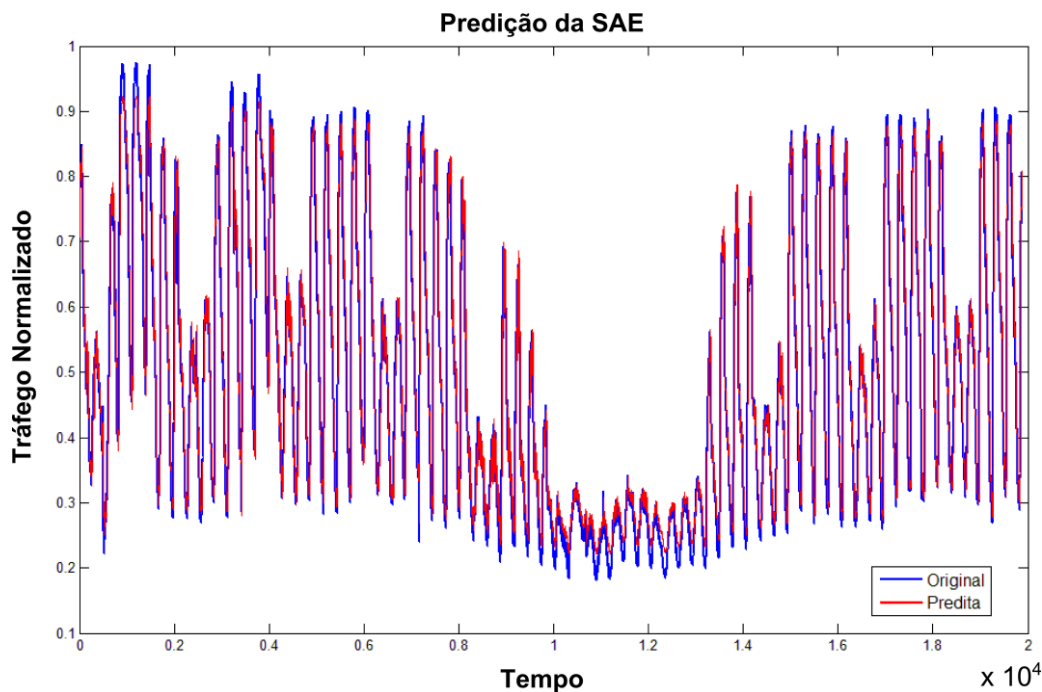


Figura 4.8: Comparação da predição do SAE com 4 camadas ocultas, para a série temporal B-5m, apresentando a série temporal Original e a série Predita. Nota-se que os valores preditos não se encaixam bem a partir de 1×10^4 até 1.3×10^4 na linha do tempo. Porém, para o resto da série, os valores preditos se encaixam bem nos dados originais.

mesmas séries temporais. Mais detalhes podem ser vistos na Figura 4.10.

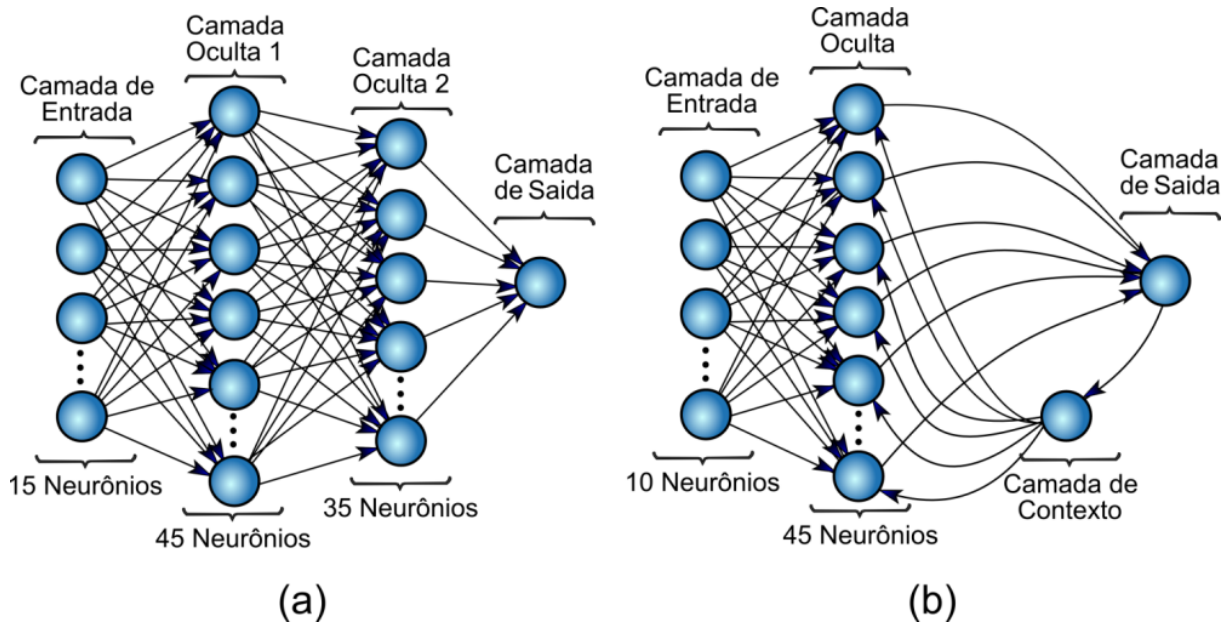


Figura 4.9: Arquiteturas das redes neurais utilizadas, nas quais obtiveram os melhores resultados: (a) Arquitetura da *Multilayer Perceptron* (MLP); (b) Arquitetura da Rede Neural de Jordan (JNN).

As redes MLP-BP, MLP-RP e JNN tiveram as médias do NRMSE mais parecidas, porém a JNN teve uma média de erros mais baixa, mesmo usando uma quantidade bem menor de neurônios. Os melhores desempenhos da JNN foram obtidos com três camadas mais a camada de contexto, por volta de 10 neurônios na entrada, um neurônio na saída e 45 neurônios na camada oculta, como mostra a Figura 4.9(b). Da mesma forma, aumentar o número de neurônios não resultou em um desempenho melhor, na verdade, aumentar muito o número de neurônios foi prejudicial ao desempenho. Os resultados começaram a piorar a partir de 40 neurônios na camada de entrada e 120 neurônios no total, como mostra a Figura 4.10.

Para o SAE, os melhores resultados foram encontrados com 6 camadas, com 20 neurônios na camada de entrada, 1 neurônio na camada de saída, 80, 60, 60 e 40 neurônios em cada uma das camadas ocultas, respectivamente. Aumentar o número de neurônios do SAE não produziu resultados melhores; na média o NRMSE foi semelhante. Resultados similares também foram encontrados com 4 camadas, assim como a MLP; entretanto, SAEs mais profundas atingiram resultados um pouco melhores.

A influência do número de neurônios no erro é mais clara na Figura 4.10. Nota-se que o Erro Quadrático Médio (*Mean Squared Error* — MSE) aumentou significativamente para redes neurais com mais de 120 neurônios. Para um número menor de neurônios, menor que 120, o erro não mudou muito. Além disso, quanto mais neurônios na RNA, mais difícil e demorado será o treinamento. Para o SAE, o MSE foi mais estável, mesmo

para arquiteturas mais profundas. Uma comparação dos NRMSEs de cada modelo de predição é apresentada na Tabela 4.3.

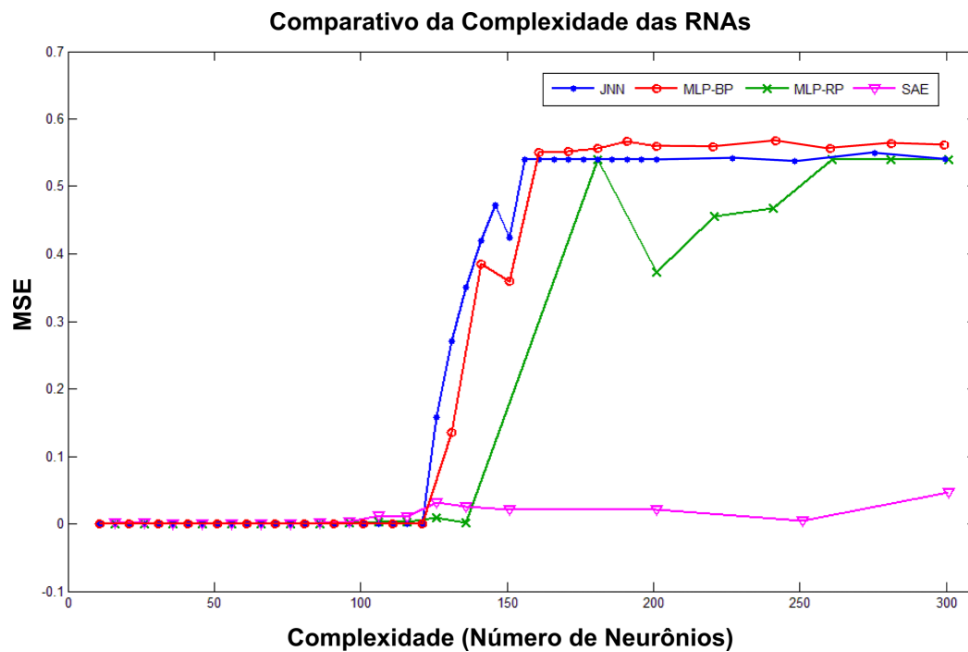


Figura 4.10: Proporção entre a complexidade, medida em número total de neurônios, da rede neural, juntamente com os respectivos erros quadráticos médios (MSE) da a predição da série temporal B-5m.

4.2.5 Análise dos Resultados dos Experimentos de Predição

A ideia principal da aprendizagem profunda é que a profundidade da RNA permite aprender dados complexos e não lineares (Bengio, 2009). Contudo, o uso do SAE para a predição de séries temporais não foi benéfico, i.e., o pré-treinamento não trouxe benefícios significantes para a predição. Os resultados com os respectivos erros normalizados NRMSE são mostrados na Tabela 4.3. O desempenho do SAE foi superado pelos outros métodos.

Tabela 4.3: Comparação dos erros normalizados (NRMSE) obtidos.

Série Temporal	<i>Normalized Root Mean Squared Error</i>			
	MLP-BP	MLP-RP	JNN	SAE
A-1d	0.19985	0.20227	0.19724	0.36600
A-1h	0.05524	0.04145	0.04197	0.09399
A-5m	0.01939	0.01657	0.01649	0.02226
B-1d	0.12668	0.14606	0.11604	0.21552
B-1h	0.04793	0.02927	0.02704	0.06967
B-5m	0.01306	0.01008	0.00994	0.01949

Na predição de séries temporais, que usa dados rotulados, o método SAE possui mais complexidade do que os outros usados, já que ele possui uma etapa extra com o treinamento não supervisionado, que inicializa os pesos da rede neural para a etapa de refinamento. Mesmo com a complexidade adicional, o SAE foi inferior. Por causa disso, a abordagem com o SAE não é recomendada para a predição de séries temporais.

Entre a MLP-BP e a MLP-RP não houve diferenças significativas, mas no geral a MLP-RP obteve resultados melhores tanto em precisão quanto em tempo de treinamento. Por fim, comparando todos os resultados mostrados na Tabela 4.3, nota-se que a JNN obteve os melhores resultados com os menores NRMSEs. Além disso, a JNN trabalhou melhor com menos neurônios, fazendo com que o treinamento seja muito mais rápido, como mostrado na Tabela 4.2.

Uma classificação comparando os resultados de tempo de treinamento e precisão da predição é apresentada na Tabela 4.4 e Tabela 4.5, respectivamente. Observa-se que a JNN foi a rede com o treinamento mais rápido para todas as séries temporais, tendo um ganho significativo para as séries A-5m, B-1d, B-1h e B-5m. A MLP-RP foi a segunda mais rápida, com a MLP-BP logo em seguida. A SAE foi a que teve o treinamento mais lento, porém a grande diferença no tempo de treinamento pode ter ocorrido pela diferença do *framework* utilizado para os testes. Pois, enquanto as redes neurais JNN, MLP-BP e MLP-RP foram treinadas com o Encog (versão Java), o SAE foi treinado usando o DeepLearn Toolbox do MATLAB.

Tabela 4.4: Ranking das redes neurais mais rápidas para cada série temporal.

Ranking	Porcentagem de aumento do tempo					
	A-1d	A-1h	A-5m	B-1d	B-1h	B-5m
1º) JNN	—	—	—	—	—	—
2º) MLP-RP	16.42%	0.61%	94.45%	201.85%	153%	69.85%
3º) MLP-BP	300%	61.1%	153.95%	416.66%	158%	155.94%
4º) SAE	13144%	39666%	19689%	15900%	29412%	18808%

Em relação aos erros normalizados (Tabela 4.5) a JNN obteve os menores erros, com exceção da série A-1h. Porém, para as séries A-5m, B-5m, A-1d, A-1h, pode-se considerar um empate técnico entre a MLP-RP e a JNN, pois o aumento de porcentagem do erro não foi muito significativo (menor que 3%). As redes neurais MLP-BP e MLP-RP obtiveram resultados semelhantes, porém a MLP-RP se saiu um pouco melhor. A MLP-BP obteve melhores resultados, quando comparada com a MLP-RP, para as menores séries (A-1d e B-1d). Entre todas as redes neurais utilizadas no experimento, a SAE obteve os piores resultados.

Existem trabalhos, no campo de reconhecimento de padrões, que mostram que o uso de *Autoencoders* é vantajoso (Palm, 2014) porque são baseados em dados não rotulados.

Tabela 4.5: Ranking das redes neurais com os menores NRMSEs para cada série temporal.

Ranking	Porcentagem de aumento do NRMSE		
Redes Neurais	A-5m	B-1h	B-5m
1º) JNN	—	—	—
2º) MLP-RP	0.48%	8.24%	1.4%
3º) MLP-BP	17.58%	77.25%	31.38%
4º) SAE	35%	157.65%	96%
Redes Neurais	A-1d	B-1d	
1º) JNN	—	—	
2º) MLP-BP	1.32%	9.17%	
3º) MLP-RP	2.55%	25.87%	
4º) SAE	85.56%	85.73%	
Redes Neurais	A-1h		
1º) MLP-RP	—		
2º) JNN	1.25%		
3º) MLP-BP	33.26%		
4º) SAE	126.75%		

Por outro lado, existem trabalhos na previsão de uso de energia, mostrando que os *Auto-encoders* são piores que RNAs mais clássicas (Busseti *et al.*, 2012), como a MLP e JNN. Reforçando o fato de que cada problema tem um método melhor para resolvê-lo, por isso é importante analisar bem os dados de entrada antes de escolher o método mais apropriado que será utilizado.

4.3 Conclusão do Capítulo

Todas as RNAs analisadas foram capazes de se ajustar e prever o tráfego de rede com uma boa precisão. Entretanto, a inicialização dos pesos da rede neural, através do pré-treinamento não supervisionado, não refletiu em melhorias no caso da predição do tráfego de Internet. Os resultados mostram que as MLPs e as RNNs são melhores que o SAE para a predição de tráfego de Internet. Além disso, a aprendizagem profunda do SAE traz mais complexidade computacional durante o treinamento, então a escolha da MLP ou RNN é mais vantajosa.

Teoricamente, de todas as RNA estudadas neste artigo, o melhor método de predição seria a rede neural recorrente JNN, uma vez que é a rede que usa observações passadas como *feedback* no aprendizado de dados temporais e sequenciais. Conforme os experimentos conduzidos, os melhores resultados, tanto em precisão quanto em tempo de treinamento, foram obtidos pela JNN, uma rede neural recorrente simples. De acordo com os

experimentos realizados e assim como esperado, portanto, o melhor método de predição para previsão em tempo real e a curto prazo é a JNN com o RPROP como algoritmo de treinamento, visto que ela obteve os menores erros em um tempo significativamente menor, como apresentado neste capítulo.

A partir desses resultados, o sistema de gerenciamento adaptativo de largura de banda proposto neste trabalho, usou a JNN como modelo de predição, uma vez que ele é o que tem o treinamento mais rápido, os menores erros e o menor número de camadas e de neurônios por camadas, o que deixa a velocidade de predição mais rápida do que redes mais complexas (com mais neurônios).

Algoritmo de Gerenciamento Adaptativo de Largura de Banda em Roteadores

Uma configuração bastante comum tanto em redes domésticas quanto redes institucionais é uma LAN se comunicando com a Internet (ou uma WAN qualquer). Nesse caso, existe pelo menos um roteador que faz a ligação entre a LAN e a Internet. O papel do roteador no tráfego de dados é fundamental, pois é ele que encaminha os pacotes para os segmentos da rede de computadores. O roteador atua como o *gateway* da comunicação, transferindo e direcionando o tráfego que chega da Internet para as sub-redes locais.

A situação anterior, geralmente, é caracterizada por uma interface de conexão do roteador ligada a WAN e outras interfaces do roteador conectadas com outros computadores e dispositivos da LAN, conforme é apresentado na Figura 5.1. O controle da largura de banda proposto é feito de forma independente para cada interface de conexão do roteador. Os experimentos para verificar a eficiência do gerenciamento foram feitos considerando um fluxo de dados da Internet para três sub-redes locais.

O atraso na transferência de pacotes em um roteador é causado, principalmente, por dois fatores: o tempo de transferência de um pacote em um enlace de conexão física e o tempo que esse pacote permanece nas filas de espera do roteador. Como exemplificado na Seção 3.3.3 (Capítulo 3), o atraso, na transferência dos pacotes em um roteador, aumenta exponencialmente quando a carga, do tráfego de rede, é muito grande e se aproxima da capacidade máxima do roteador. Assim, uma solução para evitar o congestionamento é limitar a largura de banda disponível em uma interface do roteador (Kurose e Ross,

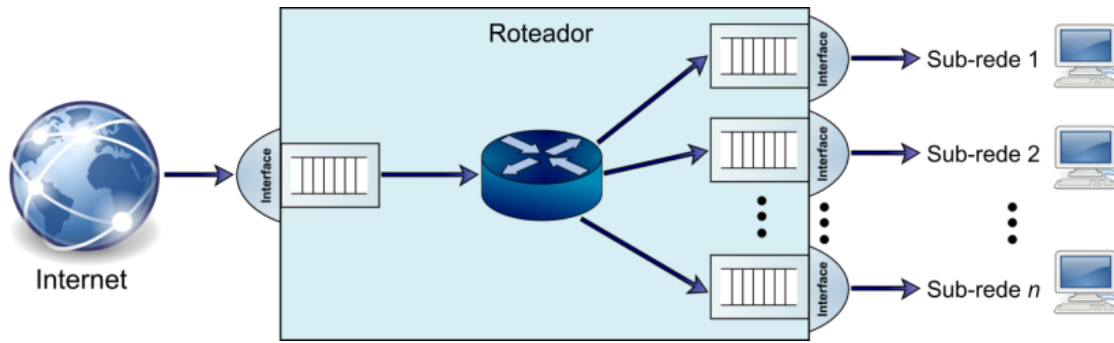


Figura 5.1: Exemplo da conexão do roteador entre a Internet e as sub-redes.

2012; Tanenbaum e Wetherall, 2010). Visando esse caso, foi proposto o algoritmo de gerenciamento adaptativo de largura de banda.

Uma das contribuições deste trabalho é o algoritmo de gerenciamento, que trabalha na alocação da largura de banda de cada interface de conexão do roteador, de forma adaptativa, baseado na previsão do tráfego. Este algoritmo foi chamado de Gerenciamento Preditivo da Largura de Banda usando Redes Neurais (GPLNEURO), sua previsão é realizada por uma rede neural artificial, focando principalmente na redução do atraso como controle de congestionamento. Além disso, o GPLNEURO pode ser utilizado como forma de análise de diferentes tipos de políticas de alocação da largura de banda e de diferentes tipos de previsão de tráfego. Isso é possível pois os módulos do GPLNEURO trabalham de forma independente. Novos módulos de previsão, análise ou controle podem ser acoplados, dependendo da necessidade da rede de computadores.

O modelo de previsão utilizado foi escolhido de acordo com os resultados do Capítulo 4. Com isso, a previsão foi realizada utilizando uma rede neural de Jordan, que é uma rede neural recorrente simples. Além da maior velocidade de treinamento, a JNN atingiu a maior precisão geral, para as séries temporais analisadas. O controle da largura de banda utilizou a disciplina de escalonamento HTB, pois ele possibilita, de uma forma simples, a distribuição seletiva da largura de banda das interfaces de comunicação do roteador.

5.1 Arquitetura do GPLNEURO

O algoritmo de Gerenciamento Preditivo de Largura de Banda atua em roteadores, na alocação adaptativa da largura de banda de suas interfaces de comunicação. O GPLNEURO foi implementado com a linguagem de programação Python, utilizando chamadas do protocolo SNMP para o monitoramento da rede e usando a disciplina HTB para controlar a largura de banda de cada interface do roteador (Hubert, 2012; Stallings, 1999). A linguagem de programação Python (versão 2.7) foi escolhida pois ela providencia facilidades de integração de código e sistemas e é padrão para os sistemas Linux.

Foi utilizada a versão 2 do protocolo SNMP juntamente com a MIB-II, pois os envolvidos no trabalho tem mais experiência com o SNMPv2, além de que ele gerencia todos os

objetos necessários para este trabalho, que estão na versão da MIB-II. Entretanto, qualquer outro protocolo de monitoramento pode ser utilizado, desde que possua a gerência dos objetos mencionados na Seção 5.2.1. Embora a versão 3 do protocolo SNMP tenha um grande ganho em segurança em relação a versão 2, nesta fase inicial do GPLNEURO foi utilizado o SNMPv2 pela sua maior simplicidade comparado ao SNMPv3. Também pode ser utilizado outros protocolos de monitoramento, como por exemplo o *Remote Network Monitoring* (RMON) (Uma e Padmavathi, 2012). Porém, o uso de outros protocolos será focado em trabalhos futuros.

A arquitetura do GPLNEURO foi baseada na arquitetura do gerenciador implementado por Costa (2013). A Figura 5.2 ilustra a arquitetura do GPLNEURO implementado no roteador. O servidor de gerenciamento, que neste caso é o SNMP, mas poderia ser qualquer um outro que faça o monitoramento do tráfego, coleta as informações do tráfego em cada interface de rede do roteador e armazena essas informações na base de dados MIB. O GPLNEURO, então, usa essas informações para prever o tráfego futuro. A partir do tráfego previsto, o GPLNEURO analisa a necessidade de alterar a largura de banda das interfaces de rede. Caso seja identificada a necessidade de alterar a largura de banda, o mecanismo HTB é acionado para limitar a largura de banda das interfaces do roteador.

