UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Aprendizado Supervisionado Usando Redes Neurais Construtivas

João Roberto Bertini Junior

São Carlos
Maio/2006

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

B544as

Bertini Junior, João Roberto.
Aprendizado supervisionado usando redes neurais construtivas / João Roberto Bertini Junior. -- São Carlos : UFSCar, 2006.
208 p.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2006.

1. Redes neurais (Computação). 2. Inteligência artificial. 3. Aprendizado do computador. 4. Reconhecimento de padrões. I. Título.

CDD: 006.32 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"Aprendizado Supervisionado Usando Redes Neurais Construtivas"

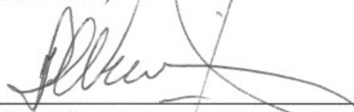
JOÃO ROBERTO BERTINI JUNIOR

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

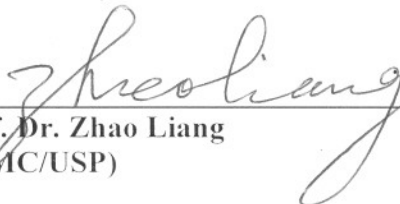
Membros da Banca:

Profa. Dra. Maria do Carmo Nicoletti
(Orientadora – DC/UFSCar)

Profa. Dra. Heloisa de Arruda Camargo
(DC/UFSCar)



Prof. Dr. Haroldo Fraga de Campos Velho
(LAC/INPE)



Prof. Dr. Zhao Liang
(ICMC/USP)

São Carlos
Maio/2006

Agradecimentos

Aos meus pais, João Roberto Bertini e Luzia Monzani Bertini pelo carinho, pelo incentivo, pela dedicação durante toda minha vida escolar e pelo apoio financeiro durante todos esses anos.

A Profa. Dr. Maria do Carmo pela amizade, por todo o conhecimento que me passou, pelos tantos conselhos dados ao longo desses dois anos, pela preocupação e principalmente pela enorme dedicação.

A Capes pelo apoio financeiro, que tornou esses dois anos mais fáceis.

"*Computer science is about computers*
the way astronomy is about telescopes"
- Edsger Dijkstra -

Resumo

Aprendizado neural construtivo é um modelo de aprendizado neural que não pressupõe a definição de uma topologia de rede fixada antes do início do treinamento. A principal característica deste modelo de aprendizado é a construção dinâmica das camadas intermediárias da rede, à medida que vão sendo necessárias ao seu treinamento.

Este trabalho investiga três frentes de pesquisas com relação ao aprendizado neural construtivo, a saber, algoritmos para o treinamento de TLUs, algoritmos neurais construtivos para problemas que envolvem duas classes e algoritmos neurais construtivos para o tratamento de problemas multiclasses.

Com relação à primeira frente de pesquisa os algoritmos discutidos para o treinamento de TLUs são o Perceptron, o Pocket, o PMR, o Thermal, o Thermal Modificado, o MinOver e o BPC.

Na frente de pesquisa relativa ao aprendizado neural construtivo para duas classes são revistos os algoritmos Tower, Pyramid, Tiling e Upstart, para que as versões multiclasses desses algoritmos possam ser tratadas. São investigados os algoritmos neurais construtivos Shift, Offset, PTI, Perceptron Cascade e Sequential e propostos dois algoritmos híbridos: o Tiling Híbrido, que não restringe o treinamento de TLUs a um único algoritmo e o OffTiling que agrega os algoritmos Tiling e Offset.

A frente que focaliza o aprendizado neural construtivo multiclasse investiga os algoritmos para o treinamento de TLUs quando o problema envolvido apresentar mais que duas classes bem como apresenta e discute as versões multiclasses dos algoritmos Tower, Pyramid, Tiling, Upstart e Perceptron Cascade.

O trabalho descreve uma avaliação empírica dos algoritmos investigados, em vários domínios de conhecimento bem como discute e analisa os resultados obtidos.

Abstract

Constructive neural learning is a neural learning model that does not assume a fixed network topology before training begins. The main characteristic of this learning model is the dynamic construction of the network's hidden layers that occurs simultaneously with training.

This work investigates three topics related to constructive neural learning namely algorithms for training an individual TLU, constructive neural algorithms for two class problems and constructive neural algorithms for multiclass problems.

The first research topic is approached by discussing a few TLU training algorithms, namely Perceptron, Pocket, Thermal, Modified Thermal, MinOver and BCP.

This work approaches constructive neural learning for two class classification tasks by initially reviewing Tower, Pyramid, Tiling and Upstart algorithms, aiming at their multiclass versions. Next five constructive neural algorithms namely Shift, Offset, PTI, Perceptron Cascade and Sequential are investigated and two hybrid algorithms are proposed: Hybrid Tiling, that does not restrict the TLU's training to only one algorithm and the OffTiling, a collaborative approach based on Tiling and Offset.

Multiclass constructive neural learning was approached by investigating TLUs training algorithms that deal with multiclass as well as by investigating multiclass versions of Tower, Pyramid, Tiling, Upstart and Perceptron Cascade.

This research work also describes an empirical evaluation of all the investigated algorithms conducted using several knowledge domains. Results are discussed and analyzed.

Lista de Figuras

Figura 2.1 – O Perceptron e algumas de suas variantes.. 5
Figura 2.2 – O Perceptron: recebe como entrada valores reais, calcula o produto interno dos valores recebidos e então produz uma saída 1 se o resultado da combinação for maior do que zero e valor −1, caso contrário............ 7
Figura 2.3 – Representação do Perceptron usando o termo *bias* como uma entrada adicional do conjunto de treinamento................................ 8
Figura 2.4 – Superfície de decisão representada por duas entradas (a) Representação de um conjunto linearmente separável (b) Conjunto não linearmente separável – não existe uma reta, no espaço bidimensional em questão, que separe as instâncias (+) das instâncias (–)....................................... 9
Figura 2.5 – Exemplos relativos ao termo *bias* - (a) hiperplano obtido sem o uso de *bias*; (b) hiperplano obtido com uso do *bias*...................... 10
Figura 3.1 – Exemplo de atualização de W por meio do cálculo dos baricentros dos exemplos classificados incorretamente.. 30
Figura 3.2 – Representação da relação entre o valor do *bias* e suas classificações...... 35
Figura 3.3 – Primeira iteração do algoritmo BCPMin sobre o conjunto da Tabela 3.1.. 42
Figura 3.4 – Obtendo os novos baricentros b1’ e b2’.. 43
Figura 3.5 – Cálculo do vetor W’, segunda iteração do BCPMin............................... 44
Figura 3.6 – Terceira iteração do BCPMin, separando linearmente o conjunto.......... 45
Figura 3.7 – Hiperplano ótimo encontrado pelo BCPMin.. 47
Figura 3.8 – Hiperplano obtido pelo algoritmo BCPMax... 52
Figura 4.1 – Possível arquitetura de uma rede Tower.. 54
Figura 4.2 – Possível arquitetura de rede Pyramid.. 57
Figura 4.3 – Possível arquitetura de uma rede Tiling.. 59
Figura 4.4 – Uma possível arquitetura de uma rede Upstart....................................... 63
Figura 4.5 – O neurônio u_n classifica corretamente os exemplos de treinamento: E^2, E^3, E^4 e E^7 Comete erro de positivo em E^5 e E^8 e erro de negativo em E^1, E^6 e E^9.. 64
Figura 5.1 – Adicionando o neurônio u_2 (em branco) a uma rede Shift. Os demais neurônios (em cinza) foram adicionados em fases anteriores. As conexões tracejadas farão parte da rede após a inclusão de u_2 70
Figura 5.2 – Conceito de Paridade-3 aprendido por uma rede Shift que usa o PMR para treinar TLUs... 75
Figura 5.3 – Rede Shift que representa o conceito de Paridade-3, na qual as TLUs foram treinadas com o BCPMin... 77
Figura 5.4 – Conceito Paridade-3 aprendido por uma rede Offset. Neurônios indexados com p estão na primeira camada e os indexados com s, estão na segunda camada... 84
Figura 5.5 – Conceito Paridade-3 aprendido por uma rede Offset após fase da poda (comparar com a Figura 5.4)... 85

Figura 5.6 –	Rede gerada pelo algoritmo OffTiling. Neurônios indexados com t estão na camada Tiling, os indexados com p e s estão na primeira e segunda camada, criadas pelo algoritmo Offset, respectivamente..........	89
Figura 5.7 –	Rede PTI que representa o conceito de Paridade-3.................................	97
Figura 5.8 –	Conceito Paridade-3 aprendido por uma rede PTI com o algoritmo BCPMin..	99
Figura 5.9 –	Conceito Paridade-3 representado por uma rede Perceptron Cascade......	105
Figura 5.10 –	Rede Sequential com o BCPMax representando o problema Paridade-3..	111
Figura 5.11 –	Rede Sequential com o IncLp representando o conceito Paridade-3........	117
Figura 6.1 –	Classificação multiclasse utilizando o PMR em abordagem individual...	122
Figura 6.2 –	Classificação multiclasse utilizando o BCP em abordagem individual....	123
Figura 6.3 –	Classificação multiclasse feita pelo PMRWTA......................................	127
Figura 6.4 –	Classificação multiclasse feita pelo algoritmo BCPWTA.......................	131
Figura 7.1 –	Representações gráficas para algoritmos multiclasses. (a) inclusão de um grupo de neurônios; (b) duas camadas com neurônios totalmente conectados; (c) nova representação da situação mostrada em (b)...........	132
Figura 7.2 –	Exemplo de arquitetura de uma rede MTower. O neurônio notado por u_{ij} é o j-ésimo neurônio da i-ésima camada intermediária.......................	133
Figura 7.3 –	Arquitetura geral de uma rede MPyramid...	136
Figura 7.4 –	Arquitetura geral de uma rede MTiling..	140
Figura 7.5 –	Arquitetura geral de uma rede MUpstart..	144
Figura 7.6 –	Possível arquitetura de uma rede MPerceptron Cascade.........................	149

Lista de Tabelas

Tabela 2.1 – Conjunto de treinamento que representa da função *and*, com o *bias* incluído.......... 11
Tabela 2.2 – Execução passo a passo do Perceptron para o conjunto de treinamento da Tabela 2.1.......... 12
Tabela 2.3 – Conjunto de treinamento que representa a o problema paridade-3, com o temo *bias* incluído.......... 14
Tabela 2.4 – Exemplo de execução, passo a passo, do algoritmo Pocket utilizando como conjunto de treinamento os exemplos da Tabela 2.3.......... 14
Tabela 2.5 – Exemplo de execução passo a passo do algoritmo PMR, para o conjunto de treinamento da Tabela 2.3.......... 17
Tabela 2.6 – Exemplo de execução passo a passo do algoritmo Thermal Perceptron.......... 20
Tabela 2.7 – Exemplo de execução do Algoritmo Thermal modificado com o conjunto de treinamento da Tabela 2.3.......... 23
Tabela 2.8 – Exemplo de execução do algoritmo MinOver usando o conjunto de treinamento dado na Tabela 2.3.......... 26
Tabela 3.1 – Conjunto Linearmente Separável com entradas reais.......... 40
Tabela 3.2 – Conjuntos V1 e V2 obtidos na primeira iteração do BCPMin. O mínimo e o máximo de cada conjunto estão assinalados com '*' e '**' respectivamente.......... 41
Tabela 3.3 – Conjunto não linearmente separável com instâncias reais.......... 46
Tabela 5.1 – Conjunto que representa o problema Paridade-3 e os valores de v^k (k = 1,...,8) da rede atual.......... 73
Tabela 5.2 – Conjunto de treinamento do neurônio alocado para corrigir erros de positivo.......... 73
Tabela 5.3 – Conjunto com as classes originais saída de O_1 e novos valores de v^k (k = 1,..., 8).......... 74
Tabela 5.4 – Conjunto de treinamento do neurônio alocado para corrigir erros de negativo.......... 74
Tabela 5.5 – Conjunto de treinamento paridade-3 com os valores de v^k (k = 1,..., 8) para a rede atual.......... 76
Tabela 5.6 – Conjunto com as classes originais, saídas de u_1 e novos valores de v^k (k = 1,..., 8).......... 76
Tabela 5.7 – Representação do problema paridade-3 mais a saída do primeiro neurônio e a função de distância aplicada entre C e O_1.......... 82
Tabela 5.8 – Conjunto de treinamento do segundo neurônio da primeira camada juntamente com a saída do neurônio e a função de distância.... 83
Tabela 5.9 – Conjunto de treinamento do terceiro neurônio da primeira camada mais a saída deste.......... 83
Tabela 5.10 – Saída dos neurônios da primeira camada para toda instância do conjunto de treinamento.......... 84
Tabela 5.11 – Matriz de representações internas sem repetições.......... 85