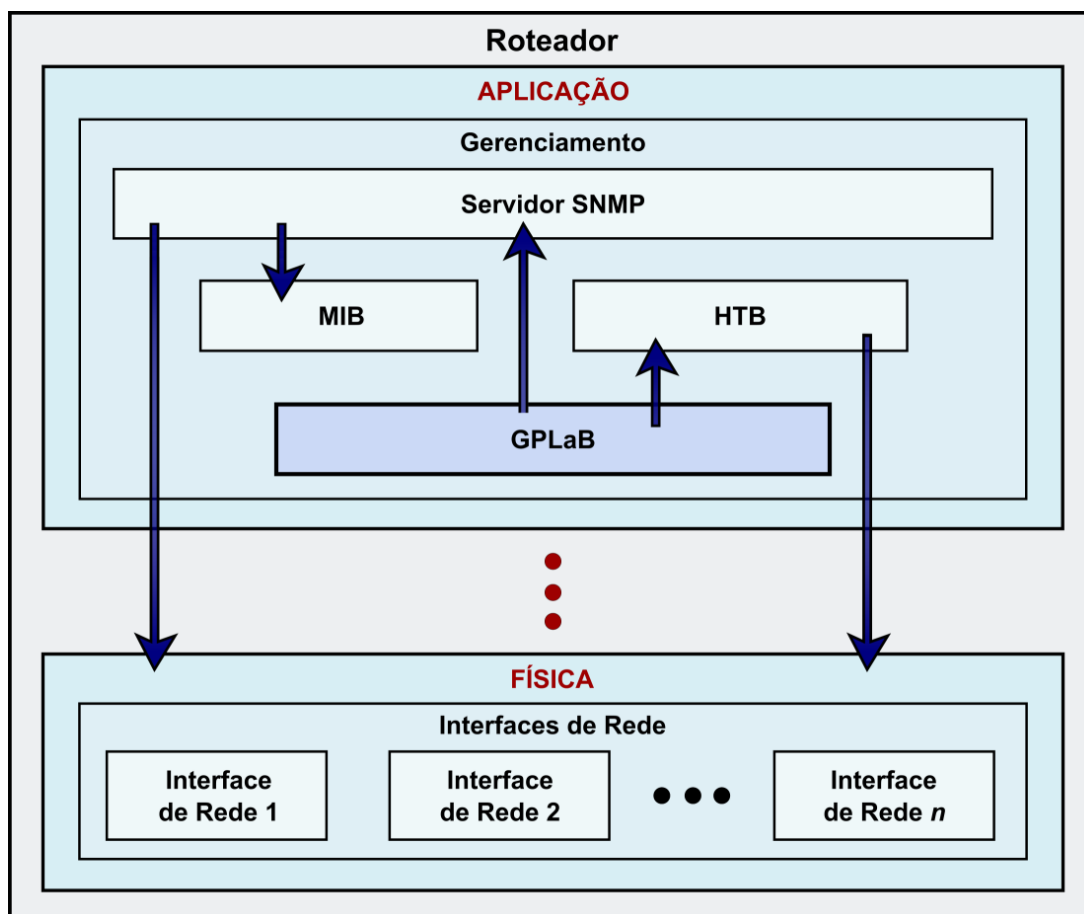


Figura 5.2: Arquitetura do Roteador usando o GPLNEURO.

5.2 Algoritmo GPLNEURO

O algoritmo GPLNEURO possui cinco módulos, onde cada módulo provê uma única função básica e independente. Os módulos do GPLNEURO são: **Monitoramento**, **Treinamento**, **Predição**, **Análise** e **Controle**. Esses módulos foram escolhidos de acordo com as funções de cada componente do gerenciador de largura de banda.

Um sistema de gerenciamento de rede completo possui duas funções, o monitoramento e o controle (Kurose e Ross, 2012). Uma rede neural precisa ser treinada antes de prever os dados da série temporal (Haykin, 1998), por isso foram criados os módulos de treinamento e predição. Com isso, o módulo **Monitoramento**, coleta as informações sobre o tráfego das interfaces de rede do roteador.

O módulo **Treinamento** treina a RNA usando os dados coletados no monitoramento. O módulo **Predição** realiza a predição do tráfego futuro a partir das observações passadas. Regras de decisão de alocação da largura de banda são avaliadas no módulo **Análise**, que decide a largura de banda adequada para cada interface. Finalmente, o módulo **Controle** executa a realocação da largura de banda em cada uma das interfaces de conexão do roteador.

Um diagrama de atividades é apresentado na Figura 5.3. Inicialmente são iniciados paralelamente os módulos **Monitoramento**, **Treinamento** e **Predição**. O módulo **Monitoramento** armazena as séries temporais coletadas em um arquivo para que os módulos **Treinamento** e **Predição** possam realizar suas tarefas.

O módulo **Treinamento** usa as séries coletadas durante o monitoramento para aplicar o algoritmo de treinamento na rede neural. Antes de ser executado, o módulo **Predição** espera a finalização do treinamento. O mesmo acontece com o módulo **Treinamento**, que espera o módulo **Monitoramento** coletar informações suficientes para realizar o treinamento.

O módulo **Predição** usa a última rede neural treinada para prever o tráfego de cada interface do roteador. Os dados de entrada para a rede neural são recolhidos do arquivo armazenado durante o monitoramento. Os resultados da predição são passados para o módulo **Análise** que verifica a predição e define a nova largura de banda.

O módulo **Controle** recebe os valores das larguras de banda de cada interface e aplica a alocação para cada uma delas. Por fim, essas sequências de atividades são repetidas, tornando o gerenciamento adaptativo. Essa sequência de atividades é exibida no Código A.7 no Apêndice A.

5.2.1 Módulo Monitoramento

O módulo **Monitoramento** é executado de forma independente dos outros módulos e é responsável por coletar todas as observações referentes ao tráfego, que serão usadas

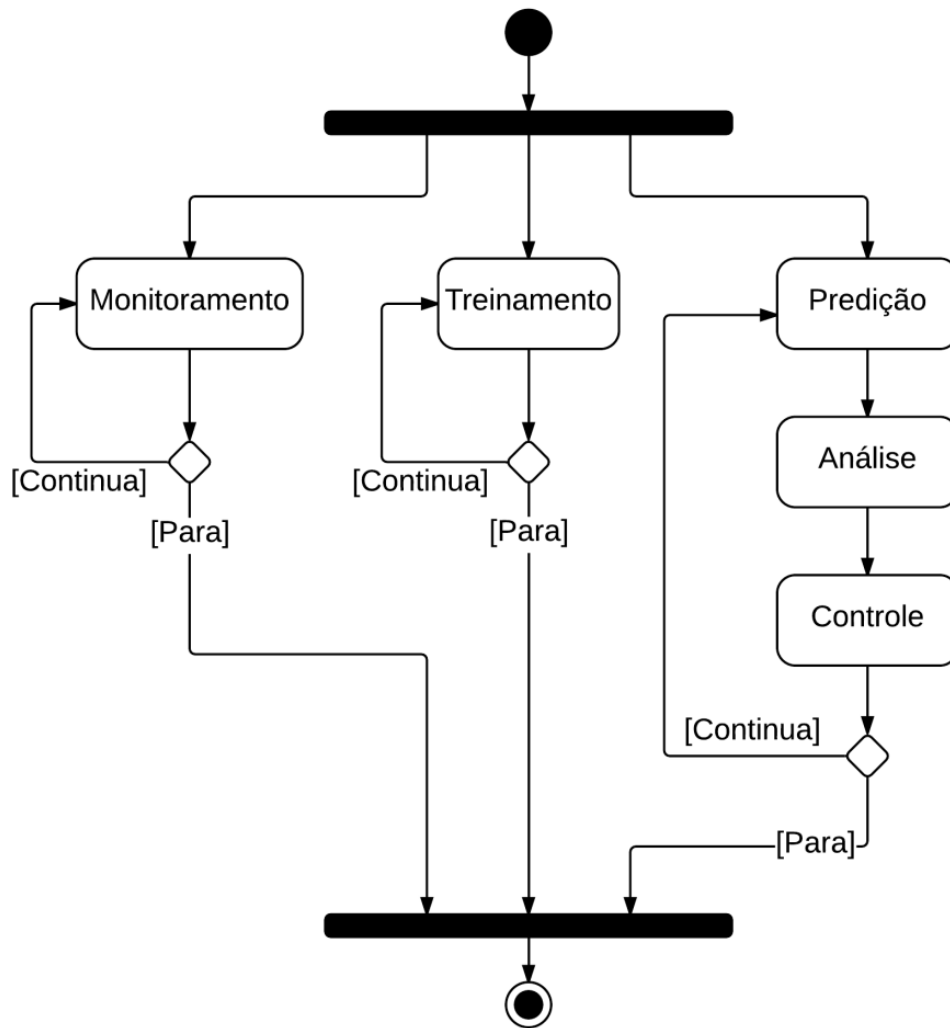


Figura 5.3: Diagrama de atividades do algoritmo GPLNEURO.

nos módulos **Treinamento** e **Predição**. Este módulo possui duas funções, apresentadas na Figura 5.4, a primeira é a leitura dos dados do tráfego das interfaces do roteador, usando o protocolo SNMP; a segunda é o armazenamento dos dados coletados em um arquivo contendo as séries temporais do tráfego das interfaces e o atraso no recebimento dos pacotes. O processo do monitoramento é repetido, atualizando as séries temporais a cada período de tempo pré-determinado.

A primeira etapa do monitoramento, referente à coleta dos dados do tráfego, utilizou o protocolo SNMP, que armazena na MIB diversas informações relativas aos equipamentos em uma rede de computadores. A escolha do SNMP para esse tipo de tarefa se deve pela sua capacidade de escalabilidade e portabilidade (Leinwand e Fang, 1996; Stallings, 1999). Muitos dispositivos de rede saem de fábrica com um agente SNMP já implementado, isso faz com que o algoritmo GPLNEURO possa ser utilizado, em qualquer um desses equipamentos, com alterações mínimas se necessário. Além disso, trabalhos futuros pretendem implementar um sistema de gerenciamento distribuído usando o GPLNEURO como base, para tal, a escalabilidade do SNMP é muito útil.

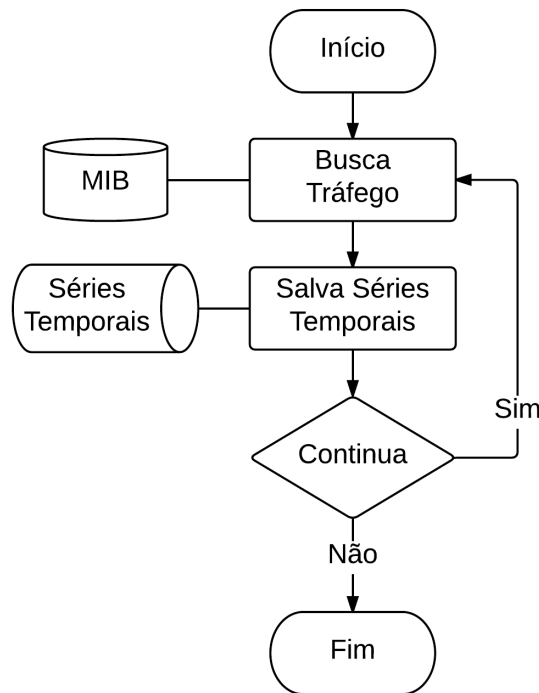


Figura 5.4: Fluxograma do módulo **Monitoramento**.

O SNMP utiliza a subárvore de interfaces IF-MIB para armazenar as informações das interfaces do dispositivo de rede. A IF-MIB é uma subárvore da MIB-II e seu OID é `iso.org.dod.internet.mgmt.mib-2.interfaces (1.3.6.1.2.1.2)` (IF-MIB, 2014). As informações disponíveis na IF-MIB ficam armazenadas na subárvore `ifTable`, que contém dados como: descrição textual sobre cada interface; largura de banda estimada de cada interface; estado de operação; pacotes enviados e recebidos; quantidade de erros e descartes de pacotes; quantidade de octetos (bytes) enviados e recebidos para cada interface.

Para o propósito deste trabalho, o SNMP foi usado para monitorar os objetos `ifInOctets` e `ifOutOctets` (`1.3.6.1.2.1.2.2.1.10` e `1.3.6.1.2.1.2.2.1.16`, respectivamente) da IF-MIB, conforme ilustra a Figura 5.5. O objeto `ifInOctets` armazena a quantidade total de octetos (bytes) recebidos na interface, ou seja, a quantidade de bytes que chegaram, em certa interface, provenientes de uma fonte externa. Por sua vez, o objeto `ifOutOctets` registra o número total de octetos transmitidos para fora da interface. A soma dos valores desses dois objetos resulta na quantidade total de bytes trafegados na interface, tanto bytes que chegam quanto bytes que saem.

Os objetos `ifInOctets` e `ifOutOctets` são do tipo *Counter32*, i.e., são contadores de 32 bits. O tipo contador acumula a quantidade de bytes a cada período de tempo. Assim, esse objeto não armazena a quantidade de bytes que está trafegando, mas sim a quantidade acumulada de bytes que já passou na interface desde a inicialização da MIB. Devido ao contador ser de 32 bits, ele possui um limite de $2^{32} = 4294967296$ valores diferentes, por isso esses objetos armazenam valores de 0 a 4294967295.

O contador de 32 bits acumula valores de forma cíclica, portanto, quando seu valor

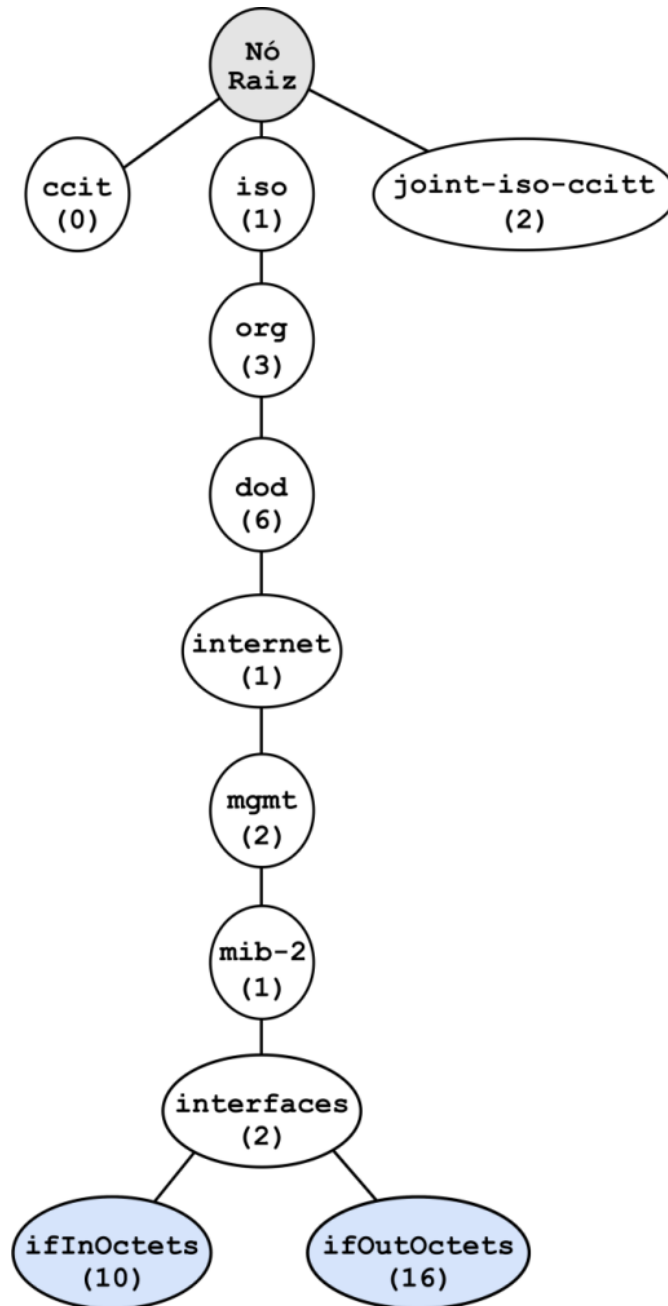


Figura 5.5: Subárvores `ifInOctets` e `ifOutOctets` do gerenciamento das interfaces da MIB. Fonte: Adaptado de IF-MIB e Stallings (1999).

atinge o limite superior seu valor volta para o valor 0, para continuar a contagem. Por isso, o módulo **Monitoramento** deve levar em consideração essa extrapolação de valores. Além disso, para saber a quantidade de bytes que trafegaram em uma interface no instante t , basta subtrair o valor do contador no instante t pelo valor do contador no instante $t - 1$.

O intervalo de tempo entre as coletas de dados no módulo **Monitoramento** foram de 1 minuto. O tempo mínimo do intervalo de atualização dos objetos SNMP (Stallings, 1999) é de 1 segundo, porém para o trabalho proposto, 1 minuto é um tempo suficiente para as predições de curto prazo. O tempo padrão do intervalo de atualização dos objetos da MIB é de 30 segundos. Para alterar esse tempo basta atribuir um novo valor para o

objeto `nsCacheTimeout` (1.3.6.1.4.1.8072.1.5.3.1.2.1.3.6.1.2.1.2.2).

Após coletar os dados do tráfego de cada interface, uma série temporal é criada e armazenada em um arquivo. Esse arquivo é posteriormente usado pelos módulos **Treinamento** e **Predição**. O módulo **Treinamento** usa os dados das séries temporais para treinar uma rede neural para cada tráfego diferente e o módulo **Predição** usa esse arquivo com as séries temporais para pegar os valores de entrada que serão preditos pelas redes neurais. Todo o ciclo do módulo **Monitoramento** pode ser visto no Código A.8 do Apêndice A.

5.2.2 Módulo Treinamento

O módulo **Treinamento**, conforme o fluxograma da Figura 5.6, realiza o treinamento das redes neurais artificiais para a predição do tráfego das interfaces de rede do roteador. O modelo de rede neural utilizado pelo GPLNEURO foi a JNN com o algoritmo de treinamento RPROP. Esse modelo de treinamento foi escolhido pela maior velocidade de treinamento e melhores resultados da predição, quando comparado com os outros modelos estudados no Capítulo 4.

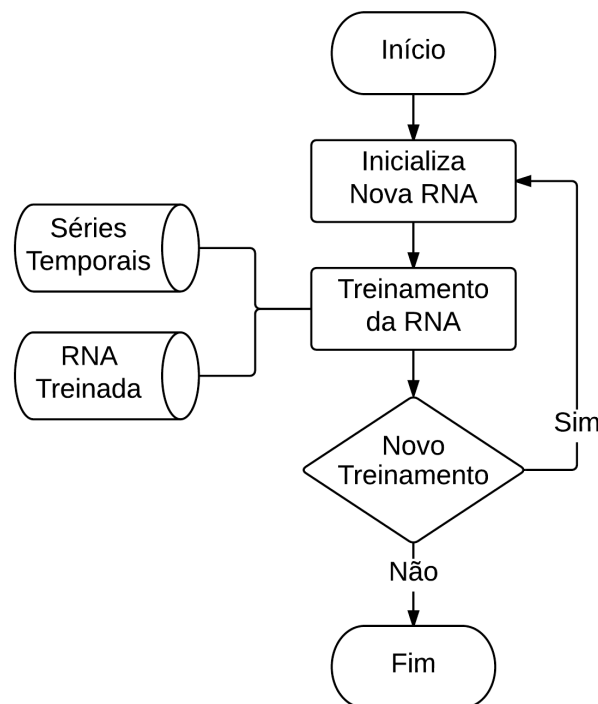


Figura 5.6: Fluxograma do módulo **Treinamento**.

Cada interface do roteador possui seu tráfego próprio, por isso cada interface deve ter a sua rede neural correspondente. Um roteador com um número n de interfaces, portanto, deve ter n redes neurais treinadas com os respectivos tráfegos de suas interfaces. O primeiro passo do treinamento é a inicialização aleatória dos pesos sinápticos das RNAs. O segundo passo, é o treinamento de fato da JNN. Os valores de entrada e saída esperada que

serão passados para o treinamento são as janelas de tempo das séries temporais coletadas e armazenadas no módulo **Monitoramento**. Antes do treinamento, esses valores são normalizados conforme discutido na Seção 4.2.2 (Capítulo 4).

A inicialização aleatória dos pesos da rede neural pode influenciar no resultado final do treinamento (Haykin, 1998), em vista disso, a mesma JNN é treinada cinco vezes e a rede neural com o melhor resultado (menor erro de predição) é escolhida para ser usada no módulo **Predição**. Melhores resultados podem ser encontrados caso as redes sejam treinadas mais do que cinco vezes, entretanto a diferença dos erros pode não ser muito significativa e o tempo para treinar mais redes neurais é proporcional ao número de RNAs treinadas. Após a fase de aprendizagem, as JNNs treinadas são armazenadas para serem utilizadas pelo módulo **Predição**. Para efeito de registro, os erros de treinamento das redes neurais também são armazenados. Os códigos fonte com o treinamento das redes neurais são apresentados no Código A.2 do Apêndice A.

As JNNs que foram treinadas são usadas para a predição durante um período pré-determinado. Ao passar do tempo, a série temporal pode apresentar características que não foram treinadas, por esse motivo é necessário treinar a rede neural novamente após certo tempo. Esse tempo pode ser de meses ou até anos, dependendo das características da série temporal. Estudos futuros pretendem analisar a degradação da precisão da predição com o passar do tempo ou, para contornar esse problema, redes neurais com treinamento dinâmico também podem ser usadas.

5.2.3 Módulo Predição

O módulo **Predição** é responsável por prever a largura de banda que será utilizada pelas interfaces de conexão do roteador. A Figura 5.7 esboça o fluxograma com as três funções básicas deste módulo, que são: (i) inicializar as redes neurais que foram treinadas; (ii) buscar os valores das séries temporais que serão usados como entrada para as redes neurais; e (iii) fazer a predição da largura de banda futura e passar o resultado desta predição para o módulo **Análise**.