Tabela 5.12 – Conjunto de treinamento com vinte instâncias descritas por valores reais.. 87
Tabela 5.13 – Matriz de representação interna gerada pela camada Tiling.................. 88
Tabela 5.14 – Conjunto de treinamento que representa o problema Paridade-3 com a adição do termo *bias*.. 94
Tabela 5.15 – Conjunto de treinamento com a saída do neurônio mestre mais as classes atualizadas para o próximo conjunto............................ 95
Tabela 5.16 – Atual matriz de representação interna da primeira camada.................. 95
Tabela 5.17 – Conjunto de treinamento do segundo neurônio auxiliar u_{13} da primeira camada... 96
Tabela 5.18 – Matriz de representação interna da primeira camada............................ 96
Tabela 5.19 – Conjunto com as instâncias de treinamento com as classes originais e as classes para treinar o próximo neurônio......................... 97
Tabela 5.20 – Conjunto de treinamento para o segundo neurônio auxiliar da primeira camada.. 98
Tabela 5.21 – Matriz de representação interna da primeira camada............................ 98
Tabela 5.22 – Conjunto de treinamento para que o primeiro neurônio corrija erros de positivo... 103
Tabela 5.23 – Conjunto de treinamento original com adição de uma entrada referente ao neurônio adicionado.. 103
Tabela 5.24 – Conjunto para o treinamento do neurônio u_2.. 104
Tabela 5.25 – Conjunto original com a adição das entradas referentes aos dois neurônios adicionados... 104
Tabela 5.26 – Conjunto de treinamento para o neurônio alocado para corrigir erro de positivo... 105
Tabela 5.27 – Conjunto para retreinar o neurônio de saída após a inserção do primeiro neurônio intermediário.. 106
Tabela 5.28 – Conjunto de exemplos que representa o problema Paridade-3.............. 110
Tabela 5.29 – Matriz de representação interna do exemplo de execução do Sequential com o BCPMax... 110
Tabela 5.30 – Conjunto de exemplos que representa o problema Paridade-3 com a adição do termo *bias*.. 115
Tabela 5.31 – Matriz de representação interna do exemplo de execução do Sequential com o IncLp.. 116
Tabela 5.32 – Matriz de representação interna do exemplo de execução do Sequential com o IncLp corrigida... 116
Tabela 6.1 – Conjunto de treinamento com três classes e vinte e quatro instâncias.... 121
Tabela 6.2 – Vetores de pesos obtidos da execução do algoritmo PMRWTA para o conjunto de treinamento da Tabela 6.1..................................... 126
Tabela 6.3 – Conjunto de vetores de pesos e *bias* obtidos na execução do algoritmo BCPWTA.. 131
Tabela 7.1 – Exemplo de neurônio morto.. 142
Tabela 7.2 – Exemplo de neurônios correlacionados.. 143
Tabela 7.3 – Exemplo de neurônio redundante... 143
Tabela 8.1 – Especificação dos domínios de conhecimento....................................... 155
Tabela 8.2 – PMR, BCPMin, MinOver e Thermal Modificado / Paridade-5............. 155
Tabela 8.3 – Tower / Paridade-5.. 156
Tabela 8.4 – Pyramid / Paridade-5.. 156
Tabela 8.5 – Upstart / Paridade-5... 156
Tabela 8.6 – Shift / Paridade-5... 156

Tabela 8.7 – Perceptron Cascade / Paridade-5 ... 157
Tabela 8.8 – Tiling / Paridade-5 ... 157
Tabela 8.9 – PTI / Paridade-5 ... 157
Tabela 8.10 – OffTiling / Paridade-5 ... 157
Tabela 8.11 – Sequential / Paridade-5 ... 158
Tabela 8.12 – PMR, BCPMin, MinOver e Thermal Modificado / Vestibular ... 159
Tabela 8.13 – Tower / Vestibular ... 159
Tabela 8.14 – Pyramid / Vestibular ... 160
Tabela 8.15 – Upstart / Vestibular ... 160
Tabela 8.16 – Shift / Vestibular ... 160
Tabela 8.17 – Perceptron Cascade / Vestibular ... 160
Tabela 8.18 – Tiling / Vestibular ... 161
Tabela 8.19 – PTI / Vestibular ... 161
Tabela 8.20 – OffTiling / Vestibular ... 161
Tabela 8.21 – Sequential / Vestibular ... 161
Tabela 8.22 – PMR, BCPMin, MinOver e Thermal Modificado / Ionosphere ... 162
Tabela 8.23 – Tower / Ionosphere ... 163
Tabela 8.24 – Pyramid / Ionosphere ... 163
Tabela 8.25 – Upstart / Ionosphere ... 163
Tabela 8.26 – Shift / Ionosphere ... 163
Tabela 8.27 – Perceptron Cascade / Ionosphere ... 164
Tabela 8.28 – Tiling / Ionosphere ... 164
Tabela 8.29 – PTI / Ionosphere ... 164
Tabela 8.30 – OffTiling / Ionosphere ... 165
Tabela 8.31 – Sequential / Ionosphere ... 165
Tabela 8.32 – PMR, BCPMin, MinOver e Thermal Modificado / Liver ... 166
Tabela 8.33 – Tower / Liver ... 166
Tabela 8.34 – Pyramid / Liver ... 166
Tabela 8.35 – Upstart / Liver ... 167
Tabela 8.36 – Shift / Liver ... 167
Tabela 8.37 – Perceptron Cascade / Liver ... 167
Tabela 8.38 – Tiling / Liver ... 167
Tabela 8.39 – PTI / Liver ... 168
Tabela 8.40 – OffTiling / Liver ... 168
Tabela 8.41 – Sequential / Liver ... 168
Tabela 8.42 – PMR, BCPMin, MinOver e Thermal Modificado / Pima ... 169
Tabela 8.43 – Tower / Pima ... 169
Tabela 8.44 – Pyramid / Pima ... 169
Tabela 8.45 – Upstart / Pima ... 170
Tabela 8.46 – Shift / Pima ... 170
Tabela 8.47 – Perceptron Cascade / Pima ... 170
Tabela 8.48 – Tiling / Pima ... 170
Tabela 8.49 – PTI / Pima ... 171
Tabela 8.50 – OffTiling / Pima ... 171
Tabela 8.51 – Sequential / Pima ... 171
Tabela 8.52 – PMR, BCPMin, MinOver e Thermal Modificado / Breast Cancer ... 172
Tabela 8.53 – Tower / Breast Cancer ... 172
Tabela 8.54 – Pyramid / Breast Cancer ... 173
Tabela 8.55 – Upstart / Breast Cancer ... 173
Tabela 8.56 – Shift / Breast Cancer ... 173