O primeiro passo do módulo **Predição** é inicializar as redes neurais que já foram treinadas. As RNAs treinadas estão armazenadas em um arquivo criado pelo módulo **Treinamento**. O segundo passo é buscar as séries temporais que foram armazenadas pelo módulo **Monitoramento**, a partir dessas séries temporais são criadas as janelas de tempo de entrada para a predição. Antes de realizar a predição é necessário normalizar os valores de entrada de acordo com a normalização realizada pela rede neural antes do treinamento. Os valores de entrada normalizados são passados para o preditor, que retorna os valores futuros da largura de banda de cada interface.

Todos os valores previstos são normalizados de volta para a escala original, desse modo, são obtidos a quantidade de bytes que está prevista para trafegar nas interfaces de rede.

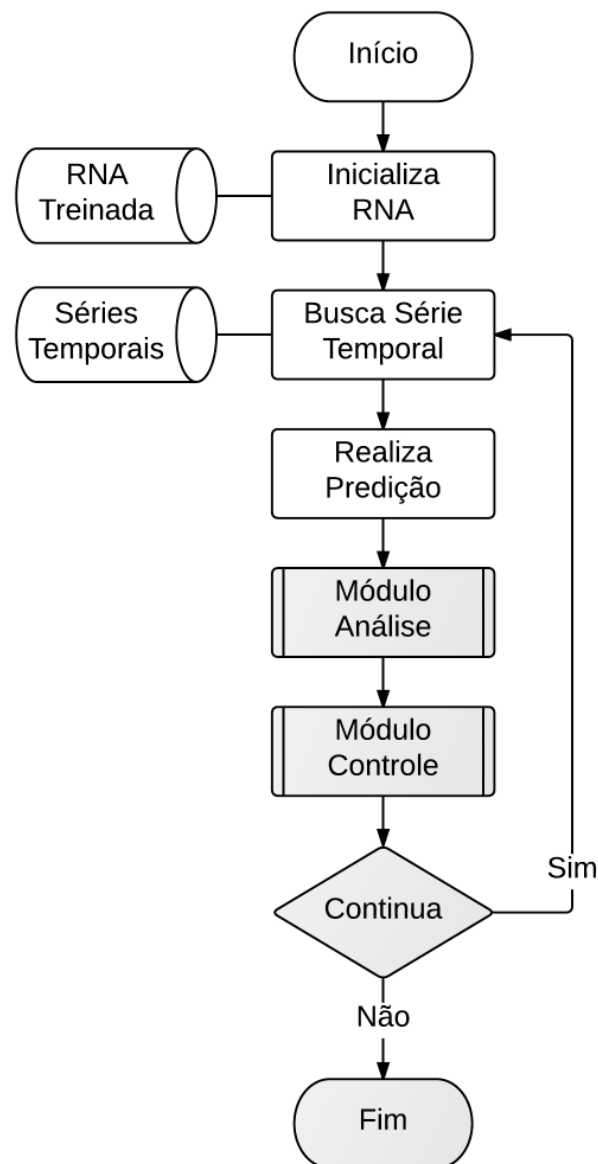


Figura 5.7: Fluxograma do módulo **Predição**.

Como discorrido no Capítulo 4, a JNN foi utilizada para as predições e sua implementação utilizou a versão Java do *framework* Encog. Como o GPLNEURO foi implementado com a linguagem Python, foi necessário criar uma função *wrapper* (empacotadora) para chamar o método de previsão da JNN. Os códigos do módulo **Predição** são expostos no Código A.9 e no Código A.10 do Apêndice A. Por fim, todas as previsões são registradas em um arquivo caso necessite de uma análise da precisão das predições.

5.2.4 Módulo Análise

O módulo **Análise** é responsável por verificar as larguras de banda futura de cada interface e atribuir essas novas larguras de banda de acordo com algumas regras pré-estabelecidas. As regras podem variar dependendo do tipo de alocação que se pretende fazer, diferentes regras de alocação de recursos resultam em diferentes efeitos no controle

de congestionamento. O foco deste trabalho é a distribuição da largura de banda de um modo mais justo durante o tráfego de dados em um roteador, visando a redução geral do atraso da transferência de dados.

O fluxograma referente ao módulo **Análise** é exibido na Figura 5.8, suas atividades são divididas em duas, a (i) verificação das regras de alocação que serão usadas para (ii) definir os valores das larguras de banda de cada interface de comunicação presentes no roteador. As duas atividades são realizadas em conjunto, onde a largura de banda é analisada e caso necessário ela é alterada de acordo com as regras pré-determinadas. As regras de alocação utilizadas no GPLNEURO foram:

1. Exceder o limiar da largura de banda usando o erro médio de predição;
2. Distribuir a largura de banda igualmente para um intervalo de execução do GPLNEURO;
3. Limite inferior da largura de banda de acordo com uma valor mínimo pré-definido;
4. Limite superior da largura de banda de acordo com o limite físico da interface.

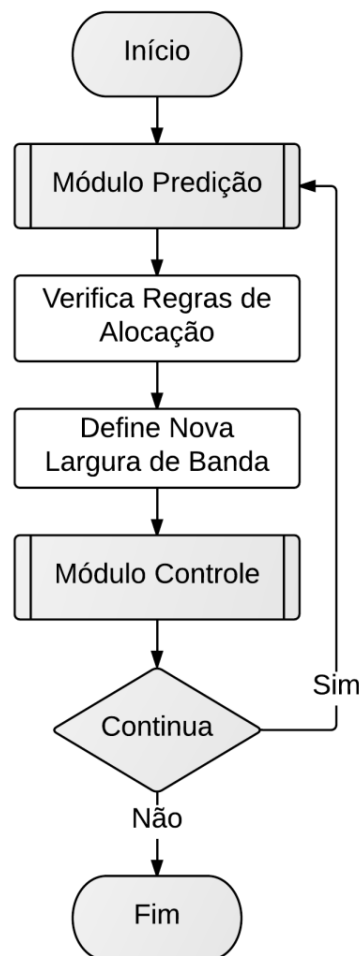


Figura 5.8: Fluxograma do módulo **Análise**.

As quatro regras são aplicadas de forma sequencial. A primeira regra de alocação é baseada no fato de que a predição não é perfeita, ou seja, toda predição tem um erro para mais ou para menos. Para o GPLNEURO o erro para mais não é problemático, pois o gerenciador irá fornecer um pouco mais de largura de banda do que o previsto. Porém, o erro para menos trás problemas de atraso, pois se uma interface receber uma largura de banda menor do que o necessário irá ocorrer uma sobrecarga desta interface, causando um atraso maior no envio de pacotes e até mesmo congestionamentos. Por isso, a fim de evitar esse problema, a largura de banda alocada é excedida a partir do erro médio de predição, este, calculado no módulo **Treinamento**. O excesso de largura de banda como garantia, dado para cada interface, é feito da seguinte maneira:

$$BW' = BW_p + (BW_p * erro_p), \quad (5.1)$$

onde BW' é a nova largura de banda de uma interface; BW_p é a largura de banda prevista; e $erro_p$ é o erro médio de predição da rede neural.

A segunda regra de alocação é influenciada pelo intervalo de execução do GPLNEURO, que é de 1 minuto. Desse modo, o gerenciador faz novas alocações a cada 1 minuto. Para um gerenciador em tempo real, as alocações deveriam ser realizadas a cada 1 segundo, no máximo. Todavia, como o foco deste trabalho é no gerenciador de curto prazo, o intervalo médio de 1 minuto foi escolhido.

A largura de banda predita, portanto, envolve o próximo minuto de tráfego, ou seja, uma predição retorna o valor que será trafegado no próximo minuto de execução. Assim, a largura de banda prevista (bytes por minuto) é distribuída, de forma igual, durante esse minuto. Essa distribuição é feita da seguinte forma:

$$BW'' = \frac{BW'}{60}, \quad (5.2)$$

onde BW'' é a nova largura de banda (depois de aplicada a segunda regra de alocação) convertida em bytes por segundo; e BW' é a largura de banda antiga em bytes por minuto.

As regras de alocação 3 e 4 definem o limite inferior e superior para a largura de banda alocada. O limite superior é o limite físico máximo da largura de banda de cada interface. O limite inferior foi definido como 40% da largura de banda máxima permitida, pois estudos indicam que a taxa de utilização da *ethernet* é em média 40% (Tanenbaum e Wetherall, 2010). O limite inferior força a largura de banda a ser no mínimo 40% de sua capacidade total e o limite superior força a largura de banda a ser no máximo sua capacidade total de largura de banda. O módulo **Análise** é codificado como mostra o Código A.11 do Apêndice A.

5.2.5 Módulo Controle

Após a análise e a definição das novas larguras de banda de cada interface do roteador, essas novas larguras de banda são repassadas para o módulo **Controle**, que executa o controle do tráfego, aumentando ou diminuindo a largura de banda de cada interface. A Figura 5.9 contém um fluxograma com as tarefas executadas pelo módulo **Controle**.

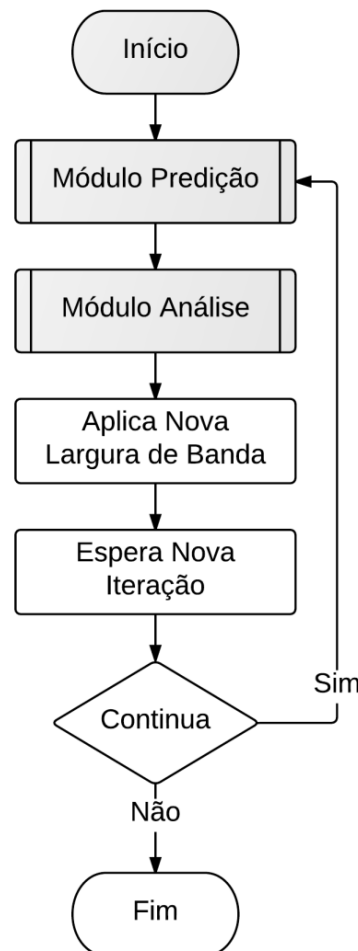


Figura 5.9: Fluxograma do módulo **Controle**.

As modificações da largura de banda são feitas usando a disciplina de escalonamento de filas HTB (Seção 3.5). A cada nova iteração do módulo **Controle**, uma nova largura de banda é definida para cada interface de rede. Para atribuir essa nova largura de banda foi criada uma classe HTB que limita a largura de banda máxima permitida em cada interface. Cada interface possui uma classe diferente, logo cada interface recebe um limite diferente de acordo com a largura de banda calculada no módulo **Análise**. A aplicação do controle é mostrada no Código A.12 do Apêndice A.

Como falado na seção anterior, o gerenciador é executado a cada 1 minuto. Depois de alterar as larguras de banda, o módulo **Controle** espera 1 minuto antes de voltar a execução para o módulo **Predição**, formando um ciclo, que se repete até o usuário do GPLNEURO abortar o programa, finalizando o gerenciamento.

5.3 Experimento com o Algoritmo GPLNEURO

O experimento com o algoritmo GPLNEURO consistiu em aplicar o GPLNEURO em um roteador e testar sua eficiência, avaliando a precisão da predição e a influência do algoritmo para a alocação de recursos. A alocação de recursos foi feita como propósito de controle de congestionamento através da distribuição mais justa da largura de banda para diminuir o atraso médio de transferências de pacotes.

Os códigos fonte de todos os módulos do algoritmo GPLNEURO, descritos na seção anterior, estão apresentados no Apêndice A. Cada módulo foi encapsulado em uma classe diferente e cada classe possui sua função específica, referentes aos módulos **Monitoramento**, **Treinamento**, **Predição**, **Análise** e **Controle**.

5.3.1 Ambiente de Testes

O experimento final foi realizado em máquinas físicas, pois máquinas virtuais não representam completamente a realidade, podendo afetar os resultados do gerenciador e dos dados trafegados no experimento. Os experimentos foram feitos no Laboratório de Redes do curso de Ciência da Computação da Universidade Federal de Uberlândia. O roteador foi implementado em um servidor que possui quatro interfaces de rede, denominadas *eth0*, *eth1*, *eth2* e *eth3*. O sistema operacional do servidor é o *CentOS release 5.11 (Final)*, mais detalhes das configurações do roteador são descritos no Apêndice B.1. Para executar o GPLNEURO o servidor necessita dos seguintes *softwares* instalados:

- Protocolo SNMP versão 2;
- Python 2.7;
- Java SE Runtime Environment 6 e Encog versão 3.2.0;
- Mecanismo de controle de tráfego HTB, comando *tc-htb* dos sistemas Linux.

A interface 0 (*eth0*) do servidor foi conectada à Internet e as interfaces restantes foram conectadas à diferentes sub-redes locais. As sub-redes foram criadas a partir de redes virtuais (*Virtual LANs* — VLANs) de um *switch*, para cada sub-rede foi conectado um computador que recebe dados do servidor. A capacidade máxima de largura de banda das interfaces do roteador é de 100 Mbit/s. As configurações das máquinas presentes nas sub-redes e do *switch* estão nos Apêndices B.2 e B.3, respectivamente. A Figura 5.10 mostra o ambiente de testes, incluindo o roteador e as sub-redes que se conectam ao roteador.

5.3.2 Execução do Experimento de Gerenciamento

O experimento realizado usou o algoritmo GPLNEURO para gerenciar o tráfego que passa nas interfaces de conexão do roteador com as VLANs. Para efeito de simplificação, a

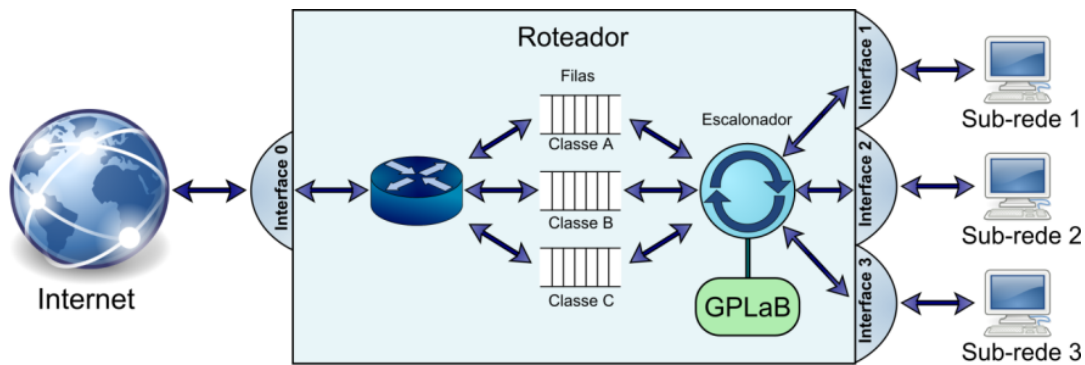


Figura 5.10: Roteador onde foi implementado o GPLNEURO para os experimentos referentes ao gerenciamento da largura de banda.

interface que se conecta à Internet foi desconsiderada nesse experimento. O GPLNEURO funciona através da predição do tráfego que passa nas interfaces *eth1*, *eth2* e *eth3* do roteador. Para gerar esse tráfego de dados foi utilizado o protocolo *HyperText Transfer Protocol* (HTTP), onde os computadores das VLANs requisitam a transferência de um arquivo presente no servidor.

O tráfego de dados no roteador foi criado de uma forma parcialmente aleatória. Primeiramente, o tráfego foi criado de modo aleatório, realizando transferências de dados variando no intervalo $[0, max]$, onde *max* é a quantidade máxima de bytes que podem ser transferidos por segundo nas interfaces do roteador. Após gerar o tráfego aleatório, ele foi moldado para apresentar características de tendência e sazonalidade.

Esta abordagem foi escolhida para deixar o tráfego gerado mais próximo do que ocorre na realidade, pois o tráfego de Internet não é aleatório e possui características de auto-similaridade, como visto no Capítulo 4.

O tráfego gerado foi moldado formando dois períodos sazonais com alto tráfego, um período sazonal de baixo tráfego, um período com tráfego médio, dois períodos de tendência decrescente e um período de tendência crescente. O tráfego total que foi gerado é executado durante aproximadamente 17 horas, a série temporal desse tráfego é gerada pelo módulo **Monitoramento** do algoritmo GPLNEURO. Assim o módulo **Treinamento** usa essa série temporal para treinar a JNN que será utilizada na predição do GPLNEURO.

A série temporal do tráfego, parcialmente aleatório, gerado pode ser vista na Figura 5.11. A série está apresentada em bytes (eixo *y*) por minuto (eixo *x*) e é composta por períodos de alto tráfego, tráfego decrescente, baixo tráfego, tráfego crescente e tráfego mediano, em uma duração total de 17 horas (1020 minutos). O limite superior do tráfego foi de $750MByte/min = 100Mbit/s * 60$, que é a capacidade máxima de bytes que podem ser transferidos por minuto nas interfaces do roteador.

Para transferir os dados e gerar o tráfego que sai das interfaces do roteador para as sub-redes, foram criados arquivos com seguintes tamanhos: 100MB; 50MB; 10MB; 1MB; 500KB; e 100KB. Esses tamanhos foram escolhidos de forma arbitrária e de forma que tenham uma variação significativa entre cada um deles, a fim de gerar uma heterogenei-

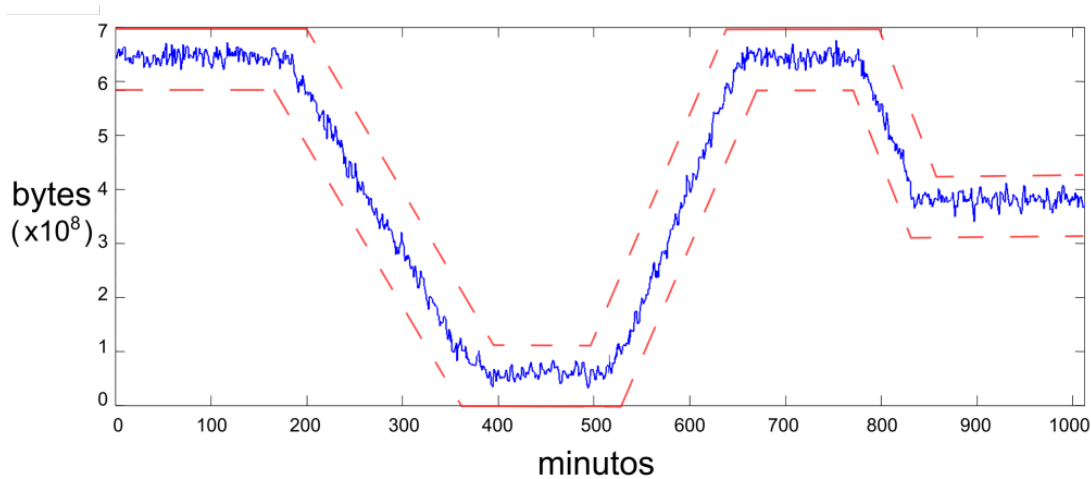


Figura 5.11: Série temporal do tráfego gerado na interface *eth3* do roteador, em um período de 1020 minutos (17 horas).

dade nos fluxos de tráfego das interfaces do roteador usado no experimento. A escolha de utilizar um tráfego gerado foi feita para criar um ambiente de testes controlado, facilitando a comparação dos testes que usam o algoritmo GPLNEURO com testes que não o usam.

A geração do tráfego, de forma parcialmente aleatória, pode ser vista no *script*, escrito na linguagem Python, apresentado no A.14 do Apêndice A. A partir do tráfego gerado, foram feitos dois experimentos. O primeiro foi o experimento de controle, no qual o tráfego é gerado, porém não houve atuação do GPLNEURO. O segundo experimento usou o GPLNEURO para atuar sobre o mesmo tráfego gerado, com isso, é possível fazer uma comparação de eficiência quando usado o algoritmo de gerenciamento.

5.3.3 Análise dos Resultados do Experimento de Gerenciamento

Das redes neurais de Jordan que foram treinadas com o tráfego gerado, a rede com o melhor resultado possui três camadas mais a camada de contexto. Como o horizonte de predição é de um valor, então a camada de saída possui somente um neurônio. O tamanho da janela de entrada foi de 5 observações, ou seja, 5 neurônios na camada de entrada. Para a camada oculta, 30 neurônios foram utilizados. O NRMSE médio das predições da JNN utilizada no GPLNEURO é de 0.018111. A Figura 5.12 compara o tráfego gerado e a predição do tráfego gerado, nela observa-se que a JNN aprendeu bem as características do tráfego gerado.

O algoritmo GPLNEURO monitora cada interface do roteador e realiza os módulos **Predição**, **Análise** e **Controle**, de forma sequencial, uma vez por minuto. O resultado das predições realizadas sobre o tráfego transmitido nas interfaces do roteador pode ser visto na Figura 5.12. O tráfego original, em azul, é o tráfego que foi gerado em uma interface do roteador e o tráfego predito, em vermelho, são os valores que a JNN previu a cada iteração. Vale ressaltar que, devido às regras de alocação definidas no módulo

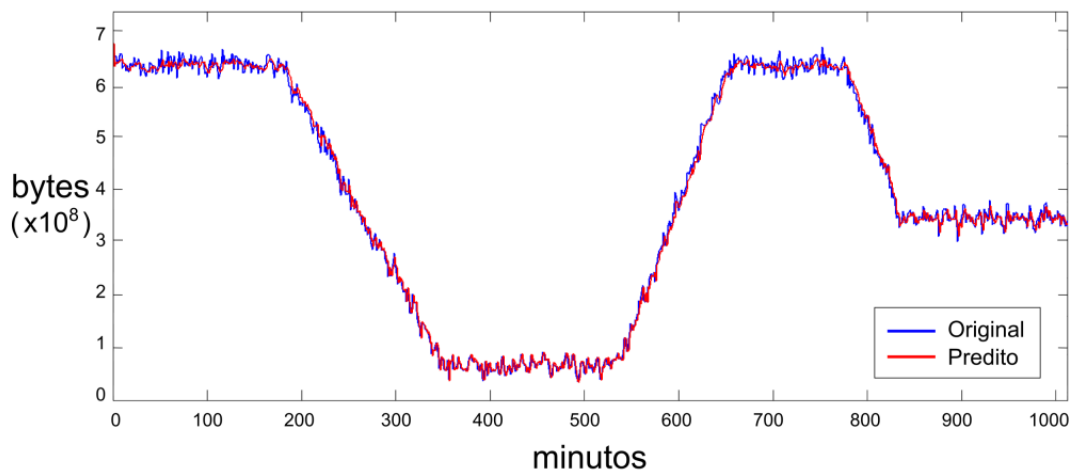


Figura 5.12: Comparação entre o tráfego gerado original e o tráfego predito pela JNN da interface *eth3* do roteador.