Tabela 8.57 – Perceptron Cascade / Breast Cancer.... 173
Tabela 8.58 – Tiling / Breast Cancer.... 174
Tabela 8.59 – PTI / Breast Cancer.... 174
Tabela 8.60 – OffTiling / Breast Cancer.... 174
Tabela 8.61 – Sequential / Breast Cancer.... 174
Tabela 8.62 – PMR, BCPMin, MinOver e Thermal Modificado / Monks1.... 176
Tabela 8.63 – Tower / Monks1…. 176
Tabela 8.64 – Pyramid / Monks1.... 176
Tabela 8.65 – Upstart / Monks1.... 177
Tabela 8.66 – Shift / Monks1.... 177
Tabela 8.67 – Perceptron Cascade / Monks1.... 177
Tabela 8.68 – Tiling / Monks1.... 177
Tabela 8.69 – PTI / Monks1.... 178
Tabela 8.70 – OffTiling / Monks1.... 178
Tabela 8.71 – Sequential / Monks1.... 178
Tabela 8.72 – PMR, BCPMin, MinOver e Thermal Modificado / Monks2.... 179
Tabela 8.73 – Tower / Monks2.... 179
Tabela 8.74 – Pyramid / Monks2.... 179
Tabela 8.75 – Upstart / Monks2.... 180
Tabela 8.76 – Shift / Monks2.... 180
Tabela 8.77 – Perceptron Cascade / Monks2.... 180
Tabela 8.78 – Tiling / Monks2.... 180
Tabela 8.79 – PTI / Monks2.... 181
Tabela 8.80 – OffTiling / Monks2.... 181
Tabela 8.81 – Sequential / Monks2.... 181
Tabela 8.82 – PMR, BCPMin, MinOver e Thermal Modificado / Monks3.... 182
Tabela 8.83 – Tower / Monks3.... 182
Tabela 8.84 – Pyramid / Monks3.... 182
Tabela 8.85 – Upstart / Monks3.... 183
Tabela 8.86 – Shift / Monks3.... 183
Tabela 8.87 – Perceptron Cascade / Monks3.... 183
Tabela 8.88 – Tiling / Monks3.... 183
Tabela 8.89 – PTI / Monks3.... 184
Tabela 8.90 – OffTiling / Monks3.... 184
Tabela 8.91 – Sequential / Monks3.... 184
Tabela 8.92 – Especificação dos domínios multiclasses.... 185
Tabela 8.93 – PMRWTA, BCPWTA, PMR Individual, BCP Individual / Íris.... 185
Tabela 8.94 – MTower / Íris.... 186
Tabela 8.95 – MPyramid / Íris.... 186
Tabela 8.96 – MUpstart / Íris.... 186
Tabela 8.97 – MPerceptron Cascade / Íris.... 186
Tabela 8.98 – MTiling / Íris.... 186
Tabela 8.99 – PMRWTA, BCPWTA, PMR Individual, BCP Individual / E.Coli.... 187
Tabela 8.100 – MTower / E.Coli.... 188
Tabela 8.101 – MPyramid / E.Coli.... 188
Tabela 8.102 – MUpstart / E.Coli.... 188
Tabela 8.103 – MPerceptron Cascade / E.Coli.... 188
Tabela 8.104 – MTiling / E.Coli.... 188
Tabela 8.105 – PMRWTA, BCPWTA, PMR Individual, BCP Individual / Balance.... 189
Tabela 8.106 – MTower / Balance.... 190

Tabela 8.107 – MPyramid / Balance........ 190
Tabela 8.108 – MUpstart / Balance........ 190
Tabela 8.109 – MPerceptron Cascade / Balance........ 190
Tabela 8.110 – MTiling / Balance........ 190
Tabela 8.111 – PMRWTA, BCPWTA, PMR Individual, BCP Individual / Contraceptive........ 191
Tabela 8.112 – MTower / Contraceptive........ 191
Tabela 8.113 – MPyramid / Contraceptive........ 192
Tabela 8.114 – MUpstart / Contraceptive........ 192
Tabela 8.115 – MPerceptron Cascade / Contraceptive........ 192
Tabela 8.116 – MTiling / Contraceptive........ 192
Tabela 8.117 – PMRWTA, BCPWTA, PMR Individual, BCP Individual / Glass........ 193
Tabela 8.118 – MTower / Glass........ 193
Tabela 8.119 – MPyramid / Glass........ 193
Tabela 8.120 – MUpstart / Glass........ 194
Tabela 8.121 – MPerceptron Cascade / Glass........ 194
Tabela 8.122 – MTiling / Glass........ 194

Lista de Algoritmos

Algoritmo 2.1 – Pseudocódigo do algoritmo Perceptron.. 11
Algoritmo 2.2 – Pseudocódigo do algoritmo Pocket.. 13
Algoritmo 2.3 – Pseudocódigo do algoritmo Pocket com Modificação Ratchet (PMR).. 16
Algoritmo 2.4 – Pseudocódigo do algoritmo Thermal Perceptron.............................. 20
Algoritmo 2.5 – Pseudocódigo do algoritmo Thermal Modificado............................ 22
Algoritmo 2.6 – Pseudocódigo do algoritmo MinOver.. 25
Algoritmo 3.1 – Pseudocódigo do algoritmo BCPMin.. 37
Algoritmo 3.2 – Pseudocódigo do algoritmo BCPMin (Continuação)........................ 38
Algoritmo 3.3 – Pseudocódigo do algoritmo BCPMin (Continuação)........................ 39
Algoritmo 3.4 – Pseudocódigo do algoritmo BCPMax.. 49
Algoritmo 3.5 – Pseudocódigo do algoritmo BCPMax (Continuação)....................... 50
Algoritmo 3.6 – Pseudocódigo do algoritmo BCPMax (Continuação)....................... 51
Algoritmo 4.1 – Pseudocódigo do algoritmo Tower... 56
Algoritmo 4.2 – Pseudocódigo do algoritmo Pyramid.. 58
Algoritmo 4.3 – Pseudocódigo do algoritmo Tiling.. 61
Algoritmo 4.4 – Pseudocódigo do algoritmo Upstart.. 67
Algoritmo 5.1 – Pseudocódigo do algoritmo Shift.. 72
Algoritmo 5.2 – Pseudocódigo do algoritmo Offset.. 81
Algoritmo 5.3 – Pseudocódigo do algoritmo híbrido OffTiling.................................. 87
Algoritmo 5.4 – Pseudocódigo do algoritmo PTI.. 93
Algoritmo 5.5 – Pseudocódigo do algoritmo Perceptron Cascade.............................. 102
Algoritmo 5.6 – Pseudocódigo do algoritmo Sequential usando o BCPMax.............. 109
Algoritmo 5.7 – Pseudocódigo do algoritmo IncLp.. 112
Algoritmo 5.8 – Pseudocódigo do algoritmo Sequential usando o IncLp................... 114
Algoritmo 6.1 – Pseudocódigo para o treinamento individual de m TLUs.................. 120
Algoritmo 6.2 – Pseudocódigo do algoritmo PMRWTA.. 125
Algoritmo 6.3 – Pseudocódigo do algoritmo BCPWTA... 130
Algoritmo 7.1 – Pseudocódigo do algoritmo MTower... 135
Algoritmo 7.2 – Pseudocódigo do algoritmo MPyramid.. 137
Algoritmo 7.3 – Pseudocódigo do algoritmo MTiling.. 141
Algoritmo 7.4 – Pseudocódigo do algoritmo MUpstart.. 148
Algoritmo 7.5 – Pseudocódigo do algoritmo MPerceptron Cascade.......................... 152