Análise, a largura de banda que é disponibilizada para a interface acaba sendo um pouco maior do que a largura de banda prevista.

A Figura 5.13 apresenta o resultado da aplicação do GPLNEURO sobre o atraso de transferência nas interfaces do roteador. Esse atraso é o tempo médio, em milissegundos (ms), de transferência de 64 bytes saindo do roteador para a sub-rede. Ele foi medido a cada 10 segundos (6 vezes por minuto) para verificar a disponibilidade e o tempo de resposta médio do roteador.

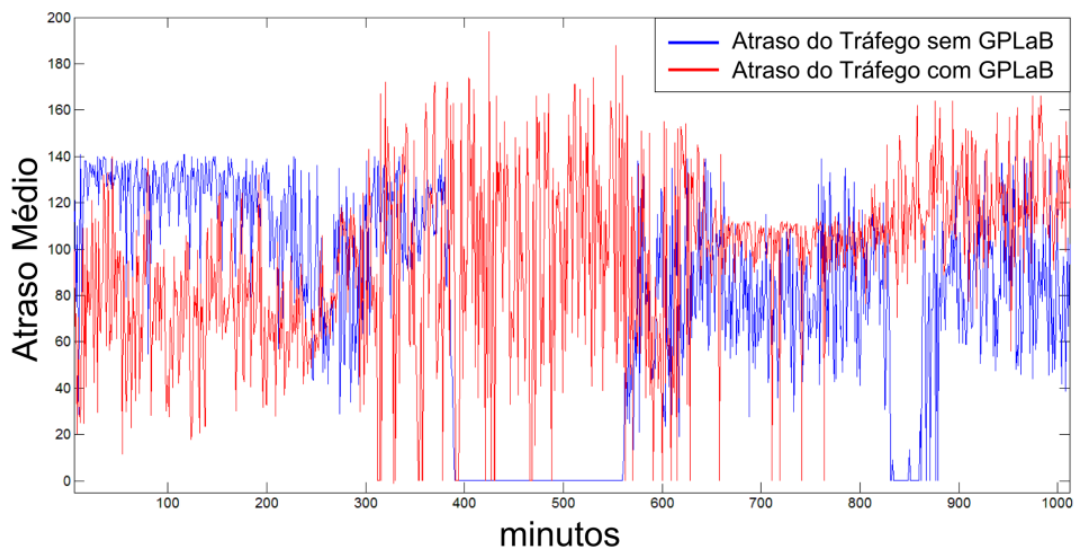


Figura 5.13: Tempo médio de atraso (ms) para cada minuto do tráfego passado na interface de rede *eth3*.

Com a atuação do GPLNEURO o atraso sofreu algumas mudanças, sendo a mais clara delas a grande variação dele quando o tráfego é baixo e, por sua vez, a largura de banda disponível também é baixa, fazendo o tempo de resposta aumentar. Essa variação é constatada entre os minutos 300 e 600 da Figura 5.13. Neste intervalo, no tráfego sem o uso do GPLNEURO (em azul), a largura disponível é muito maior do que o necessário,

assim, a transferência dos dados é muito rápida e é feita em segundos. Por isso, durante todo o minuto o atraso é baixo, pois não há dados para serem transferidos. Já no tráfego com o uso do GPLNEURO (em vermelho) a largura de banda é distribuída igualmente para cada segundo, de forma a usar somente a largura de banda necessária. Desse modo, em um minuto são trafegados a mesma quantidade de bytes por segundo, em vez de trafegar todos os dados em alguns segundos e durante o resto do tempo a largura de banda ficar subutilizada.

Tabela 5.1: Atraso (ms) médio e desvio padrão das interfaces do roteador relativos ao uso do algoritmo GPLNEURO.

Atuação do GPLNEURO	Atraso médio do tráfego de cada interface		
	<i>eth1</i>	<i>eth2</i>	<i>eth3</i>
Não	79.16396	72.25790	77.43076
Sim	72.49871	65.49184	71.38260

Atuação do GPLNEURO	Desvio padrão do atraso do tráfego de cada interface		
	<i>eth1</i>	<i>eth2</i>	<i>eth3</i>
Não	47.38965	50.39022	47.87002
Sim	53.52318	51.65973	52.73791

Outra mudança visível ao usar o GPLNEURO é a menor variação do atraso nos períodos de tráfego intenso. A Tabela 5.1 apresenta o atraso médio e o desvio padrão com e sem a atuação do GPLNEURO. Observa-se que o atraso médio é menor com a atuação do algoritmo GPLNEURO, entretanto o desvio padrão é maior ao usar o gerenciador proposto.

5.4 Conclusão do Capítulo

Neste capítulo foi proposto um novo algoritmo de gerenciamento adaptativo de largura de banda, baseado na predição do tráfego das interfaces de rede em um roteador. Esse algoritmo foi nomeado como GPLNEURO e experimentos foram feitos comparando o atraso na transferência de pacotes em dois cenários, no primeiro o GPLNEURO foi aplicado na rede de testes e no segundo não foi aplicado nenhum gerenciamento. Os experimentos realizaram uma simulação da atuação do algoritmo GPLNEURO em um ambiente real.

O algoritmo GPLNEURO atua como um sistema de gerenciamento centralizado, que auxilia no gerenciamento de desempenho em uma rede de computadores. Como mostra a Tabela 5.1 o GPLNEURO deixa a rede com um atraso médio menor que uma rede sem gerenciamento. Isso faz com que um grande volume de pacotes transferidos cheguem mais rápidos em seus destinos, diminuindo o tempo de permanência desses pacotes em

elementos intermediários da rede. Desse modo, o congestionamento pode ser evitado, pois o tempo em que os *hosts* esperam os pacotes é diminuído.

A análise do tráfego de uma rede de computadores facilita a tomada de decisões sobre a alocação de recursos como a largura de banda e a alocação de largura de banda impacta no desempenho das redes de computadores. Em virtude da heterogeneidade da rede uma alocação fixa da largura de banda não é ideal, com isso o algoritmo GPLNEURO é uma solução para realizar a realocação da largura de banda de modo dinâmico.

Finalmente, a predição do tráfego nas interfaces do roteador, através das redes neurais, mostra-se uma abordagem viável. A rede neural é capaz de prever o tráfego e o algoritmo GPLNEURO usa essa previsão para realocar a largura de banda, diminuindo o atraso médio no tráfego de dados entre o roteador e os dispositivos que se conectam a ele.

Considerações Finais

Esta dissertação apresenta duas contribuições principais, o estudo comparativo da predição de tráfego de Internet usando redes neurais artificiais (Oliveira *et al.*, 2014) e o algoritmo de Gerenciamento Preditivo de Largura de Banda usando Redes Neurais (GPL-NEURO). O estudo da predição de tráfego é importante para verificar a competitividade dos novos modelos de redes neurais de aprendizagem profunda (RNAP). Este modelo cresce em popularidade nas pesquisas de reconhecimento de padrões em imagens, áudio e vídeo, apresentando resultados bastante competitivos (Arel *et al.*, 2010; Bengio, 2009; Erhan *et al.*, 2009; Ranzato *et al.*, 2007) quando comparado com as redes neurais tradicionais, como a *Multilayer Perceptron* (MLP) e *Recurrent Neural Network* (RNN). Porém, ainda há uma falta de estudos sobre a capacidade de predição das RNAPs para o tráfego de Internet e séries temporais.

No estudo comparativo das redes neurais, além da precisão da predição, foram considerados também, o tempo de treinamento da RNA e a complexidade medida a partir do número de neurônios. Foi apresentado as especificações das RNAs tradicionais (MLP e RNN) e da RNAP e os seus respectivos algoritmos de treinamento. Os dois principais algoritmos de treinamento foram avaliados nos experimentos, sendo eles o *Backpropagation* e o *Resilient Backpropagation*. Para avaliar a RNAP foi utilizado o *Stacked Autoencoder* (SAE), que é uma rede neural profunda bastante popular e usa um algoritmo de treinamento guloso que treina uma camada por vez.

Todas as redes neurais utilizadas foram capazes de se ajustar às séries temporais do tráfego de Internet. Suas predições obtiveram bons resultados, com alto grau de generalização dos dados. Entretanto, a complexidade extra do pré-treinamento do SAE não trouxe benefícios no seu treinamento. Embora esse tipo de rede seja ideal para o

treinamento de dados não rotulados, ela não é a melhor das opções quando se trata de aprendizagem de dados rotulados, como as séries temporais. Os resultados indicam que as redes neurais tradicionais são melhores, tanto em precisão da predição quanto em tempo de treinamento, para a predição de séries temporais.

O uso e a importância de redes neurais com aprendizagem profunda está aumentando e resultados muito bons são obtidos em reconhecimento de padrões em imagens, áudio e vídeo (Arel *et al.*, 2010; Chao *et al.*, 2011; Larochelle *et al.*, 2009). No entanto, os principais algoritmos de treinamento para essas redes são algoritmos de treinamento não supervisionados, os quais usam dados não rotulados para o treinamento. Em contraste, séries temporais e tráfego de Internet são compostas de dados rotulados, por isso deve ser usado o treinamento não supervisionado como pré-treinamento e depois é aplicado o treinamento supervisionado como refinamento.

Trabalhos futuros pretendem modificar as *Deep Belief Networks* (DBN) e as *Restricted Boltzmann Machines* (RBM), que são métodos de aprendizagem profunda, para trabalhar melhor com dados rotulados (Chao *et al.*, 2011). Além disso, também pode ser feito um estudo com métodos mais novos de aprendizagem profunda que trabalham com dados contínuos, como a *Continuous Restricted Boltzmann Machine* (CRBM). Também podem ser consideradas as redes neurais com treinamento dinâmico, assim, o GPLNEURO não precisaria se preocupar em treinar novamente a rede após um período de tempo pré-determinado.

Em relação à segunda contribuição, o algoritmo de gerenciamento adaptativo de largura de banda, baseado na predição do tráfego das interfaces de rede em roteadores, foram apresentadas sua especificação, arquitetura e os experimentos realizados. Os experimentos realizaram uma simulação da atuação do algoritmo GPLNEURO em um ambiente real, focando na distribuição mais justa da largura de banda ao longo do tráfego nas interfaces do roteador.

A análise do tráfego de uma rede de computadores facilita a tomada de decisões sobre a alocação de recursos como a largura de banda. A alocação de recursos limitados impacta significativamente no desempenho das redes de computadores, contudo, dada a heterogeneidade do tráfego, uma alocação fixa de recursos pode não ser suficiente. O sistema de gerenciamento adaptativo implementado é uma solução para realizar a realocação da largura de banda de modo dinâmico. A predição do tráfego nas interfaces do roteador, através das redes neurais, mostra-se uma abordagem viável.

O algoritmo GPLNEURO, proposto nesta monografia, atua como um sistema de gerenciamento centralizado, que auxilia no gerenciamento de desempenho em uma rede de computadores, conforme os resultados apresentados nos Capítulos 4 e 5. A rede neural é capaz de prever o tráfego e o GPLNEURO usa essa previsão para adaptar a largura de banda, diminuindo o atraso médio no tráfego de dados entre o roteador e os dispositivos que se conectam a ele.

É importante salientar que, devido à limitação do laboratório onde os experimentos foram realizados, não foi possível testar o GPLNEURO em uma rede de computadores mais complexa. O uso do protocolo SNMP proporciona uma alta escalabilidade e portabilidade para o algoritmo GPLNEURO. Desse modo, trabalhos futuros podem realizar experimentos que induzam o congestionamento do tráfego, isto para verificar a eficiência do GPLNEURO no controle de congestionamento.

Outro trabalho futuro importante é comparar os resultados os resultados do GPLNEURO com outros algoritmos de gerenciamento de largura de banda, como por exemplo, o algoritmo SGALB proposto por Costa (2013). Além disso, uma proposta é alterar o GPLNEURO para que ele seja capaz de trabalhar em um sistema de gerenciamento de redes distribuído. Neste caso, seria interessante usar o tráfego de toda a rede de computadores, em vez de somente o tráfego no roteador, para a previsão de áreas da rede que podem necessitar de uma quantidade maior de recursos.

Outra sugestão de trabalho futuro é a criação de um *framework* para disponibilizar o gerenciamento de rede de uma forma mais variada. Esse *framework* pode tratar o gerenciamento da largura de banda de uma forma geral, possibilitando uma alteração diferente para propósitos diferentes do gerenciamento. Em outras palavras, o *framework* pode trabalhar com regras diferentes das regras apresentadas no módulo de análise do GPLNEURO, bem como ser capaz de usar outros métodos de predição de tráfego, como ARIMA, *Seasonal* ARIMA, *Fractionally* ARIMA (FARIMA), métodos Bayesianos, transformadas *Wavelets*, métodos de combinação de regras para preditores e entre outros.

O usuário desse *framework* proposto também poderá escolher qual o protocolo de monitoramento que seja mais adequado ele, como por exemplo o SNMPv2, SNMPv3, RMON ou outros. O sistema de gerenciamento de redes que usará o *framework* terá a possibilidade de escolher quais metodologias serão usadas em cada módulo do GPLNEURO, isso deixará o *framework* GPLNEURO mais amplo e dinâmico, possibilitando o uso mais eficiente dependendo do tipo de gerenciamento e da rede de computadores que será gerenciada.

Referências

- An, F.-T., Hsueh, Y.-L., Kim, K. S., White, I. M., e Kazovsky, L. G. (2014). A New Dynamic Bandwidth Allocation Protocol with Quality of Service in Ethernet-based Passive Optical Networks. *ArXiv e-prints*.
- Antony, S., Antony, S., Beegom, A. S. A., e Rajasree, M. S. (2012). HTB Based Packet Scheduler for Cloud Computing. In *Proceedings of the Second International Conference on Computational Science, Engineering and Information Technology*, CCSEIT '12, páginas 656–660, Coimbatore UNK, India, New York, NY, USA. ACM.
- Arel, I., Rose, D., e Karnowski, T. (2010). Deep Machine Learning - A New Frontier in Artificial Intelligence Research [Research Frontier]. *Computational Intelligence Magazine, IEEE*, 5(4):13–18.
- Astuti, D. (2003). Packet Handling - Seminar on Transport of Multimedia Streams in Wireless Internet Davide. Technical report, University of Helsinki.
- Bai, X. e Shami, A. (2013). Modeling self-similar traffic for network simulation. *ArXiv e-prints*.
- Basheer, I. e Hajmeer, M. (2000). Artificial neural networks: fundamentals, computing, design, and application. *Journal of Microbiological Methods*, 43(1):3 – 31.
- Bengio, Y. (2009). Learning Deep Architectures for AI. *Found. Trends Mach. Learn.*, 2(1):1–127.
- Bengio, Y., Lamblin, P., Popovici, D., e Larochelle, H. (2007). Greedy Layer-Wise Training of Deep Networks. In *Advances in Neural Information Processing Systems 19*, páginas 153–160.
- Bolshinsky, E. e Freidman, R. (2012). Traffic Flow Forecast Survey. Technical report, Technion - Computer Science Department.
- Box, G. E. P., Jenkins, G. M., e Reinsel, G. C. (2008). *Time Series Analysis: Forecasting and Control*. John Wiley, Hoboken, N.J, 4 edition.
- Brown, M. A. (2006). Traffic Control HOWTO. Technical report, linux-ip.net.
- Busseti, E., Osband, I., e Wong, S. (2012). Deep Learning for Time Series Modeling. Stanford, CS 229: Machine Learning. URL: <http://cs229.stanford.edu/projects2012.html>.

- Case, J. D., Fedor, M., Schoffstall, M. L., e Davin, C. (1990). RFC 1157: Simple Network Management Protocol (SNMP). URL: <ftp://ftp.internic.net/rfc/rfc1157.txt>.
- Chao, J., Shen, F., e Zhao, J. (2011). Forecasting exchange rate with deep belief networks. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, páginas 1259–1266.
- Chatfield, C. (2000). *Time-Series Forecasting*. Chapman and Hall/CRC, Boca Raton, 1 edition.
- Chatfield, C. (2004). *The Analysis of Time Series: An Introduction*. Chapman and Hall/CRC, Boca Raton, FL, 6 edition.
- Chen, M., Xu, Z., Weinberger, K. Q., e Sha, F. (2012a). Marginalized stacked denoising autoencoders. ICML.
- Chen, Y., Yang, B., e Meng, Q. (2012b). Small-time scale network traffic prediction based on flexible neural tree. *Applied Soft Computing*, 12(1):274–279.
- Comer, D. E. (2007). *Redes de Computadores E Internet*. Bookman, Porto Alegre, 4 edition.
- Cortez, P., Rio, M., Rocha, M., e Sousa, P. (2006). Internet Traffic Forecasting using Neural Networks. In *Neural Networks, 2006. IJCNN '06. International Joint Conference on*, páginas 2635–2642.
- Cortez, P., Rio, M., Sousa, P., e Rocha, M. (2008). Forecasting Internet traffic by Neural Networks under univariate and multivariate strategies. URL: <http://repositorium.sdum.uminho.pt/handle/1822/8027>.
- Cortez, P., Rio, M., Rocha, M., e Sousa, P. (2012). Multi-scale Internet traffic forecasting using neural networks and time series methods. *Expert Systems*, 29(2):143–155.
- Costa, N. T. d. (2013). Predição de Tráfego, Usando WT e ARIMA, Aplicada ao Gerenciamento Adaptativo de Largura de Banda para Interfaces de Roteadores. Mestrado, Pós-graduação em Ciência da Computação da Universidade Federal de Uberlândia.
- Crone, S. F., Hibon, M., e Nikolopoulos, K. (2011). Advances in forecasting with neural networks: Empirical evidence from the {NN3} competition on time series prediction. *International Journal of Forecasting*, 27(3):635 – 660.
- de Villiers, J. e Barnard, E. (1993). Backpropagation neural nets with one and two hidden layers. *Neural Networks, IEEE Transactions on*, 4(1):136–141.
- Encog (2014). Encog Artificial Intelligence Framework for Java and DotNet. URL: <http://www.heatonresearch.com/encog>.
- Erhan, D., Manzagol, P.-A., Bengio, Y., Bengio, S., e Vincent, P. (2009). The Difficulty of Training Deep Architectures and the Effect of Unsupervised Pre-Training. In *Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS)*, páginas 153–160.

- Feng, H. e Shu, Y. (2005). Study on network traffic prediction techniques. In *Wireless Communications, Networking and Mobile Computing, 2005. Proceedings. 2005 International Conference on*, volume 2, páginas 1041–1044.
- Hallas, M. e Dorffner, G. (1998). A Comparative Study on Feedforward and Recurrent Neural Networks in Time Series Prediction Using Gradient Descent Learning. URL: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.6673>.
- Haykin, S. (1998). *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition.
- Haykin, S. e Moher, M. (2009). *Communication Systems*. Wiley, Hoboken, NJ, 5 edition.
- Hinton, G. E., Osindero, S., e Teh, Y.-W. (2006). A Fast Learning Algorithm for Deep Belief Nets. *Neural Comput.*, 18(7):1527–1554.
- Hornik, K. e Leisch, F. (2000). *Neural Network Models*, páginas 348–362. John Wiley & Sons, Inc.
- Hornik, K., Stinchcombe, M., e White, H. (1989). Multilayer Feedforward Networks Are Universal Approximators. *Neural Netw.*, 2(5):359–366.
- Hosseini, S., Moshiri, B., Rahimi-Kian, A., e Araabi, B. (2012). Short-term traffic flow forecasting by mutual information and artificial neural networks. In *Industrial Technology (ICIT), 2012 IEEE International Conference on*, páginas 1136–1141.
- Huang, Y., Sheng, H., e Chen, J. (2014). Intelligent Congestion Avoidance Algorithm and System – Application of Data Vitalization. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, páginas 847–856.
- Hubert, B. (2012). Linux Advanced Routing and Traffic Control HOWTO. Technical report, Netherlabs BV.
- Hyndman, R. (2014). Time Series Data Library. URL: <http://data.is/TSDLdemo>.
- Hyndman, R. J. e Athanasopoulos, G. (2013). *Forecasting: principles and practice*. OTexts, Heathmont, Vic.
- IF-MIB (2014). Net-SNMP Interfaces MIB Object. URL: <http://www.net-snmp.org/docs/mibs/interfaces.html>.
- Kalekar, P. S. (2004). Time series Forecasting using Holt-Winters Exponential Smoothing. URL: http://www.it.iitb.ac.in/~praj/acads/seminar/04329008_ExponentialSmoothing.pdf.
- Karris, S. (2009). *Networks: Design and Management*. Orchard Publications, Fremont, Calif.
- Kirchgässner, G., Wolters, J., e Hassler, U. (2013). *Introduction to Modern Time Series Analysis*. Springer, Berlin New York, 2 edition.
- Kouhi, S. e Keynia, F. (2013). A new cascade NN based method to short-term load forecast in deregulated electricity market. *Energy Conversion and Management*, 71:76–83.