Sumário

Resumo i

Abstract ii

Lista de Figuras iii

Lista de Tabelas v

Lista de Algoritmos x

Sumário xi

Capítulo 1 1

Introdução

Capítulo 2 5

O Perceptron e suas Variantes Pocket, PMR, Thermal, Thermal Modificado e MinOver

2.1 Introdução.. 5

2.2 O Perceptron... 6

2.2.1 O Termo Bias.. 9

2.2.2 O Algoritmo Perceptron.. 10

2.2.3 Exemplo de Execução do Algoritmo Perceptron.. 11

2.3 O Algoritmo Pocket.. 12

2.3.1 Exemplo de Execução do Algoritmo Pocket... 14

2.4 O Algoritmo Pocket com Modificação Ratchet (PMR)...................................... 15

2.4.1 Exemplo de Execução do PMR.. 17

2.5 O Algoritmo Thermal Perceptron.. 18

2.5.1 Exemplo de Execução do Algoritmo Thermal Perceptron........................... 20

2.5.2 O Algoritmo Thermal Perceptron Modificado... 21

2.5.3 Exemplo de Execução do Algoritmo Thermal Modificado.......................... 22

2.6 O Algoritmo MinOver... 23

2.6.1 Exemplo de Execução do Algoritmo MinOver.. 25

Capítulo 3 **27**

O Algoritmo BCP – Uma Abordagem Geométrica para o Treinamento de TLUs

3.1 Introdução... 27

3.2 A Proposta Geral do Algoritmo BCP.. 28

3.3 Considerações Preliminares sobre as Versões BCPMin e BCPMax................... 32

3.4 BCPMin – Minimizando o Número de Instâncias Classificadas Incorretamente... 34

3.4.1 Aprendizado usando o BCPMin em um Conjunto de Treinamento Linearmente Separável.. 40

3.4.2 Aprendizado usando o BCPMin em um Conjunto de Treinamento Não Linearmente Separável... 46

3.5 BCPMax – Maximizando o Número de Instâncias Excluídas............................. 47

3.6 Aprendizado usando o BCPMax em um Conjunto Não Linearmente Separável... 51

Capítulo 4 **53**

Aprendizado Neural Construtivo – Revisão dos Algoritmos Tower, Pyramid, Tiling e Upstart

4.1 Introdução... 53

4.2 O Algoritmo Tower.. 54

4.3 O Algoritmo Pyramid.. 56

4.4 O Algoritmo Tiling.. 59

4.4.1 Utilizando o PMR e o BCP na Construção de uma Rede Tiling – O Algoritmo Tiling Híbrido... 62

4.5 O Algoritmo Upstart.. 63

Capítulo 5 **68**

Aprendizado Neural Construtivo – Algoritmos Shift, Offset, PTI, Perceptron Cascade e Sequential

5.1 Introdução... 68

5.2 O Algoritmo Shift.. 68

5.2.1 Exemplo de Execução do Algoritmo Shift usando o Algoritmo PMR........ 73

5.2.2 Exemplo de Execução do Algoritmo Shift usando o Algoritmo BCPMin.. 75

5.3 O Algoritmo Offset.. 77

5.3.1 Fase I - Construção da Primeira Camada de uma Rede Offset................... 78

5.3.2 Fase I - Construção da Segunda Camada de uma Rede Offset................... 79

5.3.3 Fase II – A Poda em uma Rede Offset.. 79

5.3.4 Considerações sobre o Pseudocódigo do Algoritmo Offset........................ 80

5.3.5 Exemplo de Execução do Algoritmo Offset usando o Algoritmo PMR...... 82

5.3.6 Uma Alternativa para o Tratamento de Problemas com Atributos Reais para uma Rede Offset – A Proposta Híbrida OffTiling............................. 85

5.3.7 Exemplo de Execução do Algoritmo OffTiling usando o Algoritmo BCPMin.. 87

5.4 O Algoritmo *Partial Target Inversion* (PTI) – Uma versão do Algoritmo Tiling.. 90

5.4.1 Exemplo de Execução do PTI Utilizando o PMR como Algoritmo para o Treinamento de TLUs.. 94

5.4.2 Exemplo de Execução do Algoritmo PTI Utilizando o BCPMin como Algoritmo para o Treinamento de TLUs... 97

5.5 O Algoritmo Perceptron Cascade.. 99

5.5.1 Exemplo de Execução do Algoritmo Perceptron Cascade Usando o PMR.. 103

5.5.2 Exemplo de Execução do Algoritmo Perceptron Cascade Usando o BCPMin.. 105

5.6 O Algoritmo Sequential.. 105

5.6.1 O Algoritmo Sequential com o BCPMax... 108

5.6.1.1 Exemplo de Execução do Sequential em um Conjunto Não Linearmente Separável, usando o BCPMax........................... 109

5.6.2 O Algoritmo *Incremental Linear Programing* (IncLp)............................. 111

5.6.3 O Algoritmo Sequential com o IncLP.. 113

5.6.3.1 Exemplo de Execução do Sequential em um Conjunto Não Linearmente Separável usando o IncLp................................ 115

Capítulo 6 **118**

Aprendizado em Domínios Multiclasses – Versões Estendidas dos Algoritmos PMR e BCP

6.1 Introdução.. 118
6.2 As Versões Multiclasses dos Algoritmos PMR e BCP – Abordagem Individual.. 119
6.2.1 Exemplo do PMR Multiclasse – Abordagem Individual............................ 120
6.2.2 Exemplo do BCP Multiclasse – Abordagem Individual............................. 122
6.3 A Versão Multiclasse do Algoritmo PMR – Abordagem WTA.......................... 123
6.3.1 Exemplo do PMR Multiclasse – Abordagem WTA.................................. 126
6.4 A Versão Multiclasse do Algoritmo BCP – Abordagem WTA........................... 127
6.4.1 Exemplo do BCP Multiclasse – Abordagem WTA................................... 130

Capítulo 7 **132**

Aprendizado Neural Construtivo Multiclasse – Versões Estendidas dos Algoritmos Tower, Pyramid, Tiling, Upstart e Perceptron Cascade

7.1 Introdução.. 132
7.2 O Algoritmo Multiclasse MTower.. 133
7.3 O Algoritmo Multiclasse MPyramid.. 135
7.4 O Algoritmo Multiclasse MTiling... 138
7.4.1 Estratégias de Poda para o Algoritmo MTiling... 141
7.4.1.1 Poda de Neurônios Mortos... 142
7.4.1.2 Poda de Neurônios Correlacionados... 142
7.4.1.3 Poda de Neurônios Redundantes... 143
7.5 O Algoritmo Multiclasse MUpstart.. 144
7.6 O Algoritmo Multiclasse MPerceptron Cascade.. 149

Capítulo 8 **153**

Uma Avaliação Empírica do Aprendizado Neural Construtivo em Diferentes Domínios de Conhecimento

8.1 Introdução.......... 153
8.2 Avaliações de Algoritmos para Treinamento de TLUs e de Algoritmos Neurais Construtivos em Nove Domínios com Duas Classes.......... 154
8.2.1 O Domínio Paridade-5.......... 155
8.2.2 O Domínio Sistema Vestibular (SV).......... 158
8.2.3 O Domínio Ionosphere.......... 162
8.2.4 O Domínio Liver.......... 165
8.2.5 O Domínio Pima.......... 168
8.2.6 O Domínio Breast Cancer.......... 171
8.2.7 Os Domínios Monks.......... 174
8.2.7.1 O Domínio Monks1.......... 175
8.2.7.2 O Domínio Monks2.......... 178
8.2.7.3 O Domínio Monks3.......... 181
8.3 Avaliação de Algoritmos Neurais Construtivos Multiclasses.......... 184
8.3.1 O Domínio Íris.......... 185
8.3.2 O Domínio E.Coli.......... 187
8.3.3 O Domínio Balance.......... 189
8.3.4 O Domínio Contraceptive.......... 191
8.3.5 O Domínio Glass.......... 192

Capítulo 9 **195**

Conclusões

9.1 Conclusões sobre Algoritmos de Treinamento de TLUs.......... 195
9.2 Conclusões sobre Algoritmos Neurais Construtivos.......... 196
9.3 Conclusões sobre Algoritmos de Treinamento Multiclasses.......... 199
9.4 Trabalhos Futuros.......... 201

Referências Bibliográficas **202**

1 capítulo

Introdução

Dentre os vários modelos de Aprendizado de Máquina (AM) tratados como uma área de pesquisa da Inteligência Artificial (IA), o chamado *aprendizado indutivo* é o que tem sido mais amplamente pesquisado; esse fato pode ser constatado nas inúmeras publicações disponíveis sobre AM (ver [Mitchell 1997], por exemplo).

Inúmeras técnicas viabilizam o desenvolvimento de sistemas de aprendizado indutivo de máquina, dentre elas, redes neurais (RN)[1] têm se mostrado um modelo bastante eficiente para a aquisição automática de conhecimento (ver [Mitchell 1997], [Langley 1995], [Gallant 1994], [Haykin 1994], [Bishop 1997], por exemplo). Redes neurais são particularmente atrativas devido ao paralelismo potencial e à tolerância a ruído que oferecem.

Como discutido em [Palma Neto & Nicoletti 2005], "Definida de maneira informal e simplista, uma RN é uma rede com muitos processadores simples (cada um deles tendo, possivelmente, uma pequena quantidade de memória local) conectados por meio de canais de comunicação (conexões) aos quais usualmente estão associados valores (pesos) numéricos. Uma rede neural pode então ser caracterizada:

(1) por seus processadores, chamados neurônios;
(2) pela função de ativação que representa o estado do neurônio;
(3) pelo padrão de conexão existente entre os neurônios;
(4) por seu algoritmo de treinamento (também chamado de algoritmo de aprendizado)."

O algoritmo de treinamento de uma rede é um conjunto de regras por meio das quais os pesos das conexões são ajustados usando, para o ajuste, um conjunto de exemplos, chamado de *conjunto de treinamento.* Uma RN aprende a partir de um conjunto de treinamento – tal conjunto é de importância fundamental para o aprendizado da rede.

A arquitetura de uma rede neural pode ser caracterizada pelos neurônios que a compõem, pelo padrão de conexão entre eles e pela organização desses neurônios em

[1] Também referidas como redes neurais artificiais.

camadas (quando for o caso). A arquitetura de uma rede desempenha um papel fundamental na sua capacidade de generalização; sabe-se, entretanto, que quando da definição de uma RN na maioria das situações, não existe uma maneira de determinar o melhor número de camadas intermediárias da rede e tampouco o número de neurônios em cada uma das camadas.

O chamado aprendizado neural construtivo é caracterizado como um modelo de aprendizado neural que não pressupõe a definição de uma arquitetura de rede fixa, antes do início do treinamento. "A principal característica desse modelo de aprendizado é a construção dinâmica das camadas intermediárias da rede, à medida que vão sendo necessárias ao treinamento. Essa característica faz com que o processo de determinação da arquitetura da rede fique interligado ao próprio processo de aprendizado – ambos os processos, o de treinamento e o de construção da rede acontecem simultaneamente e são interdependentes" [Palma Neto & Nicoletti 2005].

Este trabalho é uma continuação do trabalho descrito em [Palma Neto 2004], no qual foram investigadas as versões originais de seis algoritmos neurais construtivos, a saber: os algoritmos, Tower e Pyramid, propostos em [Gallant 1986], que podem ser abordados como versões construtivas do algoritmo Perceptron; o algoritmo Tiling [Mézard & Nadal 1989], o algoritmo Upstart [Frean 1990a], o DistAl [Yang *et al.* 1998] e o Cascade Correlation [Fahlman & Lebiere 1990]. Os quatro primeiros algoritmos foram originalmente propostos para tarefas de aprendizado em domínios com duas classes apenas.

Este trabalho focaliza aprendizado neural construtivo considerando três frentes de investigação:

- Investigação dos algoritmos de treinamento de TLUs, outros que o Perceptron e variantes, a saber: Thermal, Thermal Modificado, MinOver e BCP.
- Os algoritmos neurais construtivos Shift e PTI [Amaldi & Guenin 1997], Offset [Martinez & Estève 1992], Perceptron Cascade [Burgess 1994] e Sequential [Marchand *et al.* 1990].
- As extensões dos algoritmos Tower, Pyramid, Tiling, Upstart e Perceptron Cascade para domínios multiclasses.

O trabalho investiga as extensões e os algoritmos listados acima, avaliando empiricamente cada um deles com relação a desempenho, vantagens e desvantagens, facilidade de treinamento, tamanho e topologia de rede criada, com o intuito de determinar

qual a real contribuição de cada um deles e a correspondente adequabilidade a situações de aprendizado.

Este trabalho está organizado da seguinte maneira: No Capítulo 2 são apresentados e discutidos o conceito de TLU (*Threshold Logic Unit*), o algoritmo Perceptron e cinco de suas variações: o algoritmo Pocket, o Pocket com Modificação Ratchet, o Thermal Perceptron, o Thermal Modificado e o MinOver. Na seção referente ao MinOver são propostas duas alterações que melhoram o desempenho deste algoritmo.

A apresentação e discussão do Perceptron e do conjunto de algoritmos que são baseados no Perceptron é importante para o entendimento do trabalho, uma vez que muitos dos algoritmos neurais construtivos utilizam o Perceptron (ou uma de suas variantes) como algoritmo básico.