- Kurose, J. F. e Ross, K. W. (2012). *Computer Networking: A Top-Down Approach*. Pearson, Boston, 6 edition.
- Larochelle, H., Erhan, D., e Vincent, P. (2009). Deep Learning using Robust Interdependent Codes. In D. V. Dyk e M. Welling, editores, *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS-09)*, volume 5, páginas 312–319. Journal of Machine Learning Research - Proceedings Track.
- Leinwand, A. e Fang, K. (1996). *Network Management: A Practical Perspective*. Addison-Wesley Professional, Reading, Mass, 2 edition.
- McCloghrie, K. e Rose, M. T. (1991). RFC 1213: Management Information Base for Network Management of TCP/IP-based internets:MIB-II. URL: <ftp://ftp.internic.net/rfc/rfc1213.txt>.
- Morettin, P. A. e de Castro Toloi, C. M. (2006). *Análise de séries temporais*. Edgard Blucher, São Paulo.
- Oliveira, T. P., Barbar, J. S., e Soares, A. S. (2014). Multilayer Perceptron and Stacked Autoencoder for Internet Traffic Prediction. In C.-H. Hsu, X. Shi, e V. Salapura, editores, *Network and Parallel Computing - 11th IFIP WG 10.3 International Conference, NPC 2014, Ilan, Taiwan, September 18-20, 2014. Proceedings*, volume 8707 of *Lecture Notes in Computer Science*, páginas 61–71. Springer.
- Palm, R. (2014). DeepLearnToolbox, a Matlab toolbox for Deep Learning. URL: <https://github.com/rasmusbergpalm/DeepLearnToolbox>.
- Pandey, G., Siddiqui, K. M., e Choudhary, A. (2013). Telecom Voice Traffic Prediction for GSM using Feed Forward Neural Network. *International Journal of Engineering Science*.
- Peterson, L. L. e Davie, B. S. (2011). *Computer Networks: A Systems Approach*. Morgan Kaufmann, Amsterdam Boston, 5 edition.
- Python (2014). Linguagem de programação Python. URL: <https://www.python.org/>.
- Rajput, S., Kumar, V., e Paul, S. (2014). Comparative analysis of random early detection (RED) and virtual output queue (VOQ) algorithms in differentiated services network. In *Signal Processing and Integrated Networks (SPIN), 2014 International Conference on*, páginas 237–240.
- Ranzato, M., Jan Boureau, Y., e Cun, Y. L. (2007). Sparse Feature Learning for Deep Belief Networks. In J. Platt, D. Koller, Y. Singer, e S. Roweis, editores, *Advances in Neural Information Processing Systems 20*, páginas 1185–1192. MIT Press, Cambridge, MA.
- Risso, F. e Gevros, P. (1999). Operational and Performance Issues of a CBQ Router. *SIGCOMM Comput. Commun. Rev.*, 29(5):47–58.
- Rose, M. T. e McCloghrie, K. (1990). RFC 1155: Structure and identification of management information for TCP/IP-based internets. URL: <ftp://ftp.internic.net/rfc/rfc1155.txt>.

- Rutka, G. e Lauks, G. (2007). Study on Internet Traffic Prediction Models. *Telecommunications Engineering*.
- Sahin, S., Tolun, M. R., e Hassanpour, R. (2012). Hybrid expert systems: A survey of current approaches and applications. *Expert Systems with Applications*, 39(4):4609–4617.
- Siripongwutikorn, P., Banerjee, S., e Tipper, D. (2003). A survey of adaptive bandwidth control algorithms. *Communications Surveys Tutorials, IEEE*, 5(1):14–26.
- SNMPv2 Working Group, Case, J., McCloghrie, K., Rose, M., e Waldbusser, S. (1996). RFC 1901: Introduction to Community-based SNMPv2. URL: <ftp://ftp.internic.net/rfc/rfc1901.txt>. Status: EXPERIMENTAL.
- Stallings, W. (1999). *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*. Addison-Wesley Professional, Reading, Mass, 3 edition.
- Stepnicka, M., Donate, J. P., Cortez, P., Vavříková, L., e Gutierrez, G. (2011). Forecasting seasonal time series with computational intelligence: contribution of a combination of distinct methods. In *Proceedings of the 7th conference of the European Society for Fuzzy Logic and Technology (EUSFLAT-2011) and LFA-2011.*, páginas 464–471.
- Tanenbaum, A. S. e Wetherall, D. J. (2010). *Computer Networks*. Pearson Prentice Hall, Boston, 5 edition.
- Tong, H., Li, C., He, J., e Chen, Y. (2005). Internet Traffic Prediction by W-boost: Classification and Regression. In *Proceedings of the Second International Conference on Advances in Neural Networks - Volume Part III*, ISNN'05, páginas 397–402, Chongqing, China, Berlin, Heidelberg. Springer-Verlag.
- Ufldl (2014). Unsupervised Feature Learning and Deep Learning. Stanford's online wiki. Stacked Autoencoders. URL: http://ufldl.stanford.edu/wiki/index.php/Stacked_Autoencoders.
- Uma, M. e Padmavathi, G. (2012). An Efficient Network Traffic Monitoring for Wireless Networks. *International Journal of Computer Applications*, 53(9):51–59.
- Wang, X., Jung, S., e Meditch, J. (1998). Dynamic bandwidth allocation for VBR video traffic using adaptive wavelet prediction. In *Communications, 1998. ICC 98. Conference Record. 1998 IEEE International Conference on*, volume 1, páginas 549–553 vol.1.
- Yan, Z., Chen, C. W., e Liu, B. (2014). Admission Control for Wireless Adaptive HTTP Streaming: An Evidence Theory Based Approach. In *Proceedings of the ACM International Conference on Multimedia, MM '14*, páginas 893–896, Orlando, Florida, USA, New York, NY, USA. ACM.
- Zhang, R., Chai, Y., e an Fu, X. (2012). A network traffic prediction model based on recurrent wavelet neural network. In *Computer Science and Network Technology (ICCSNT), 2012 2nd International Conference on*, páginas 1630–1633.
- Zhang, Y. e Ansari, N. (2013). On Architecture Design, Congestion Notification, TCP Incast and Power Consumption in Data Centers. *Communications Surveys Tutorials, IEEE*, 15(1):39–64.

Zhani, M., Elbiaze, H., e Kamoun, F. (2008). Analysis of prediction performance of training-based models using real network traffic. In *Performance Evaluation of Computer and Telecommunication Systems, 2008. SPECTS 2008. International Symposium on*, páginas 472–479.

Códigos Fonte dos Experimentos

Aqui são apresentados os códigos fonte utilizados para os experimentos de predição de tráfego e do Gerenciador Preditivo de Largura de Banda usando Redes Neurais (GPLNEURO). A Seção A.1 apresenta os códigos, escritos na linguagem de programação Java, usando o *framework* Encog, que foram usados para os experimentos relativos à predição dos modelos MLP-BP, MLP-RP e JNN. Ainda nessa seção, é apresentado o código, escrito em MATLAB, usando a ferramenta DeepLearn Toolbox, que faz o treinamento e a predição da rede neural de aprendizagem profunda SAE. A Seção A.2 apresenta os códigos utilizados para criar o algoritmo GPLNEURO e *script* para a geração do tráfego entre o roteador e os computadores das sub-redes locais. Ambos utilizam a linguagem de programação Python 2.7. Também é apresentado o código em Java, que foi usado para realizar a predição do módulo **Predição** do algoritmo GPLNEURO.

A.1 Códigos para os Experimentos de Predição de Tráfego

Código A.1: Classe Java principal que realiza os experimentos das redes MLP-BP, MLP-RP e JNN

```
1 package encog.prediction.nn;  
2  
3 import java.io.File;  
4 import java.util.Collections;  
5 import java.util.LinkedList;  
6 import java.util.List;
```

```

7  import encog.prediction.utils.FileUtils;
8  import encog.prediction.utils.ResultsTuple;
9
10 /**
11  * Classe para realizar os experimentos comparativos das predicoes entre
12  *   redes neurais diferentes.
13  * @author Tiago
14  */
15 public class TestPredictions {
16     public static int id = 1; // id pra colocar nos nomes e diferenciar
17     // redes com mesmas camadas
18     public static final int MAX_TRAINING_EPOCHS = 1000;
19     public static final double MIN_NORMALIZED_VALUE = 0.1d; // nao eh
20     // zero pra nao dar problema de divisao por zero
21     public static final double MAX_NORMALIZED_VALUE = 0.9d; //
22
23     /**
24     * Executa uma predicao e retorna o MSE
25     * @param name nome que sera salvo
26     * @param layers vetor com numero de neuronios em cada camada da
27     *   rede neural
28     * @param isRecurrent se a rede neural eh recorrente ou nao
29     * @return
30     */
31     public static final Double testOnePrediction(String name, int
32         layers[], boolean isRecurrent) {
33         System.out.println(name + "\t.....");
34         String inputTrainPath = Prediction.DATA_SET_PATH + File.
35             separator + Prediction.T;
36         String inputTestPath = Prediction.DATA_SET_PATH + File.
37             separator + Prediction.T;
38
39         Prediction prediction = new Prediction();
40         prediction.createSepparateDataSet(inputTrainPath,
41             inputTestPath, layers[0], layers[layers.length-1]);
42
43         if(isRecurrent) {
44             prediction.createRecurrentNetwork(layers);
45         } else {
46             prediction.createFeedForwardNetwork(layers);
47         }
48
49         prediction.createTraining();
50         prediction.train(MAX_TRAINING_EPOCHS, false);
51         prediction.evaluate(false);
52
53         FileUtils.savePrediction(prediction, name);

```

```

46         List<Double> miscList = prediction.getPredictionResults().
           get(5);
47         return miscList.get(6);
48     }
49
50     /**
51      * Atribui o nome do arquivo como 'name-layers'
52      * @param name
53      * @param layers
54      * @return
55      */
56     public static final String getFileName(String name, int[] layers) {
57         String finalName = name;
58         for(int i = 0; i < layers.length; i++) {
59             finalName += "-" + layers[i];
60         }
61         return finalName + "--" + TestPredictions.id++ + ".txt";
62     }
63
64     /**
65      * Executa um numero 'max' de predicoes com a RNN
66      * @param max
67      * @return
68      */
69     public static final void testSeveralPredictionsRNN(int max) {
70         int[] layers = {5, 5, 1};
71         String nn = "rnn";
72         String name = null;
73         Double mse = null;
74         ResultsTuple rt = null;
75
76         List<ResultsTuple> results = new LinkedList<ResultsTuple>()
           ;
77         for(int i = 5; i <= max; i+=5) {
78             layers[0] = i;
79             for(int j = 5; j <= max; j+=5) {
80                 layers[1] = j;
81                 name = getFileName(nn, layers);
82                 mse = testOnePrediction(name, layers, true)
           ;
83
84                 rt = new ResultsTuple(mse, name, layers);
85                 results.add(rt);
86             }
87         }
88
89         FileUtils.saveComplexityPerNeuronsResults(results, true);

```

```

90         Collections.sort(results);
91         FileUtils.saveMSEResults(results, true);
92     }
93
94     /**
95      * Executa um numero 'max' de predicoes com a FNN
96      * @param max
97      * @return
98      */
99     public static final void testSeveralPredictionsFNN(int max) {
100         int[] layers = {5, 5, 5, 1};
101         String nn = "fnn";
102         String name = null;
103         Double mse = null;
104         ResultsTuple rt = null;
105
106         List<ResultsTuple> results = new LinkedList<ResultsTuple>()
107             ;
108         for(int i = 5; i <= max; i+=10) {
109             layers[0] = i;
110             for(int j = 5; j <= max; j+=10) {
111                 layers[1] = j;
112                 for(int k = 5; k <= max; k+=10) {
113                     layers[2] = k;
114                     name = getFileName(nn, layers);
115                     mse = testOnePrediction(name,
116                         layers, false);
117
118                     rt = new ResultsTuple(mse, name,
119                         layers);
120                     results.add(rt);
121                 }
122             }
123         }
124
125         FileUtils.saveComplexityPerNeuronsResults(results, false);
126         Collections.sort(results);
127         FileUtils.saveMSEResults(results, false);
128     }
129
130     public static void main(String[] args) {
131         testSeveralPredictionsFNN(95); // testando MLP - 3 camadas
132         testSeveralPredictionsFNN(65); // testando MLP - 4
133                                     camadas
134         testSeveralPredictionsRNN(100); // testando JNN
135     }
136 }

```

Código A.2: Classe que implementa os preditores MLP-BP, MLP-RP e JNN

```

1 package encog.prediction.nn;
2
3 import java.io.File;
4 import java.util.LinkedList;
5 import java.util.List;
6
7 import org.apache.commons.math3.stat.descriptive.DescriptiveStatistics;
8 import org.encog.engine.network.activation.ActivationSigmoid;
9 import org.encog.ml.data.temporal.TemporalDataDescription;
10 import org.encog.ml.data.temporal.TemporalMLDataSet;
11 import org.encog.ml.data.temporal.TemporalPoint;
12 import org.encog.ml.train.MLTrain;
13 import org.encog.neural.networks.BasicNetwork;
14 import org.encog.neural.networks.layers.BasicLayer;
15 import org.encog.neural.networks.training.propagation.resilient.
    ResilientPropagation;
16 import org.encog.neural.pattern.JordanPattern;
17 import org.encog.util.csv.CSVFormat;
18 import org.encog.util.csv.ReadCSV;
19
20 import encog.prediction.utils.PredictionUtils;
21 import encog.prediction.utils.StatisticsUtils;
22
23 /**
24  * Classe que encapsula os modelos de predicoes dos experimentos.
25  * @author Tiago
26  */
27 public class Prediction {
28     public static final String PATH = "data";
29     public static final String DATA_SET_FOLDER = "robJH";
30     public static final String DATA_SET_FOLDER2 = "TR";
31     public static final String DATA_SET_PATH = PATH + File.separator +
        DATA_SET_FOLDER;
32     public static final String DATA_SET_PATH2 = PATH + File.separator +
        DATA_SET_FOLDER2;
33     public static final char CSV_DELIMITEIR = ',';
34     public static final char CSV_DECIMAL = '.';
35     public static final String A5m = "internet-traffic-data-in-bits-fr-
        eu-5m.csv";
36     public static final String A1h = "internet-traffic-data-in-bits-fr-
        eu-1h.csv";
37     public static final String A1d = "internet-traffic-data-in-bits-fr-
        eu-1d.csv";
38     public static final String B5m = "internet-traffic-data-in-bits-fr-
        uk-5m.csv";

```

```

39     public static final String B1h = "internet-traffic-data-in-bits-fr-
        uk-1h.csv";
40     public static final String B1d = "internet-traffic-data-in-bits-fr-
        uk-1d.csv";
41     public static final String T = "trafego1.csv";
42     public static final String T2 = "trafego2.csv";
43     public static final String TR1 = "TR1.csv";
44     public static final String TR2 = "TR2.csv";
45     public static final String TR3 = "TR3.csv";
46     public static final float DATA_SET_TRAINING_PERCENTAGE = 0.5f; //
        treina com a metade da serie
47     public static final int DEFAULT_INPUT_WINDOW_SIZE = 10;
48     public static final int DEFAULT_HIDDEN_LAYER_SIZE = 10;
49     public static final int DEFAULT_OUTPUT_WINDOW_SIZE = 1;
50     public static final float LEARNING_RATE = 0.01f;
51     public static final float MOMENTUM = 0f;
52     // numero de vezes maxima (epocas) que o treinamento nao melhora (o
        erro eh o mesmo), caso passe desse numero, o treinamento para
53     public static final int TRAIN_NOT_IMPROVING_LIMIT = 25;
54
55     private List<Double> timeSeries;
56     private TemporalMLDataSet trainSet;
57     private TemporalMLDataSet dataSet;
58     private BasicNetwork network;
59     private MLTrain train;
60     private double finalTrainError;
61     private List<List<Double>> predictionResults;
62     private List<List<Double>> predictionResults2;
63     private List<Double> trainErrors;
64     // marca se eh recorrente ou nao (feedforward)
65     private boolean recurrent;
66     private int trainSize;
67     private int trainEpochs;
68     private long trainTime;
69     // valores min e max antes de normalizar
70     private Double minTSValuePreNorm;
71     private Double maxTSValuePreNorm;
72
73     public Prediction() {}
74
75     /**
76      * Le do arquivo 'inputPath' a serie temporal
77      * @param inputPath
78      * @return
79      */
80     public void readTimeSeries(String inputPath) {

```

```

81         CSVFormat format = new CSVFormat(CSV_DECIMAL,
82             CSV_DELIMITEUR);
83         ReadCSV timeSeriesCSV = new ReadCSV(inputPath, false,
84             format);
85
86         timeSeries = new LinkedList<>();
87         while(timeSeriesCSV.next()) {
88             Double d = timeSeriesCSV.getDouble(0);
89             timeSeries.add(d);
90         }
91     }
92
93     /**
94      * Cria o dataSet e o trainSet a partir do timeSeries e do
95      * DATA_SET_TRAINING_PERCENTAGE
96      */
97     public void createDataSet() {
98         this.dataSet = createDataSet(true,
99             DEFAULT_INPUT_WINDOW_SIZE, DEFAULT_OUTPUT_WINDOW_SIZE);
100         this.trainSet = createDataSet(false,
101             DEFAULT_INPUT_WINDOW_SIZE, DEFAULT_OUTPUT_WINDOW_SIZE);
102     }
103
104     /**
105      * Cria o dataSet e o trainSet a partir do timeSeries e do
106      * DATA_SET_TRAINING_PERCENTAGE
107      */
108     public void createDataSet(int inputSize, int outputSize) {
109         this.dataSet = createDataSet(true, inputSize, outputSize);
110         this.trainSet = createDataSet(false, inputSize, outputSize);
111     }
112
113     /**
114      * Cria o trainSet a partir de diferentes timeSeries
115      * //// metodo meio gambiarra, hehe
116      */
117     public void createSeparateDataSet(String trainCSV_TS_Path, String
118         testCSV_TS_Path, int inputSize, int outputSize) {
119         // train set
120         readTimeSeries(trainCSV_TS_Path);
121         normalizeMinMax(TestPredictionNew.MIN_NORMALIZED_VALUE,
122             TestPredictionNew.MAX_NORMALIZED_VALUE);
123         this.trainSet = createDataSet(true, inputSize, outputSize);
124         int trainSize__ = trainSize;
125         // test set
126         readTimeSeries(testCSV_TS_Path);

```

```

119         normalizeMinMax(TestPredictionNew.MIN_NORMALIZED_VALUE,
120                           TestPredictionNew.MAX_NORMALIZED_VALUE);
121         this.dataSet = createDataSet(true, inputSize, outputSize);
122         trainSize = trainSize__;
123     }
124
125     /**
126     * if useAllTS == true, usa todo o timeSeries
127     * else usa a porcentagem do timeSeries a partir do
128     * DATA_SET_TRAINING_PERCENTAGE
129     * @return
130     */
131     private TemporalMLDataSet createDataSet(boolean useAllTS, int
132         inputSize, int outputSize) {
133         TemporalMLDataSet dataSet = new TemporalMLDataSet(inputSize
134             , outputSize);
135         TemporalDataDescription desc = new TemporalDataDescription(
136             TemporalDataDescription.Type.RAW, true, true);
137         dataSet.addDescription(desc);
138
139         int dataSetSize = (int)(timeSeries.size() *
140             DATA_SET_TRAINING_PERCENTAGE);
141
142         if(useAllTS) {
143             dataSetSize = timeSeries.size();
144         }
145         trainSize = dataSetSize;
146
147         for(int i = 0; i < dataSetSize; i++) {
148             TemporalPoint point = new TemporalPoint(1);
149             point.setSequence(i);
150             point.setData(0, timeSeries.get(i));
151
152             dataSet.getPoints().add(point);
153         }
154
155         dataSet.generate();
156         return dataSet;
157     }
158
159     /**
160     * Cria a rede neural que sera utilizada na predicao usando os
161     * valores padroes pre-definidos
162     * @return
163     */
164     public void createRecurrentNetwork() {

```

```

158         int[] layers = {DEFAULT_INPUT_WINDOW_SIZE,
159                         DEFAULT_HIDDEN_LAYER_SIZE, DEFAULT_OUTPUT_WINDOW_SIZE};
160         createRecurrentNetwork(layers);
161     }
162
163     /**
164      * Cria a RNN com o numero de neuronios passado como parametro no
165      * vetor 'layers
166      * @param layers - um vetor com as camadas e numero de neuronios em
167      * cada camada
168      */
169     public void createRecurrentNetwork(int[] layers) {
170         JordanPattern pattern = new JordanPattern();
171         //ElmanPattern pattern = new ElmanPattern();
172         pattern.setActivationFunction(new ActivationSigmoid());
173         if(layers.length < 2) {
174             throw new RuntimeException("layers.length = " +
175                                     layers.length + "\nPrecisa de pelo menos duas
176                                     camadas, uma de entrada e uma de saida!");
177         }
178         pattern.setInputNeurons(layers[0]);
179         pattern.setOutputNeurons(layers[layers.length - 1]);
180         for(int i = 1; i < layers.length - 1; i++) {
181             pattern.addHiddenLayer(layers[i]);
182         }
183
184         network = (BasicNetwork) pattern.generate();
185         recurrent = true;
186     }
187
188     /**
189      * Cria a FNN usando os valores padroes pre-definidos
190      * @return
191      */
192     public void createFeedForwardNetwork() {
193         int[] layers = {DEFAULT_INPUT_WINDOW_SIZE,
194                         DEFAULT_HIDDEN_LAYER_SIZE, DEFAULT_OUTPUT_WINDOW_SIZE};
195         createFeedForwardNetwork(layers);
196     }
197
198     /**
199      * Cria a FNN com o numero de neuronios passado como parametro no
200      * vetor 'layers
201      * @param layers - um vetor com as camadas e numero de neuronios em
202      * cada camada
203      */
204     public void createFeedForwardNetwork(int[] layers) {

```

```

197         network = new BasicNetwork();
198
199         if(layers.length < 2) {
200             throw new RuntimeException("layers.length = " +
                layers.length + "\nPrecisa de pelo menos duas
                camadas, uma de entrada e uma de saida!");
201         }
202         for(int i = 0; i < layers.length; i++) {
203             network.addLayer(new BasicLayer(layers[i]));
204         }
205
206         network.getStructure().finalizeStructure();
207         network.reset();
208         recurrent = false;
209     }
210
211     /**
212      * Cria um objeto relativo ao algoritmo de treinamento
213      */
214     public void createTraining() {
215         // TODO: colocar um parametro em vez de comentar o codigo
                pra trocar de algoritmo de treinamento
216         //train = new Backpropagation(network, dataSet,
                LEARNING_RATE, MOMENTUM);
217         train = new ResilientPropagation(network, trainSet);
218     }
219
220     /**
221      * Executa o treinamento ate 'maxTrainingEpochs' vezes
222      * @param maxTrainingEpochs numero de epocas de duracao do
                treinamento
223      */
224     public void train(int maxTrainingEpochs) {
225         train(maxTrainingEpochs, false);
226     }
227
228     /**
229      * Executa o treinamento ate 'maxTrainingEpochs' vezes
230      * @param maxTrainingEpochs numero de epocas de duracao do
                treinamento
231      * @param print imprime os resultados de cada epoca de treinamento
232      */
233     public void train(int maxTrainingEpochs, boolean print) {
234         trainErrors = new LinkedList<Double>();
235         trainEpochs = 0;
236
237         double previousError = 0;