É praticamente inexistente a experimentação de algoritmos construtivos usando outros algoritmos básicos, que não o Perceptron e algumas de suas variantes. Isto motivou o levantamento, a investigação e implementação de outras propostas de algoritmos de treinamento de TLU. O Capítulo 3 apresenta o resultado dessa investigação e, para tanto, descreve em detalhes o algoritmo BCP (*Barycentric Correction Procedure*). O algoritmo BCP apresentado no Capítulo 3 é, na verdade, uma variante que foi proposta neste trabalho para solucionar uma situação não prevista no algoritmo original.

O Capitulo 4 descreve brevemente as principais características dos algoritmos neurais construtivos: Tower, Pyramid, Tiling (e uma proposta deste trabalho, chamado de Tiling Híbrido) e Upstart. A apresentação e discussão de cada um desses algoritmos, ainda que breves, são necessárias uma vez que esses algoritmos serão comparados com os algoritmos neurais construtivos que são objeto de estudo desta dissertação e que são abordados no Capítulo 5. As implementações e discussões apresentadas no Capítulo 4 são também fundamentais para subsidiar a apresentação e o entendimento das extensões desses algoritmos para o tratamento de problemas multiclasses no Capítulo 7.

O Capítulo 5 diz respeito à segunda frente de investigação deste trabalho, a relativa aos cinco algoritmos neurais construtivos: Shift, Offset, PTI, Perceptron Cascade e Sequential. Na seção que investiga o algoritmo Offset esse trabalho propõe o algoritmo OffTiling que utiliza uma abordagem híbrida para a construção da rede. Cada uma das seções deste capítulo focaliza um dos algoritmos, descrevendo suas principais características, fornecendo seu pseudocódigo e exemplificando o seu funcionamento no problema paridade-3 ímpar.

Uma das frentes abordadas neste trabalho é o estudo das extensões de alguns algoritmos neurais construtivos para tratar problemas multiclasses. Para isso, uma introdução sobre o tratamento desse tipo de problema é apresentada no Capítulo 6, bem como são apresentadas duas possíveis abordagens para o treinamento de neurônios para problemas com mais de duas classes, a individual e a WTA (*winner takes all*).

A continuação da pesquisa envolvendo o aprendizado em domínios multiclasses está descrita no Capítulo 7, no qual são apresentadas e discutidas as extensões dos algoritmos Tower, Pyramid, Tiling, Upstart e Perceptron Cascade para problemas multiclasses.

O Capítulo 8 trata exclusivamente da apresentação e discussão da avaliação empírica dos algoritmos focalizados neste trabalho. O capítulo esta organizado em duas seções; a primeira contempla experimentos realizados em nove domínios de conhecimento com duas classes e a segunda, em cinco domínios de conhecimento multiclasse.

Finalmente, no Capítulo 9, são resumidas as principais conclusões do trabalho realizado, suas contribuições e elencadas possíveis tarefas que viabilizam a sua continuidade.

2 capítulo

O Perceptron e suas Variantes Pocket, PMR, Thermal, Thermal Modificado e MinOver

2.1 Introdução

O Perceptron [Rosenblatt *et al.* 1958] é o algoritmo básico utilizado em vários algoritmos neurais construtivos. O desempenho das redes obtidas usando algoritmos construtivos, portanto, é profundamente influenciado pelo desempenho do Perceptron.

Devido ao problema de desempenho do Perceptron em alguns domínios de conhecimento [Misnky & Papert 1969], muitas vezes é mais conveniente utilizar alguma de suas variantes. Este capítulo investiga a família Perceptron (ou seja, o Perceptron e algumas de suas variantes). O objetivo é avaliar as motivações para as diferentes propostas bem como identificar as efetivas contribuições de cada uma delas.

O capítulo apresenta inicialmente o Perceptron e, na seqüência, são abordadas as variantes Pocket [Gallant 1986], Pocket com Modificação Ratchet (PMR) [Gallant 1986], Thermal [Frean 1992], Thermal Modificado [Burgess 1994] e MinOver [Krauth & Mezard 1987], como mostra a Figura 2.1.

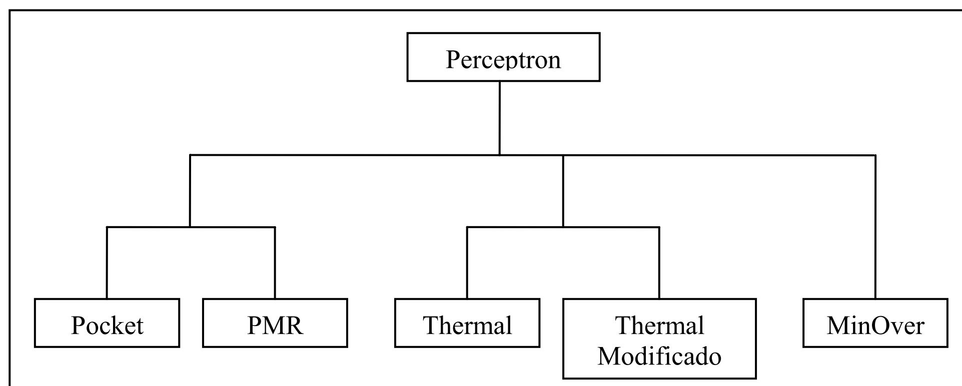


Figura 2.1 – O Perceptron e algumas de suas variantes

Os algoritmos Loss Minimization [Hrycej 1992] e AdaTron [Anlauf & Biehl 1989] não foram abordados nesta pesquisa por não terem sido cogitados na proposta desse trabalho. A investigação desses dois algoritmos, entretanto, é parte da proposta de continuaidade do

trabalho realizado, descrita em Conclusões. Este capítulo investiga o Perceptron e algumas de suas variantes, a saber: o Pocket, o Pocket com Modificação Ratchet, o Thermal e uma variação deste, referenciado como Thermal Modificado, e o MinOver.

2.2 O Perceptron

Uma *Threshold Logic Unit* (TLU), também conhecida como Perceptron, pode ser vista como um separador linear e pode ser treinada para classificar conjuntos de instâncias como pertencentes a uma, de duas regiões separadas por um hiperplano.

O treinamento de uma TLU presume a disponibilidade de um *conjunto de treinamento*[2]. Um conjunto de treinamento é um conjunto de instâncias de treinamento, usualmente descrito por vetores de pares atributo–valor_de_atributo e uma classe associada (o que caracteriza o aprendizado supervisionado[3]). No que segue um conjunto de treinamento será notado como em (2.1).

$$E = \{(E^1,C^1),(E^2,C^2),...,(E^n,C^n)\} \qquad (2.1)$$

no qual cada E^i, $1 \leq i \leq n$, é um vetor p-dimensional de números binários ou reais e C^i é a classe de E^i. Dependendo dos valores que C^i assume, a TLU recebe nomes especiais. Se $C^i \in \{-1,1\}$, a TLU é chamada bipolar, se $C^i \in \{0,1\}$, a TLU é chamada binária. No que segue será usado somente TLUs bipolares.