```

```

238         int notImprovingCount = 0;
239         long start = System.currentTimeMillis();
240
241         while(trainEpochs < maxTrainingEpochs) {
242             train.iteration();
243             trainEpochs++;
244             trainErrors.add(train.getError());
245             if(print) {
246                 System.out.println("Epoch: " + trainEpochs
247                                     + "\tError: " + train.getError());
248             }
249             if(previousError == train.getError()) {
250                 notImprovingCount++;
251                 if(notImprovingCount >=
252                     TRAIN_NOT_IMPROVING_LIMIT) {
253                     // se o erro nao mudou nada em '
254                     TRAIN_NOT_IMPROVING_LIMIT' vezes
255                     , entao para o treinamento
256                     break;
257                 }
258             } else {
259                 notImprovingCount = 0;
260             }
261             previousError = train.getError();
262         }
263         train.finishTraining();
264         long end = System.currentTimeMillis();
265
266         trainTime = end - start;
267         finalTrainError = train.getError();
268         if(print) {
269             System.out.println("Final Error: " +
270                               finalTrainError);
271             System.out.println("Train time: " + trainTime);
272         }
273     }
274
275     public void normalizeZScore() {
276         List<Double> ts = StatisticsUtils.normalizeZScore(
277             timeSeries);
278         timeSeries = ts;
279     }
280
281     public void normalizeMinMax(double minTarget, double maxTarget) {
282         DescriptiveStatistics ds = StatisticsUtils.
283             getDescriptiveStatistics(timeSeries);
284         minTSValuePreNorm = ds.getMin();

```

```

278         maxTSValuePreNorm = ds.getMax();
279         List<Double> ts = StatisticsUtils.normalizeMinMax(
                timeSeries, minTSValuePreNorm, maxTSValuePreNorm,
                minTarget, maxTarget);
280         timeSeries = ts;
281     }
282
283     /**
284      * Avalia a predicacao com os resultados preditos e os resultados
285      * esperados
286      * @param print imprime o resultado de cada observacao
287      */
288     public void evaluate(boolean print) {
289         predictionResults = PredictionUtils.evaluate(this.
                getNetwork(), this.getDataSet(), this, false, print);
290         predictionResults2 = PredictionUtils.evaluate(this.
                getNetwork(), this.getDataSet(), this, true, print);
291     }
292
293     public void evaluate() {
294         evaluate(false);
295     }
296
297     public List<List<Double>> getPredictionResults() {
298         return predictionResults;
299     }
300
301     public List<List<Double>> getPredictionResultsNormalizesd() {
302         return predictionResults2;
303     }
304
305     public List<Double> getTimeSeries() {
306         return timeSeries;
307     }
308
309     public TemporalMLDataSet getTrainSet() {
310         return trainSet;
311     }
312
313     public TemporalMLDataSet getDataSet() {
314         return dataSet;
315     }
316
317     public BasicNetwork getNetwork() {
318         return network;
319     }

```

```
320     public double getFinalTrainError() {
321         return finalTrainError;
322     }
323
324     public boolean isRecurrent() {
325         return recurrent;
326     }
327
328     public List<Double> getTrainErrors() {
329         return trainErrors;
330     }
331
332     public int getTrainSize() {
333         return trainSize;
334     }
335
336     public int getTrainEpochs() {
337         return trainEpochs;
338     }
339
340     public long getTrainTime() {
341         return trainTime;
342     }
343
344     public Double getMinTSValuePreNorm() {
345         return minTSValuePreNorm;
346     }
347
348     public Double getMaxTSValuePreNorm() {
349         return maxTSValuePreNorm;
350     }
351 }
```

Código A.3: Classe de utilidades usada para salvar os resultados dos experimentos

```

1  package encog.prediction.utils;
2
3  import java.io.BufferedWriter;
4  import java.io.File;
5  import java.io.FileWriter;
6  import java.io.IOException;
7  import java.util.Collections;
8  import java.util.LinkedList;
9  import java.util.List;
10
11 import org.encog.persist.EncogDirectoryPersistence;
12
13 import encog.prediction.nn.Prediction;
14
15 /**
16  * Classe de utilidades para salvar os resultados.
17  * @author Tiago
18  */
19 public class FileUtils {
20     public static final String PATH = "results" + File.separator +
21         Prediction.DATA_SET_FOLDER;
22     public static final String FNN = "fnn";
23     public static final String RNN = "rnn";
24     public static final String MSE_RESULTS = "NRMSE_RESULTS.txt";
25     public static final String COMPLEXITY_RESULTS = "COMPLEXITY_RESULTS
26         .txt";
27
28     public final static void savePrediction(Prediction prediction,
29         String fileName) {
30         try {
31             File file = new File(PATH + File.separator + FNN +
32                 File.separator + fileName);
33             String egName = fileName.substring(0, fileName.
34                 lastIndexOf('.') + ".eg");
35             File egFile = new File(PATH + File.separator + FNN
36                 + File.separator + egName);
37
38             if(fileName.charAt(0) == 'r') {
39                 file = new File(PATH + File.separator + RNN
40                     + File.separator + fileName);
41                 egFile = new File(PATH + File.separator +
42                     RNN + File.separator + egName);
43             }
44
45             if(!file.exists()) {
46                 file.createNewFile();

```

```

39         }
40
41         FileWriter fw = new FileWriter(file.getAbsolutePath()
42         ());
43         BufferedWriter bw = new BufferedWriter(fw);
44
45         String nn = fileName;
46         String type = "FeedForward";
47         String layers = "[ ";
48         String minTSValuePreNorm = String.valueOf(
49         prediction.getMinTSValuePreNorm());
50         String maxTSValuePreNorm = String.valueOf(
51         prediction.getMaxTSValuePreNorm());
52         String trainErrors = prediction.getTrainErrors().
53         toString();
54         String trainSize = String.valueOf(prediction.
55         getTrainSize());
56         String trainEpochs = String.valueOf(prediction.
57         getTrainEpochs());
58         String trainTime = String.valueOf(prediction.
59         getTrainTime());
60         String out = prediction.getPredictionResults().get
61         (0).toString();
62         String correct = prediction.getPredictionResults().
63         get(1).toString();
64         String error = prediction.getPredictionResults().
65         get(2).toString();
66         String residual = prediction.getPredictionResults()
67         .get(3).toString();
68         String absError = prediction.getPredictionResults()
69         .get(4).toString();
70
71         List<Double> miscList = prediction.
72         getPredictionResults().get(5);
73         String averageError = String.valueOf(miscList.get
74         (0));
75         String minError = String.valueOf(miscList.get(1));
76         String maxError = String.valueOf(miscList.get(2));
77         String averageResiduals = String.valueOf(miscList.
78         get(3));
79         String mse = String.valueOf(miscList.get(4));
80         String rmse = String.valueOf(miscList.get(5));
81         String nrmse = String.valueOf(miscList.get(6));
82         String mae = String.valueOf(miscList.get(7));
83         String mape = String.valueOf(miscList.get(8));
84         String smape = String.valueOf(miscList.get(9));
85         //////////////////////////////////////////

```

```

71      // valores depois de normalizar de volta
72      String out2 = prediction.
        getPredictionResultsNormalized().get(0).
        toString();
73      String correct2 = prediction.
        getPredictionResultsNormalized().get(1).
        toString();
74      String error2 = prediction.
        getPredictionResultsNormalized().get(2).
        toString();
75      String residual2 = prediction.
        getPredictionResultsNormalized().get(3).
        toString();
76      String absError2 = prediction.
        getPredictionResultsNormalized().get(4).
        toString();

77
78      List<Double> miscList2 = prediction.
        getPredictionResultsNormalized().get(5);
79      String averageError2 = String.valueOf(miscList2.get
        (0));
80      String minError2 = String.valueOf(miscList2.get(1))
        ;
81      String maxError2 = String.valueOf(miscList2.get(2))
        ;
82      String averageResiduals2 = String.valueOf(miscList2
        .get(3));
83      String mse2 = String.valueOf(miscList2.get(4));
84      String rmse2 = String.valueOf(miscList2.get(5));
85      String nrmse2 = String.valueOf(miscList2.get(6));
86      String mae2 = String.valueOf(miscList2.get(7));
87      String mape2 = String.valueOf(miscList2.get(8));
88      String smape2 = String.valueOf(miscList2.get(9));
89
90      if(prediction.isRecurrent()) {
91          type = "Recurrent-Jordan";
92      }
93
94      for(int i = 0; i < prediction.getNetwork().
        getLayerCount(); i++) {
95          layers = layers + prediction.getNetwork().
            getLayerNeuronCount(i) + " ";
96      }
97      layers = layers + "]";
98      bw.write("nn = " + nn);

        bw.

        newLine();

```

```

99         bw.write("type = " + type);
100
101         newLine();
102         bw.write("Layers = " + layers);
103
104         bw.newLine();
105         bw.write("minTSValuePreNorm = " + minTSValuePreNorm);
106         bw.newLine();
107         bw.write("maxTSValuePreNorm = " + maxTSValuePreNorm);
108         bw.newLine();
109         bw.write("trainErrors = " + trainErrors);
110         bw.newLine();
111         bw.write("trainSize = " + trainSize);
112         bw.newLine();
113         bw.write("trainEpochs = " + trainEpochs);
114         bw.newLine();
115         bw.write("trainTime = " + trainTime);
116         bw.newLine();
117         bw.write("out = " + out);
118
119         bw.newLine();
120         bw.write("correct = " + correct);
121         bw.newLine();
122         bw.write("errors = " + error);
123         bw.newLine();
124         bw.write("residuals = " + residual);
125         bw.newLine();
126         bw.write("absErrors = " + absError);
127         bw.newLine();
128         bw.write("averageError = " + averageError);
129         bw.newLine();
130         bw.write("minError = " + minError);
131         bw.newLine();
132         bw.write("maxError = " + maxError);
133         bw.newLine();
134         bw.write("averageResiduals = " + averageResiduals);
135         bw.newLine();
136         bw.write("MSE = " + mse);
137
138         bw.newLine();
139         bw.write("RMSE = " + rmse);
140
141         bw.newLine();
142         bw.write("NRMSE = " + nrmse);
143         bw.newLine();
144         bw.write("MAE = " + mae);
145
146         bw.newLine();

```

```

120         bw.write("MAPE = " + mape);
                                                    bw.
                newLine();
121         bw.write("SMAPE = " + smape);
                                                    bw.newLine();
122         bw.write("-----
                "); bw.newLine();
123         bw.write("- valores depois de normalizar de volta -
                "); bw.newLine();
124         bw.write("-----
                "); bw.newLine();
125         bw.write("out = " + out2);
                                                    bw.
                newLine();
126         bw.write("correct = " + correct2);
                                                    bw.newLine();
127         bw.write("errors = " + error2);
                                                    bw.newLine();
128         bw.write("residuals = " + residual2);
                                                    bw.newLine();
129         bw.write("absErrors = " + absError2);
                                                    bw.newLine();
130         bw.write("averageError = " + averageError2);
                                                    bw.newLine();
131         bw.write("minError = " + minError2);
                                                    bw.newLine();
132         bw.write("maxError = " + maxError2);
                                                    bw.newLine();
133         bw.write("averageResiduals = " + averageResiduals2)
                ; bw.newLine();
134         bw.write("MSE = " + mse2);
                                                    bw.
                newLine();
135         bw.write("RMSE = " + rmse2);
                                                    bw.newLine();
136         bw.write("NRMSE = " + nrmse2);
                                                    bw.newLine();
137         bw.write("MAE = " + mae2);
                                                    bw.
                newLine();
138         bw.write("MAPE = " + mape2);
                                                    bw.newLine();
139         bw.write("SMAPE = " + smape2);
                                                    bw.newLine();
140         bw.close();
141         // Salva o EG file , do EncogDirectoryPersistence

```

```

142         EncogDirectoryPersistence.saveObject(egFile,
143         prediction.getNetwork());
144     } catch(IOException e) {
145         e.printStackTrace();
146     }
147 }
148 /**
149  * Salva os MSE ordenados de forma crescente
150  * @param results
151  */
152 public static final void saveMSEResults(List<ResultsTuple> results,
153     boolean isRecurrent) {
154     try {
155         File file = new File(PATH + File.separator + FNN +
156             File.separator + MSE_RESULTS);
157
158         if(isRecurrent) {
159             file = new File(PATH + File.separator + RNN
160                 + File.separator + MSE_RESULTS);
161         }
162
163         if(!file.exists()) {
164             file.createNewFile();
165         }
166
167         FileWriter fw = new FileWriter(file.getAbsolutePath()
168             ());
169         BufferedWriter bw = new BufferedWriter(fw);
170         bw.write("NAME\t\t\t\t\tNRMSE");
171         bw.newLine();
172         for(ResultsTuple rt : results) {
173             bw.write(rt.toString());
174             bw.newLine();
175         }
176         bw.close();
177     } catch(IOException e) {
178         e.printStackTrace();
179     }
180 }
181 /**
182  * Salva os graficos de acordo com a complexidade da rede (numero
183  * de neuronios)
184  * @param results
185  */

```

```

182     public static void saveComplexityPerNeuronsResults(List<
           ResultsTuple> results, boolean isRecurrent) {
183         try {
184             LinkedList<Integer> windowSizeX = new LinkedList<
                   Integer>(); // nro neuronios na entrada
185             LinkedList<Double> windowSizeY = new LinkedList<
                   Double>(); // erro
186             LinkedList<Integer> complexityX = new LinkedList<
                   Integer>(); // nro neuronios
187             LinkedList<Double> complexityY = new LinkedList<
                   Double>(); // erro
188             LinkedList<WindowSizeTuple> wstList = new
                   LinkedList<WindowSizeTuple>();
189             LinkedList<NetworkComplexityTuple> nctList = new
                   LinkedList<NetworkComplexityTuple>();
190             for(ResultsTuple rt : results) {
191                 WindowSizeTuple wst = new WindowSizeTuple(
                       rt.mse, rt.name, rt.layers[0]);
192                 wstList.add(wst);
193                 NetworkComplexityTuple nct = new
                       NetworkComplexityTuple(rt.mse, rt.name,
                       rt.getNetworkComplexity());
194                 nctList.add(nct);
195             }
196
197             Collections.sort(wstList);
198             Collections.sort(nctList);
199
200             int i = 0;
201             while(i < wstList.size()) {
202                 WindowSizeTuple wst = wstList.get(i);
203                 int valor = wst.windowSize;
204                 i++;
205                 double media = wst.mse;
206                 int nroValores = 1;
207                 while(i < wstList.size() && valor ==
                       wstList.get(i).windowSize) {
208                     media += wstList.get(i).mse;
209                     nroValores++;
210                     i++;
211                 }
212                 media = media/nroValores;
213                 windowSizeX.add(valor);
214                 windowSizeY.add(media);
215             }
216             i = 0;
217             while(i < nctList.size()) {

```

```

218         NetworkComplexityTuple nct = nctList.get(i)
219         ;
220         int valor = nct.netComplexity;
221         i++;
222
223         double media = nct.mse;
224         int nroValores = 1;
225         while(i < nctList.size() && valor ==
226             nctList.get(i).netComplexity) {
227             media += nctList.get(i).mse;
228             nroValores++;
229             i++;
230         }
231         media = media/nroValores;
232
233         complexityX.add(valor);
234         complexityY.add(media);
235     }
236
237     File file = new File(PATH + File.separator + FNN +
238         File.separator + COMPLEXITY_RESULTS);
239     if(isRecurrent) {
240         file = new File(PATH + File.separator + RNN
241             + File.separator + COMPLEXITY_RESULTS);
242     }
243
244     if(!file.exists()) {
245         file.createNewFile();
246     }
247
248     FileWriter fw = new FileWriter(file.getAbsolutePath()
249         ());
250     BufferedWriter bw = new BufferedWriter(fw);
251     bw.write("windowSizeX = " + windowSizeX);
252     bw.newLine();
253     bw.write("windowSizeY = " + windowSizeY);
254     bw.newLine();
255     bw.write("complexityX = " + complexityX);
256     bw.newLine();
257     bw.write("complexityY = " + complexityY);
258     bw.close();
259 } catch(IOException e) {
260     e.printStackTrace();
261 }
262
263 private FileUtils() { }

```

```

260 }
261 //-----
262 package encog.prediction.utils;
263
264 import java.util.Comparator;
265
266 /**
267  * Comparador para verificar os resultados do tamanho da janela de entrada.
268  * @author Tiago
269  */
270 public class WindowSizeTuple implements Comparator<WindowSizeTuple>,
    Comparable<WindowSizeTuple>{
271     public double mse;
272     public String name;
273     public int windowSize;
274
275     public WindowSizeTuple(double mse, String name, int windowSize) {
276         this.mse = mse;
277         this.name = name;
278         this.windowSize = windowSize;
279     }
280
281     @Override
282     public int compare(WindowSizeTuple r1, WindowSizeTuple r2) {
283         if(r1.windowSize == r2.windowSize) return 0;
284         else if(r1.windowSize < r2.windowSize) return -1;
285         else return 1;
286     }
287
288     @Override
289     public int compareTo(WindowSizeTuple o) {
290         return compare(this, o);
291     }
292 }
293
294 /**
295  * Classe para armazenar e comparar a complexidade, nome e erro da RNA.
296  * @author Tiago
297  */
298 public class NetworkComplexityTuple implements Comparator<
    NetworkComplexityTuple>, Comparable<NetworkComplexityTuple>{
299     public double mse;
300     public String name;
301     public int netComplexity;
302
303     public NetworkComplexityTuple(double mse, String name, int
        netComplexity) {

```

```

304         this.mse = mse;
305         this.name = name;
306         this.netComplexity = netComplexity;
307     }
308
309     @Override
310     public int compare(NetworkComplexityTuple r1,
311                     NetworkComplexityTuple r2) {
312         if(r1.netComplexity == r2.netComplexity) return 0;
313         else if(r1.netComplexity < r2.netComplexity) return -1;
314         else return 1;
315     }
316
317     @Override
318     public int compareTo(NetworkComplexityTuple o) {
319         return compare(this, o);
320     }
321 }
322
323 /**
324  * Comparador para verificar os resultados do erro.
325  * @author Tiago
326  */
327 public class ResultsTuple implements Comparator<ResultsTuple>, Comparable<
ResultsTuple>{
328     public double mse;
329     public String name;
330     public int [] layers;
331
332     public ResultsTuple(double mse, String name, int [] layers) {
333         this.mse = mse;
334         this.name = name;
335         this.layers = layers.clone();
336     }
337
338     /**
339     * Retorna a soma do numero de neuronios
340     * @return
341     */
342     public int getNetworkComplexity() {
343         int nc = 0;
344         for(int i = 0; i < layers.length; i++) {
345             nc += layers[i];
346         }
347         return nc;
348     }

```

```
349         @Override
350         public String toString() {
351             return name + "\t\t\t" + mse;
352         }
353
354         @Override
355         public int compare(ResultsTuple r1, ResultsTuple r2) {
356             if(r1.mse == r2.mse) return 0;
357             else if(r1.mse < r2.mse) return -1;
358             else return 1;
359         }
360
361         @Override
362         public int compareTo(ResultsTuple o) {
363             return compare(this, o);
364         }
365     }
```

Código A.4: Classe de utilidades para avaliar os resultados das predições

```

1 package encog.prediction.utils;
2
3 import java.util.LinkedList;
4 import java.util.List;
5 import org.apache.commons.math3.stat.descriptive.DescriptiveStatistics;
6 import org.encog.ml.MLRegression;
7 import org.encog.ml.data.MLData;
8 import org.encog.ml.data.MLDataPair;
9 import org.encog.ml.data.MLDataSet;
10 import org.encog.util.simple.EncogUtility;
11 import encog.prediction.nn.Prediction;
12 import encog.prediction.nn.TestPrediction;
13
14 /**
15  * Classe de utilidades para avaliar as predicoes.
16  * @author Tiago
17  */
18 public class PredictionUtils {
19     /**
20      * Evaluate the network and display (to the console) the output for
21      * every
22      * value in the training set. Displays ideal and actual.
23      *
24      * @param network
25      *      The network to evaluate.
26      * @param training
27      *      The training set to evaluate.
28      */
29     public static final List<List<Double>> evaluate(final MLRegression
30         network, final MLDataSet training, Prediction prediction,
31         boolean invertNormalize, boolean print) {
32         LinkedList<Double> outList = new LinkedList<>();
33         LinkedList<Double> correctList = new LinkedList<>();
34         LinkedList<Double> errorList = new LinkedList<>();
35         LinkedList<Double> residualList = new LinkedList<>();
36         LinkedList<Double> absErrorList = new LinkedList<>();
37         Double out, correct, error, residual, absError;
38         Double sumAbsError = 0d, sqrSumError = 0d, mapeError = 0d,
39             smapeError = 0d;
40         int smapeSum = 0;
41
42         for (final MLDataPair pair : training) {
43             final MLData output = network.compute(pair.getInput
44                 ());
45             if (print) {
46                 System.out.println(

```

```

42         "Input=" + EncogUtility.
            formatNeuralData(pair.getInput()
            )
43         + "\tActual=" + EncogUtility.
            formatNeuralData(output)
44         + "\tIdeal=" + EncogUtility.
            formatNeuralData(pair.getIdeal()
            )
45         + "\tError=" + (output.getData(0) -
            pair.getIdeal().getData(0))
46     );
47 }
48 out = output.getData(0);
49 correct = pair.getIdeal().getData(0);
50
51 if(invertNormalize) {
52     out = StatisticsUtils.minMaxI(output.
        getData(0), prediction.
        getMinTSValuePreNorm(), prediction.
        getMaxTSValuePreNorm(), TestPrediction.
        MIN_NORMALIZED_VALUE, TestPrediction.
        MAX_NORMALIZED_VALUE);
53     correct = StatisticsUtils.minMaxI(pair.
        getIdeal().getData(0), prediction.
        getMinTSValuePreNorm(), prediction.
        getMaxTSValuePreNorm(), TestPrediction.
        MIN_NORMALIZED_VALUE, TestPrediction.
        MAX_NORMALIZED_VALUE);
54 }
55
56 error = out - correct;
57 absError = Math.abs(error);
58 residual = correct - out;
59 outList.add(out);
60 correctList.add(correct);
61 errorList.add(error);
62 absErrorList.add(absError);
63 residualList.add(residual);
64 sumAbsError += absError;
65 sqrSumError += error * error;
66 mapeError += Math.abs(residual / correct);
67 if(correct + out != 0) {
68     smapeError += Math.abs(residual / (correct
        + out));
69     smapeSum++;
70 }
71 }

```

```

72         LinkedList<List<Double>> result = new LinkedList<>();
73         result.add(outList);
74         result.add(correctList);
75         result.add(errorList);
76         result.add(residualList);
77         result.add(absErrorList);
78         DescriptiveStatistics dsError = StatisticsUtils.
            getDescriptiveStatistics(errorList);
79         DescriptiveStatistics dsResidual = StatisticsUtils.
            getDescriptiveStatistics(residualList);
80         DescriptiveStatistics dsCorrect = StatisticsUtils.
            getDescriptiveStatistics(correctList);
81         Double averageError = dsError.getMean();
82         Double minError = dsError.getMin();
83         Double maxError = dsError.getMax();
84         Double averageResiduals = dsResidual.getMean();
85         double size = training.size();
86         Double mse = sqrSumError / size;
87         Double rmse = Math.sqrt(mse);
88         Double nrmse = rmse / (dsCorrect.getMax() - dsCorrect.
            getMin());
89         Double mae = sumAbsError / size;
90         Double mape = (mapeError / size) * 100;
91         Double smape = (smapeError / smapeSum) * 200;
92         // lista com o resto dos valores
93         List<Double> miscList = new LinkedList<Double>();
94         miscList.add(averageError);          miscList.add(
            minError);          miscList.add(maxError);
95         miscList.add(averageResiduals); miscList.add(mse);
            miscList.add(rmse);
96         miscList.add(nrmse);          miscList.add(mae);
            miscList.add(mape);
97         miscList.add(smape);
98         result.add(miscList);
99         return result;
100     }
101
102     public static final List<List<Double>> evaluate(final MLRegression
        network, final MLDataSet training, Prediction prediction,
        boolean invertNormalize) {
103         return evaluate(network, training, prediction,
            invertNormalize, false);
104     }
105
106     private PredictionUtils() {}
107 }

```

Código A.5: Classe com métodos estatísticos de normalização

```

1  package encog.prediction.utils;
2
3  import java.util.LinkedList;
4  import java.util.List;
5  import org.apache.commons.math3.stat.descriptive.DescriptiveStatistics;
6
7  /**
8   * Classe de utilidades para avaliacoess e funcoes estatisticas.
9   * @author Tiago
10  */
11  public class StatisticsUtils {
12      public static final double zScore(double x, double mean, double
13          standardDeviation) {
14          return (x - mean) / standardDeviation;
15      }
16
17      public static final List<Double> normalizeZScore(List<Double> array
18          ) {
19          DescriptiveStatistics ds = getDescriptiveStatistics(array);
20          Double avg = ds.getMean();
21          Double sd = ds.getStandardDeviation();
22          Double aNorm = null;
23          LinkedList<Double> normList = new LinkedList<>();
24
25          for(Double a : array) {
26              aNorm = zScore(a, avg, sd);
27              normList.add(aNorm);
28          }
29
30          return normList;
31
32      public static final double minMax(double x, double minValue, double
33          maxValue, double minTarget, double maxTarget) {
34          return ((maxTarget - minTarget) * (x - minValue)) / (
35              maxValue - minValue) + minTarget;
36      }
37
38      /**
39       * funcao inversa da minMax
40       */
41      public static final double minMaxI(double x, double minValue,
42          double maxValue, double minTarget, double maxTarget) {
43          return ((maxValue - minValue) * (x - minTarget)) / (
44              maxTarget - minTarget) + minValue;
45      }
46  }

```

```

41
42     public static final List<Double> normalizeMinMax(List<Double> array
43         , double minTarget, double maxTarget) {
44         DescriptiveStatistics ds = getDescriptiveStatistics(array);
45         Double minValue = ds.getMin();
46         Double maxValue = ds.getMax();
47         return normalizeMinMax(array, minValue, maxValue, minTarget
48             , maxTarget);
49     }
50
51     public static final List<Double> normalizeMinMax(List<Double> array
52         , double minValue, double maxValue, double minTarget, double
53         maxTarget) {
54         Double aNorm = null;
55         LinkedList<Double> normList = new LinkedList<>();
56
57         for(Double a : array) {
58             aNorm = minMax(a, minValue, maxValue, minTarget,
59                 maxTarget);
60             normList.add(aNorm);
61         }
62
63         return normList;
64     }
65
66     public static final DescriptiveStatistics getDescriptiveStatistics(
67         List<Double> array) {
68         DescriptiveStatistics ds = new DescriptiveStatistics();
69         for(Double value : array) {
70             ds.addValue(value);
71         }
72         return ds;
73     }
74
75     private StatisticsUtils() {}
76 }

```

Código A.6: Função escrita em MATLAB, usando o DeepLearn Toolbox, para treinar um SAE com uma série temporal

```

1 function sae
2     format long
3     % TODO: passar as variaveis como parametro, pra nao ter que ficar
4     trocando na mao
5     pasta = 'teste_trafego_robH/uk-1d/';
6     data_set = 'internet-traffic-data-in-bits-fr-uk-1d.csv';
7     ts = csvread(data_set)';
8     ts = normalize_min_max(ts, 0.1, 0.9);
9     [nr, nc] = size(ts); % numero de linhas e numero de colunas da serie
10    temporal
11    train_size = floor(nc/2); % treinando com a metade da serie
12    pre_train_epochs = 900; % numero de epocas de pre-treinamento
13    train_epochs = 100; % numero de epocas de treinamento
14    in_dim = 5;
15    out_dim = 1;
16
17    [train_in, train_out] = create_data_set(ts(1:train_size), in_dim,
18        out_dim);
19    [test_in, test_out] = create_data_set(ts, in_dim, out_dim);
20    % PRE-TREINAMENTO - SAE - TREINAMENTO NAO SUPERVISIONADO
21    rand('state',0)
22    hidden_layers = [80 60 60 40];
23    sae = saesetup([in_dim hidden_layers out_dim]);
24    sae.ae{1}.activation_function = 'sigm';
25    sae.ae{1}.learningRate = 0.01; % 1
26    %sae.ae{1}.momentum = 0.0; % 0.5
27    sae.ae{1}.inputZeroMaskedFraction = 0.0; % 0.5
28
29    sae.ae{2}.activation_function = 'sigm';
30    sae.ae{2}.learningRate = 0.01; % 1
31    %sae.ae{2}.momentum = 0.0; % 0.5
32    sae.ae{2}.inputZeroMaskedFraction = 0.0; % 0.5
33
34    sae.ae{3}.activation_function = 'sigm';
35    sae.ae{3}.learningRate = 0.01; % 1
36    %sae.ae{3}.momentum = 0.0; % 0.5
37    sae.ae{3}.inputZeroMaskedFraction = 0.0; % 0.5
38
39    opts.numepochs = pre_train_epochs;
40    opts.batchsize = 1;
41    pre_train_time = tic;
42    sae = saetrain(sae, train_in, opts);
43    pre_train_time = toc(pre_train_time);
44    % CONVERTER PARA FFNN PARA TREINAMENTO SUPERVISIONADO
45    % Use the SDAE to initialize a FFNN

```

```

43     nn = nnsetup([in_dim hidden_layers out_dim]);
44     nn.activation_function = 'sigm';
45     nn.learningRate = 0.01;
46     nn.momentum = 0.00;
47     % add pretrained weights
48     nn.W{1} = sae.ae{1}.W{1};
49     nn.W{2} = sae.ae{2}.W{1};
50     % Train the FFNN
51     opts.numepochs = train_epochs;
52     opts.batchsize = 1;
53     train_time = tic;
54     [nn, L, train_errors] = nntrain(nn, train_in, train_out, opts);
55     train_time = toc(train_time);
56     layers = [in_dim hidden_layers out_dim];
57     results_train = test_data_set(nn, train_in, train_out); % resultados
                    com a serie treinada [1:train_size]
58     results_test = test_data_set(nn, test_in, test_out); % resultados com a
                    serie completa result_errors = results_test(:,3);
59
60     result_errors = results_test(:,3);
61     average_error = mean(result_errors);
62     min_error = min(result_errors);
63     max_error = max(result_errors);
64     average_residuals = mean(results_test(:,4));
65
66     mse = mse_calc(result_errors);
67
68     teste = ['teste_teste','SAE_1'];
69     dir_arquivo = [pasta, teste];
70     save(dir_arquivo, 'sae', 'nn', 'layers', 'train_errors', 'train_size',
        'pre_train_epochs', 'train_epochs', 'results_train', 'results_test',
        'mse', 'average_error', 'min_error', 'max_error', '
        average_residuals', 'train_time', 'pre_train_time');
71
72     [linhas, colunas] = size(results_test);
73     plot(linspace(1, linhas, linhas), ts(in_dim + 1:numel(ts)), 'b',
        linspace(1, linhas, linhas), results_test(:,1), 'r');
74     print(gcf, '-dpng', [dir_arquivo, '.png']);
75     plot(linspace(1, numel(train_errors), numel(train_errors)), train_errors,
        'g');
76     print(gcf, '-dpng', [dir_arquivo, '_erro.png']);
77
78     plot(linspace(1, train_size - in_dim, train_size - in_dim), ts(in_dim + 1:
        train_size), 'b', linspace(1, train_size - in_dim, train_size - in_dim),
        results_train(:,1), 'r');
79 end
80 %

```

```

81 function normalized = normalize_min-max(array , min , max)
82     # Normalize to [0 , 1]:
83     m = min(array);
84     range = max(array) - m;
85     array = (array - m) / range;
86     # Scale to [min,max]:
87     range2 = max - min;
88     normalized = (array*range2) + min;

```

A.2 Códigos do Algoritmo GPLNEURO

Código A.7: Classe principal do GPLNEURO

```

1  #
    ###-----###
2  # #### Gerenciamento Preditivo de Largura de Banda usando Redes Neurais (
    GPLNEURO) ####
3  #
    ###-----###
4  # Data: 18-Nov-2014
5  # Autor: Tiago Prado Oliveira
6  # Instituicao: Universidade Federal de Uberlandia (UFU) - Faculdade de
    Computacao (FACOM)
7  # Descricao: Programa que faz o gerenciamento adaptativo da largura de
    banda das interfaces de um roteador. O gerenciamento e' baseado na
    predicao do trafego dessas interfaces , alocando a largura de banda mais
    adequada de acordo com os valores futuros previstos .
8  # Objetivo: O objetivo e' evitar congestionamentos nas interfaces de
    comunicacao do roteador , tratando a largura de banda de forma mais
    justa , de forma a estabilizar o trafego transmitido pelo roteador .
9  # #####
10
11 import time , utils , sys
12 from monitoring import Monitoring
13 from prediction import Prediction
14 from analysis import Analysis
15 from control import Control
16 from threading import Thread
17
18 #### Classe do Gerenciador Preditivo de Largura de Banda
19 #### Nela e' criado uma Thread para o Monitoramento , que executa
    simultaneamente com o processo pai. No processo pai e' realizado a
    Predicao => Analise => Controle , de forma sequencial.
20 class GPLNEURO(Thread):

```

```

21     ### param 'monitor_only' flag para indicar se somente o modulo de
        monitoramento que sera executado, caso seja 'True', entao as
        etapas que envolvem predicao-analise-controlle nao serao
        executadas
22     def __init__(self, monitor_only=False):
23         Thread.__init__(self)
24         self.running = True
25         self.monitor_only = monitor_only
26
27     def run(self):
28         # monitoramento
29         self.monitoring_agent = Monitoring()
30         self.monitoring_agent.start()
31
32         while not self.monitor_only and self.running:
33             timestamp = time.time()
34
35             # predicao
36             nn = Prediction(utils.input_window)
37             predicted_bw = nn.predict();
38             print 'Prediction...', predicted_bw
39
40             if predicted_bw is not None:
41                 # analise
42                 analyser = Analysis()
43                 allocate_bw = analyser.analyse(predicted_bw
44                 );
45                 print 'Analysis...', allocate_bw
46                 # controle
47                 controler = Control()
48                 controler.allocate(allocate_bw);
49                 print 'Control...'
50
51                 # tempo que a predicao->analise->controlle durou
52                 gplab_time = time.time() - timestamp
53                 # espera ate no maximo 'utils.gplab_sleep' segundos
54                 , antes de repetir a iteracao
55                 utils.sleep_max(utils.gplab_sleep, gplab_time)
56                 print 'Fim loop...', gplab_time
57
58                 # espera o monitoramento acabar, antes de finalizar o
59                 GPLNEURO
60                 self.monitoring_agent.join()
61
62     ### Para a execucao do GPLNEURO
63     def stop(self):
64         self.monitoring_agent.stop()

```

```
62         self.running = False
63
64     # inicia o gerenciador GPLNEURO
65     if __name__ == '__main__':
66         gerenciador = GPLNEURO(False)
67         gerenciador.start()
68         time.sleep(130000)#36h
69         gerenciador.stop()
70         gerenciador.join()
```

Código A.8: Classe referente ao módulo **Monitoramento**

```

1  # -
2  # MONITORAMENTO
3  # -
4  # O monitoramento e' independente dos outros modulos. O modulo de
    monitoramento e' responsavel de coletar todas as observacoes que serao
    usadas nos modulos de treinamento e predicao.
5  # #####
6
7  import time, subprocess, re
8  import utils
9  from threading import Thread
10
11  ### Classe do Monitoramento
12  class Monitoring(Thread):
13      def __init__(self, version='2c', community='public', server='
        localhost'):
14          Thread.__init__(self)
15          self.running = True
16          self.version = version
17          self.community = community
18          self.server = server
19          self.walk_cmd = 'snmpwalk -v ' + self.version + ' -c ' +
                self.community + ' ' + self.server + ' '
20
21      def run(self):
22          # cria o cabeçalho do arquivo de dados do monitoramento
23          mnt_file = open(utils.mnt_file, 'w')
24          header = ['timestamp', 'ifInOctets-eth1', 'ifOutOctets-eth1',
                ', 'delay-eth1', 'ifInOctets-eth2', 'ifOutOctets-eth2',
                ', 'delay-eth2', 'ifInOctets-eth3', 'ifOutOctets-eth3',
                ', 'delay-eth3']
25          try:
26              mnt_file.write(', '.join(header) + '\n')
27          finally:
28              mnt_file.close()
29
30          while self.running:
31              timestamp = time.time()
32              # bytes que entraram e saíram nas interfaces
33              ifInOctets = self.getIfOctets('ifInOctets')
34              ifOutOctets = self.getIfOctets('ifOutOctets')
35              # atraso
36              delays = self.getDelay()
37              # salva tudo
38              row = [ repr(timestamp), ifInOctets[0], ifOutOctets
                    [0], delays[0], ifInOctets[1], ifOutOctets[1],

```

```

delays[1], ifInOctets[2], ifOutOctets[2], delays
[2] ]

39
40 mnt_file = open(utils.mnt_file, 'a')
41 try:
42     mnt_file.write(', '.join(row) + '\n')
43 finally:
44     mnt_file.close()
45
46 # tempo que o monitoramento durou, i.e, tempo desta
    iteracao
47 monitoring_time = time.time() - timestamp
48 # espera ate no maximo 'utils.monitoring_sleep'
    segundos
49 utils.sleep_max(utils.monitoring_sleep,
    monitoring_time)
50 print 'Monitoring...', monitoring_time
51
52 ### Retorna uma lista com os octetos que passaram em cada interface
    de comunicacao roteador-LANs
53 ## param 'oid' da IF-MIB; ex: ifInOctets, ifOutOctets
54 ## return ifOctets das interfaces [eth1, eth2, eth3]
55 def getIfOctets(self, oid):
56     snmpwalk = subprocess.Popen([self.walk_cmd + oid], stdout=
        subprocess.PIPE, shell=True)
57     (out, err) = snmpwalk.communicate()
58
59     if snmpwalk.returncode != 0 or err is not None:
60         if err is None:
61             err = ''
62         print 'Erro: ' + err
63         raise Exception('Erro no snmpwalk ifInOctets')
64
65     # pega somente os valores necessarios das interfaces
        necessarias
66     out = out.split('\n')[2:5]
67     return [ x.split()[-1] for x in out ]
68
69 ### Retorna o atraso atual de cada interface
70 ## return atraso das interfaces [eth1, eth2, eth3]
71 def getDelay(self):
72     delay1_cmd = 'ping -c 1 ' + utils.HOST1_IP
73     delay2_cmd = 'ping -c 1 ' + utils.HOST2_IP
74     delay3_cmd = 'ping -c 1 ' + utils.HOST3_IP
75
76     cp1 = utils.ConcurrentProcess(delay1_cmd)
77     cp2 = utils.ConcurrentProcess(delay2_cmd)

```

```
78         cp3 = utils.ConcurrentProcess(delay3_cmd)
79
80         cp1.start(); cp2.start(); cp3.start()
81         cp1.join(); cp2.join(); cp3.join()
82
83         (out1, err1) = cp1.getResult()
84         (out2, err2) = cp2.getResult()
85         (out3, err3) = cp3.getResult()
86
87         delay1 = '-1'; delay2 = '-1'; delay3 = '-1'
88         if err1 is None:
89             delay1 = self._delay_str(out1)
90         if err2 is None:
91             delay2 = self._delay_str(out2)
92         if err3 is None:
93             delay3 = self._delay_str(out3)
94
95         return [delay1, delay2, delay3]
96
97     def _delay_str(self, ds):
98         if '0 received' in ds:
99             return '-1'
100         else:
101             result = re.search('time=(.*) ms', ds)
102             return result.group(1)
103
104     #### Para o monitoramento
105     def stop(self):
106         self.running = False
```

Código A.9: Classe referente ao módulo **Predição**

```

1  # - -
2  # PREDICAO
3  # - -
4  # O treinamento e' feito com os dados obtidos no monitoramento. Somente
   # apos o treinamento que a rede neural podera realizar a predicao.
5  # A predicao e' feita a partir dos 'x_window' valores mais recentes
   # coletados no modulo de monitoramento. A predicao, entao, passa o valor
   # futuro previsto para o modulo de analise.
6  # #####
7
8  import os, subprocess, shutil, time, utils
9  from monitoring import Monitoring
10
11  ### Classe de Predicao
12  class Prediction:
13      def __init__(self, x_window):
14          # tamanho da janela de entrada que sera usada na predicao
15          self.x_window = x_window
16
17      ### Realiza a predicao para cada interface de comunicacao roteador-
   # LANs [eth1, 2 e 3]
18      ## return lista com os valores previstos (bytes) de cada interface
19      def predict(self):
20          # lista com as ultimas linhas do arquivo de monitoramento
21          mnt_lines = []
22          try:
23              shutil.copy2(utils.mnt_file, utils.mnt_tmp_file)
24              mnt_tf = open(utils.mnt_tmp_file, 'r')
25              br = utils.BackwardsReader(mnt_tf)
26              # pega os 'x_window+1' valores anteriores, para a
   # predicao do proximo valor
27              for i in xrange(self.x_window + 1):
28                  line = br.readline();
29                  if line is None:
30                      return None
31                  elif line[0] == '\t':
32                      # caso essa linha seja o cabecalho
33                      return None
34                  line.strip('\n')
35                  line.strip('\r')
36                  line = line.split(',')
37                  line = [ int(float(x)) for x in line ]
38                  # insere o 'line' no comeco da lista
39                  mnt_lines[0:0] = [line]
40          except:
41              return None

```

```

42     # se pegou todas as linhas, entao pode fazer predicao
43     nn_in1 = [] # entradas para a rede neural do eth1
44     nn_in2 = [] # ... eth2
45     nn_in3 = [] # ... eth3
46
47     for i in xrange(len(mnt_lines) - 1):
48         # trafego de cada interface, que passou na ultima
49         # iteracao (proximo - atual)
50         in1 = mnt_lines[i+1][1] - mnt_lines[i][1]
51         out1 = mnt_lines[i+1][2] - mnt_lines[i][2]
52         in2 = mnt_lines[i+1][4] - mnt_lines[i][4]
53         out2 = mnt_lines[i+1][5] - mnt_lines[i][5]
54         in3 = mnt_lines[i+1][7] - mnt_lines[i][7]
55         out3 = mnt_lines[i+1][8] - mnt_lines[i][8]
56         # verifica se o Counter de 32 bits (4294967296)
57         # voltou para zero
58         _max = 4294967296 - 1
59         if in1 < 0:
60             in1 = in1 + _max
61         if out1 < 0:
62             out1 = out1 + _max
63         if in2 < 0:
64             in2 = in2 + _max
65         if out2 < 0:
66             out2 = out2 + _max
67         if in3 < 0:
68             in3 = in3 + _max
69         if out3 < 0:
70             out3 = out3 + _max
71         # soma o trafego de entrada com o de saida, para
72         # obter o trafego total
73         tr1 = in1 + out1
74         tr2 = in2 + out2
75         tr3 = in3 + out3
76
77         nn_in1.append(str(tr1))
78         nn_in2.append(str(tr2))
79         nn_in3.append(str(tr3))
80
81     # faz a predicao
82     snn_in1 = ','.join(nn_in1)
83     snn_in2 = ','.join(nn_in2)
84     snn_in3 = ','.join(nn_in3)
85     p_cmd = 'java -jar rnn.jar ' + snn_in1 + ' ' + snn_in2 + ' '
86           + snn_in3
87     pred = utils.ConcurrentProcess(p_cmd)
88     pred.start()

```

```
85         pred.join()
86
87         (out, err) = pred.getResult()
88         if err is not None:
89             return None
90         # salva a previsao
91         pred_file = open('prediction.csv', 'a')
92         try:
93             pred_file.write(str(time.time()) + ',' + out)
94         finally:
95             pred_file.close()
96         # pega resultado da predicao e passa pra analise
97         predicted_bw = [ int(x) for x in out.split(',') ]
98         return predicted_bw
```

Código A.10: Classe utilizada para criar a JNN usada pelo módulo **Predição**

```

1  package com.ufu.encog.nn.prediction;
2
3  import java.io.File;
4  import org.encog.neural.networks.BasicNetwork;
5  import org.encog.persist.EncogDirectoryPersistence;
6
7  /**
8   * Faz a predicao para o algoritmo GPLNEURO
9   * @author Tiago
10  */
11  public class RNNPrediction {
12      public static final double MIN_NORMALIZED_VALUE = 0.1d; // nao eh
13          zero pra nao dar problema de divisao por zero
14      public static final double MAX_NORMALIZED_VALUE = 0.9d;
15      public static final double MIN_PRE_NORMALIZED_VALUE = 800000000; //
16          TODO: mudar isso pra ler do arquivo
17      public static final double MAX_PRE_NORMALIZED_VALUE = 316; // TODO:
18          mudar isso pra ler do arquivo
19
20      public static final double minMax(double x, double minValue, double
21          maxValue, double minTarget, double maxTarget) {
22          return ((maxTarget - minTarget) * (x - minValue)) / (
23              maxValue - minValue) + minTarget;
24      }
25
26      public static final double minMaxI(double x, double minValue,
27          double maxValue, double minTarget, double maxTarget) {
28          return ((maxValue - minValue) * (x - minTarget)) / (
29              maxTarget - minTarget) + minValue;
30      }
31
32      public static void main(String[] args) {
33          // args[0] = "valor1, valor2, valor3, valor4, valor5" do
34              eth1
35          double[] entrada1 = { 0, 0, 0, 0, 0 };
36          double[] saida1 = { 0, 0, 0, 0, 0 };
37          double[] entrada2 = { 0, 0, 0, 0, 0 };
38          double[] saida2 = { 0, 0, 0, 0, 0 };
39          double[] entrada3 = { 0, 0, 0, 0, 0 };
40          double[] saida3 = { 0, 0, 0, 0, 0 };
41
42          String [] sEntrada1 = args[0].split(",");
43          String [] sEntrada2 = args[1].split(",");
44          String [] sEntrada3 = args[2].split(",");
45
46          for(int i = 0; i < entrada1.length; i++) {

```


Código A.11: Classe referente ao módulo **Análise**

```

1  # -      -
2  # ANALISE
3  # -      -
4  # A analise executa avaliaco es sobre o trafego que chega nas interfaces de
   comunicacao do roteador. A alteracao da largura de banda depende do
   trafego previsto , entao , o modulo de analise verifica o trafego futuro e
   decide a nova largura de banda que sera usada em cada interface do
   roteador .
5  # #####
6
7  import math, utils
8
9  ### Classe de Analise
10 class Analysis:
11     ### Analisa a largura de banda que sera alocada para cada interface
   do roteador, dependendo do algoritmo de decisao , essa classe
   pode ser mais complexa
12     ## param 'predicted' lista com os valores preditos (bytes) de cada
   interface
13     ## return lista com a largura de banda (bytes) que sera alocada
   para cada interface
14     def analyse(self , predicted):
15         allocate = []
16         for future_value in predicted:
17             # acrescenta o valor do erro medio da predicao
18             allocate_bw = future_value * (1 + utils.nnp_err)
19             # deve alocar a largura de banda que sera utilizada
   nos proximos 'gplab_sleep' segundos
20             allocate_bw = int(math.ceil(allocate_bw / float(
   utils.gplab_sleep)))
21             # nao deixa passar da largura de banda maxima nem
   abaixo da minima
22             if allocate_bw > utils.intf_max_bw:
23                 allocate_bw = utils.intf_max_bw
24             elif allocate_bw < utils.intf_min_bw:
25                 allocate_bw = utils.intf_min_bw
26             allocate.append(allocate_bw)
27
28     return allocate

```

Código A.12: Classe referente ao módulo **Controle**

```

1  # - -
2  # CONTROLE
3  # - -
4  # O controle executa, de fato, o controle do trafego, ou seja, o modulo de
   controle e' responsavel por alocar a largura de banda de cada interface
   de comunicacao do roteador. O controle recebe, do modulo de analise, as
   novas larguras de banda e utiliza o Hierarchical Token Bucket (HTB), que
   e' uma disciplina de escalonamento de classes, para alterar a largura
   de banda das interfaces. Por fim, o fluxo volta de volta para o modulo
   de predicao, que segue para a analise e retorna ao controle, e assim por
   diante.
5  # #####
6
7  import os, utils
8
9  ### Classe de Controle
10 class Control:
11     def __init__(self):
12         # armazena os valores atuais da largura de banda
13         self.current_bw = [-1, -1, -1]
14
15         ### Realisa a alocao da largura de banda das interfaces de
           comunicacao do roteador
16         ## param 'bw' lista com a largura de banda (bytes) que sera alocada
           para as interfaces [bw:eth1, bw:eth2, bw:eth3]
17     def allocate(self, bw):
18         # se a largura de banda for a mesma de antes, entao nao
           muda nada
19         if self.current_bw != bw:
20             self.current_bw = list(bw)
21             print self.current_bw
22             list_cmd = self._get_cmd_list()
23             for cmd in list_cmd:
24                 os.system('; '.join(cmd))
25
26         ### Retorna a lista de comandos que serao executados para a
           alocao da largura de banda
27     def _get_cmd_list(self):
28         # converte os bytes para kbits
29         bw = [str(x*8/1024.) for x in self.current_bw]
30         print '\n\nimprime bw',bw
31
32         eth1 = 'eth1'
33         eth1_bw = str(bw[0])
34         eth1_ip = utils.HOST1_IP
35         eth1_classid = eth1[-1]

```

```

36         eth1_cmd = self._cmd_tc_list(eth1, eth1_bw, eth1_ip,
37                                     eth1_classid)
38
39         eth2 = 'eth2'
40         eth2_bw = str(bw[1])
41         eth2_ip = utils.HOST2_IP
42         eth2_classid = eth2[-1]
43         eth2_cmd = self._cmd_tc_list(eth2, eth2_bw, eth2_ip,
44                                     eth2_classid)
45
46         eth3 = 'eth3'
47         eth3_bw = str(bw[2])
48         eth3_ip = utils.HOST3_IP
49         eth3_classid = eth3[-1]
50         eth3_cmd = self._cmd_tc_list(eth3, eth3_bw, eth3_ip,
51                                     eth3_classid)
52
53         return [eth1_cmd, eth2_cmd, eth3_cmd]
54
55     #### Cria os comandos para o controle de trafego com o HTB
56     ## parm interface, largura de banda (kbits), ip, id da classe
57     def _cmd_tc_list(self, intf, intf_bw, intf_ip, intf_classid):
58         flush = 'tc qdisc del dev ' + intf + ' root'
59         add_qdisc = 'tc qdisc add dev ' + intf + ' root handle 1:
60                     htb'
61         add_class = 'tc class add dev ' + intf + ' parent 1:
62                     classid 1:' + intf_classid + ' htb rate ' + intf_bw + '
63                     kbit'
64         add_filter = 'tc filter add dev ' + intf + ' protocol ip
65                     parent 1:0 prio 1 u32 match ip dst ' + intf_ip + '/24
66                     flowid 1:' + intf_classid
67
68         return [flush, add_qdisc, add_class, add_filter]

```

Código A.13: Classe de utilidades usada pelo GPLNEURO

```

1  # - -
2  # UTILIDADES
3  # - -
4  # Modulo de utilidades , funcoes variadas e variaveis globais
5  # #####
6
7  import time, subprocess, random, string
8  from threading import Thread
9
10 # Servidor
11 ROUTER_IP = '200.19.151.7'
12 # Gateway
13 ROUTER_eth1 = '192.168.1.1'
14 ROUTER_eth2 = '192.168.2.1'
15 ROUTER_eth3 = '192.168.3.1'
16 # Maquinas das diferentes LANs
17 HOST1_IP = '192.168.1.121'
18 HOST2_IP = '192.168.2.122'
19 HOST3_IP = '192.168.3.120'
20 # Arquivo com o trafego monitorado
21 mnt_file = 'monitoring.csv'
22 mnt_tmp_file = 'monitoring_tmp.csv'
23 # Tempo que as operacoes devem esperar antes de serem realizadas novamente
24 monitoring_sleep = 60
25 gplab_sleep = 60
26 # Largura de banda (bytes) maxima das interfaces - 100mbit=13107200bytes
    =12.5MB
27 intf_max_bw = 13107200
28 # Largura de banda (bytes) minima das interfaces = 40% da maxima
29 intf_min_bw = intf_max_bw * 0.4
30 # Erro medio da previsao da rede neural treinada
    #####
    mudar isso
31 nnp_err = 0.018111
32 # Tamanho da janela de entrada da rede neural
33 input_window = 5
34
35 ### Espera 's_time' segundos , porem a espera maxima e' de 's_max' segundos
36 def sleep_max(s_max, s_time):
37     sleep_time = s_max - s_time
38     if sleep_time > 0:
39         time.sleep(sleep_time)
40
41 ### Gera uma lista com numeros aleatorios
42 def rand_list(min_v, max_v, list_len):
43     return [ random.randrange(min_v, max_v) for x in xrange(list_len) ]

```

```

44
45 ### Thread para chamar comandos simultaneos
46 class ConcurrentProcess(Thread):
47     def __init__(self, command):
48         Thread.__init__(self)
49         self.command = command
50         self.output = None
51         self.error = None
52
53     def run(self):
54         sp = subprocess.Popen([self.command], stdout=subprocess.
55             PIPE, shell=True)
56         (self.output, self.error) = sp.communicate()
57
58     def getResult(self):
59         return (self.output, self.error)
60
61 ### Classe para ler as linhas de um arquivo do final para o inicio
62 class BackwardsReader:
63     """Read a file line by line, backwards"""
64     BLKSIZE = 4096
65     def __init__(self, file):
66         self.file = file
67         self.buf = ""
68         try:
69             self.file.seek(-1, 2)
70             self.trailing_newline = 0
71             lastchar = self.file.read(1)
72             if lastchar == "\n":
73                 self.trailing_newline = 1
74                 self.file.seek(-1, 2)
75         except IOError:
76             raise Exception("Erro ao ler as ultimas linhas do
77                 arquivo")
78
79     def readline(self):
80         while 1:
81             newline_pos = string.rfind(self.buf, "\n")
82             pos = self.file.tell()
83             if newline_pos != -1:
84                 # Found a newline
85                 line = self.buf[newline_pos+1:]
86                 self.buf = self.buf[:newline_pos]
87                 if pos != 0 or newline_pos != 0 or self.
88                     trailing_newline:
89                     line += "\n"
90             return line

```

```
88         else :
89             if pos == 0:
90                 # Start-of-file
91                 return ""
92             else :
93                 # Need to fill buffer
94                 toread = min(self.BLKSIZE, pos)
95                 self.file.seek(-toread, 1)
96                 self.buf = self.file.read(toread) +
97                     self.buf
98                 self.file.seek(-toread, 1)
99                 if pos - toread == 0:
100                     self.buf = "\n" + self.buf
```

Código A.14: *Script* de geração do tráfego entre roteador e VLANs

```

1 import os, time, subprocess, random, utils
2 from threading import Thread
3
4 ### Classe que gera o trafego entre os hosts (1, 2 e 3) e o roteador
5 class TrafficGenerator:
6     def __init__(self):
7         # o tempo (segundos) que o trafego gerado deve consumir
8         self.sec = sec
9         # quantidade maxima de downloads simultaneos (threads)
10        self.max_sd = max_sd
11        # tamanho maximo (bytes) que e' baixado nos 'sec' segundos
12        self.max_ds = max_ds
13
14    def generate(self):
15        # largura de banda (maximo)
16        bw = self.max_ds
17        # intervalos por hora / periodo
18        intervals = 3600 / self.sec
19
20        # 3 primeira horas - trafego alto 1
21        high1 = utils.rand_list(3/4.*bw, bw, 3*intervals)
22        # 3 horas - trafego descendente 1
23        down1 = utils.rand_list(3/4.*bw, bw, 3*intervals)
24        down1 = self._get_trend(down1, 7/8.*bw, bw/8., -1)
25        # 3 horas - trafego baixo 1
26        low1 = utils.rand_list(0, bw/4., 3*intervals)
27        # 2 horas - trafego crescente 2
28        up1 = utils.rand_list(0, bw/4., 2*intervals)
29        up1 = self._get_trend(up2, bw/8., 7/8.*bw, 1)
30        # 2 horas - trafego alto 3
31        high2 = utils.rand_list(3/4.*bw, bw, 2*intervals)
32        # 1 hora - trafego descendente 3
33        down2 = utils.rand_list(3/4.*bw, bw, intervals)
34        down2 = self._get_trend(down3, 7/8.*bw, bw/2., -1)
35        # 3 horas - trafego medio 1
36        medium1 = utils.rand_list(3/8.*bw, 5/8.*bw, 3*intervals)
37
38        mnt_file = open('teste.txt', 'w')
39        arquivo = [high1, down1, low1, up1, high2, down2, medium1]
40        try:
41            salva = ''
42            for item in arquivo:
43                for valor in item:
44                    salva = salva + str(valor) + ', '
45                salva = salva + '\n'
46            mnt_file.write(salva)

```

```

47         finally:
48             mnt_file.close()
49
50         ### Transforma um trafego em uma tendencia
51         ## param 't' e 'o trafego (lista)
52         ## param 'ini_v' e 'end_v' valores iniciais e finais da tendencia
53         ## param 'trend_dir' 1 para tendencia crescente e -1 para
54         decrecente
55         ## return lista com o trafego em tendencia
56         def _get_trend(self, t, ini_v, end_v, trend_dir):
57             # distancia entre inicio e fim da tendencia
58             dist = len(t)
59             # valor que sera aplicado na tendencia
60             step_v = abs(ini_v - end_v)/dist
61             # aplicar o 'step_v' incrementalmente para criar uma
62             tendencia linear
63             for i in xrange(dist):
64                 t[i] = t[i] + trend_dir * ((i+1) * step_v)
65             return t
66
67     def traffic_shape():
68         t_file = open('_gen_traffic.csv', 'r')
69         try:
70             lines = [ int(float(x.strip('\n'))) for x in t_file.
71                       readlines() ]
72
73         finally:
74             t_file.close()
75
76         # criar o trafego para os 10 segundos
77         # 100kb = 1, 500kb = 2, 1mb = 3, 10mb = 4, 50mb = 5, 100mb = 6
78         # aproximei os valores para evitar erros
79         dic = {1:102400, 2:512000, 3:1048576, 4:10485760, 5:52428800,
80               6:104857600}
81         new_lines = [] # lista com lista das threads
82         new_lines_id = []
83         # soma 20500000, aproximadamente 20mb no limite, para evitar erros
84         de aproximacao
85         for value in lines:
86             v = value + 20500000
87             threads = []
88             threads_id = []
89             while sum(threads) + dic[1] <= v:
90                 if sum(threads) + dic[6] <= v:
91                     threads.append(dic[6])
92                     threads_id.append(6)
93                 elif sum(threads) + dic[5] <= v:
94                     threads.append(dic[5])
95                     threads_id.append(5)

```

```

89         elif sum(threads) + dic[4] <= v:
90             threads.append(dic[4])
91             threads_id.append(4)
92         elif sum(threads) + dic[3] <= v:
93             threads.append(dic[3])
94             threads_id.append(3)
95         elif sum(threads) + dic[2] <= v:
96             threads.append(dic[2])
97             threads_id.append(2)
98         elif sum(threads) + dic[1] <= v:
99             threads.append(dic[1])
100             threads_id.append(1)
101         break
102     new_lines.append(threads)
103     new_lines_id.append(threads_id)
104     t_file = open('_gen_Threads_sum.csv', 'w')
105     try:
106         texto = ''
107         for item in new_lines:
108             texto = texto + str(sum(item)) + '\n'
109         t_file.write(texto)
110     finally:
111         t_file.close()
112     t_file = open('_gen_Threads.txt', 'w') # cada linha eh 10 segundos
113         de trafego , com as threads (qual arquivo ira baixar)
114     try:
115         texto = ''
116         for item in new_lines_id:
117             linha = [ str(i) for i in item]
118             texto = texto + ','.join(linha) + '\n'
119         t_file.write(texto)
120     finally:
121         t_file.close()
122     def execute_traffic():
123         # 100kb = 1, 500kb = 2, 1mb = 3, 10mb = 4, 50mb = 5, 100mb = 6
124         # aproximei os valores para evitar erros
125         dic = {1:'100kb.dat', 2:'500kb.dat', 3:'1mb.dat', 4:'10mb.dat', 5:'
126             50mb.dat', 6:'100mb.dat'}
127
128         t_file = open('_gen_Threads.txt', 'r')
129         lines = []
130         try:
131             lines = [ x.strip('\n') for x in t_file.readlines() ]
132             lines = [ x.strip('\r') for x in lines ]
133             lines = [ x.split(',') for x in lines ]
134         finally:

```

```

134         t_file.close()
135     # gera as threads
136     i = 0
137     j = 0
138     tl = []
139     for line in lines:
140         timestamp = time.time()
141         for item in line:
142             file_id = int(item)
143             t = Traffic(dic[file_id])
144             t.setName(str(i) + '-' + str(j) + '_' + dic[file_id]
145                      ])
146             tl.append(t)
147             j += 1
148             t.start()
149         print '\n\n',i,'\n\n'
150         # cada linha tem o trafego de 10 segundos
151         for_time = time.time() - timestamp
152         utils.sleep_max(10, for_time)
153         i += 1
154     for t in tl:
155         t.join()
156
157     ### Thread que faz a transferencia do arquivo do servidor
158     class Traffic(Thread):
159         def __init__(self, file_name):
160             Thread.__init__(self)
161             self.file_name = file_name
162
163         def run(self):
164             self.teste()
165
166         def teste(self):
167             #print self.getName()
168             #os.system('wget --no-proxy http://200.19.151.7/' + self.
169                      file_name + ' -O ' + self.getName())
170             os.system('wget --no-proxy --no-verbose http
171                      ://200.19.151.7/' + self.file_name + ' -O ' + self.
172                      getName())
173             os.system('rm -f ' + self.getName())

```

Configuração dos Equipamentos de Hardware

Aqui são apresentados as configurações de todos os equipamentos utilizados para o experimento do algoritmo GPLNEURO. Na Seção B.1, estão as configurações do computador servidor onde foi implementado o roteador. A Seção B.2 mostra as configurações dos três computadores que foram conectados ao roteador. Por último, na Seção B.3, estão apresentadas todas as configurações do *switch*, no qual os computadores e o roteador foram interligados. O *switch* utilizado é do modelo TRENDnet TEG-160WS *Smart Gigabit Switch*, versão B1.

B.1 Configuração do Roteador Implementado em um Servidor

Tabela B.1: Configuração do Roteador.

Característica	Descrição
CPU	Intel(R) Xeon(R) 3040 1.86GHz
RAM	4GB DDR2 533MHz
Sistema Operacional	i386 GNU/Linux 2.6.18 — CentOS release 5.11 (Final)
Interface de Rede <i>eth0</i>	RealTek RTL8139 100Mbps
Interface de Rede <i>eth1</i>	RealTek RTL8139 100Mbps
Interface de Rede <i>eth2</i>	RealTek RTL8139 100Mbps
Interface de Rede <i>eth3</i>	RealTek RTL8139 100Mbps

B.2 Configuração dos Computadores Conectados ao Roteador

Tabela B.2: Configuração do Computador 1.

Característica	Descrição
CPU	Intel(R) Core(TM) 2 E8400 3.00GHz
RAM	2GB DDR2 400MHz
Sistema Operacional	i386 GNU/Linux 3.13.0 — Xubuntu 14.04 LTS
Interface de Rede	Intel 82557 Ethernet Pro 100Mbps

Tabela B.3: Configuração do Computador 2.

Característica	Descrição
CPU	Intel(R) Core(TM) 2 Quad Q8200 2.33GHz
RAM	8GB DDR2 400MHz
Sistema Operacional	i386 GNU/Linux 3.13.0 — Xubuntu 14.04 LTS
Interface de Rede eth3	Realtek RTL8411 100Mbps

Tabela B.4: Configuração do Computador 3.

Característica	Descrição
CPU	Intel(R) Core(TM) 2 Quad Q6600 2.40GHz
RAM	4GB DDR2 667MHz
Sistema Operacional	i386 GNU/Linux 3.13.0 — Xubuntu 14.04 LTS
Interface de Rede eth3	NVIDIA MCP73 100Mbps

B.3 Configuração do *Switch* TRENDnet Usado para Criar as VLANs

Tabela B.5: Switch Status

Product Name	16G Web-SMART Switch
Firmware Version	2.00.06
Protocol Version	2.001.001
IP Address	192.168.0.1
Subnet mask	255.255.255.0
Default gateway	192.168.0.254
Trap IP	0.0.0.0
MAC address	00-40-F4-D1-FF-40
System Name	Switch Tiago
Location Name	lab redes
Login Timeout (minutes)	5
System UpTime	6 days 21 hours 26 mins 34 seconds

Tabela B.6: **PORT Status 10/100/1000 Mbps**

ID	Speed		Flow Control		QoS
	Set	Status	Set	Status	
01	Auto	100M Full	Enable	On	Normal
02	Auto	Down	Enable	On	Normal
03	Auto	1G Full	Enable	On	Normal
04	Auto	Down	Enable	On	Normal
05	Auto	100M Full	Enable	On	Normal
06	Auto	Down	Enable	On	Normal
07	Auto	1G Full	Enable	On	Normal
08	Auto	Down	Enable	On	Normal
09	Auto	1G Full	Enable	On	Normal
10	Auto	Down	Enable	On	Normal
11	Auto	100M Full	Enable	On	Normal
12	Auto	Down	Enable	On	Normal
13	Auto	Down	Enable	On	Normal
14	Auto	Down	Enable	On	Normal
15	Auto	Down	Enable	On	Normal
16	Auto	Down	Enable	On	Normal

Tabela B.7: **Port-based VLAN Settings**

ID	Description	Member
01	VLAN 1	01 02 03 04
02	VLAN 2	13 14 15 16
03	VLAN 3	09 10 11 12
04	VLAN 4	05 06 07 08

Tabela B.8: **TRUNK Status**

ID	Member
01	Disable
02	Disable

Tabela B.9: **Mirror Status**

Sniffer Mode	Disable
Sniffer Port	
Source Port	