Os atributos que descrevem cada uma das instâncias (sejam instâncias de treinamento ou não) serão notados por X_i e o seu valor, por x_i, para $i \in \{0,...,p\}$, sendo X_0 a dimensão correspondente ao termo *bias*. Quando não for necessário discriminar entre instâncias que são (ou não) exemplos de treinamento, um vetor qualquer do espaço p-dimensional será referenciado por X.

Como comentado em [Parekh 1998] "Uma TLU bipolar implementa um hiperplano de dimensão (p–1) definido pelas coordenadas $X_1, X_2,..., X_p$ que passa por WX = 0 e particiona o espaço p dimensional das instâncias de treinamento em duas regiões". Sendo que

[2] Neste trabalho as expressões 'instância de treinamento' e 'exemplo de treinamento' serão usadas indistintamente.
[3] Uma instância de treinamento nem sempre incorpora a classe em sua descrição. Métodos de aprendizado de máquina que lidam com instâncias de treinamento nas quais as classes não comparecem são chamados métodos não supervisionados.

$W = \langle w_1,...,w_p\rangle$ são os pesos sinapticos que devem ser aprendido pela regra de atualização de pesos do Perceptron.

Uma vez determinado o vetor de pesos W, o Perceptron pode ser usado como classificador. Dada uma entrada p dimensiona $X = \langle x_1,...,x_p\rangle$, a saída $O(\langle x_1,\ldots,x_p\rangle)$ é calculada pelo Perceptron de acordo com a equação (2.1).

$$O(\langle x_1,\ldots,x_p\rangle) = \begin{cases} 1 \text{ se } \sum_{i=1}^{p} w_i x_i + b > 0 \\ -1 \text{ caso contrário} \end{cases} \tag{2.1}$$

onde cada w_i é uma constante real (peso), que determina a contribuição da entrada x_i na saída do Perceptron. O valor (b) é um limite (*bias*) que o produto interno, $w_1x_1 + w_2x_2 +....+ w_px_p$ entre a entrada e o vetor de pesos deve superar para que a saída do Perceptron seja ativa (i.e. igual a 1)[Mitchell 1997], como mostra a Figura 2.2.

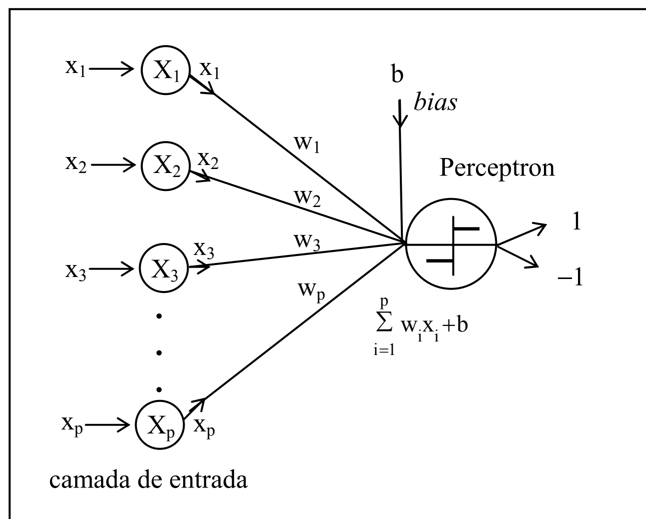


Figura 2.2 – O Perceptron: recebe como entrada valores reais, calcula o produto interno dos valores recebidos e então produz uma saída 1 se o resultado da combinação for maior do que zero e valor −1, caso contrário

Como o treinamento do Perceptron consiste na determinação de valores para os pesos $w_0, w_1,\ldots,w_p$ é conveniente tratar o termo *bias* como uma entrada adicional, de valor fixo ($X_0 = 1$), no conjunto de treinamento, como mostra a Figura 2.3.

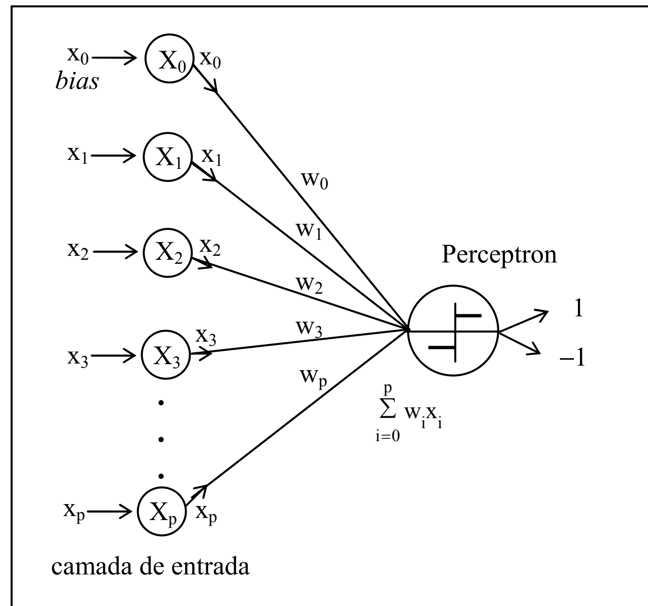


Figura 2.3 – Representação do Perceptron usando o termo *bias* como uma entrada adicional do conjunto de treinamento

Note que esta notação é equivalente à apresentada na Figura 2.2. Como o *bias* agora é considerado uma entrada, a equação (2.1) pode ser escrita como mostrado na equação (2.2).

$$O(\langle x_0, x_1, x_2, \ldots, x_p \rangle) = \begin{cases} 1 \text{ se } \sum_{i=0}^{p} w_i x_i > 0 \\ -1 \text{ caso contrário} \end{cases} \tag{2.2}$$

Como comentado anteriormente, o Perceptron pode ser visto como a representação da superfície de um hiperplano em um espaço p-dimensional onde os exemplos de treinamento estão representados. O Perceptron vai produzir saída com valor 1 para aqueles exemplos que estão em um lado do hiperplano ($WX > 0$) e saída -1 para os exemplos que estão do outro lado do hiperplano ($WX \leq 0$). Dessa forma se o conjunto de treinamento for *linearmente separável* o Perceptron classificará todas as instâncias corretamente.

Um conjunto é linearmente separável se existir um hiperplano que separe as instâncias pertencentes a classes diferentes. A Figura 2.4 (a) mostra uma situação em que as instâncias pertencentes à classe (+) e à classe (–) são linearmente separáveis, e mostra uma possível reta que as separa. Já na Figura 2.4 (b) não é possível encontrar uma reta que separe as instâncias de classe (+) das de classe (–).

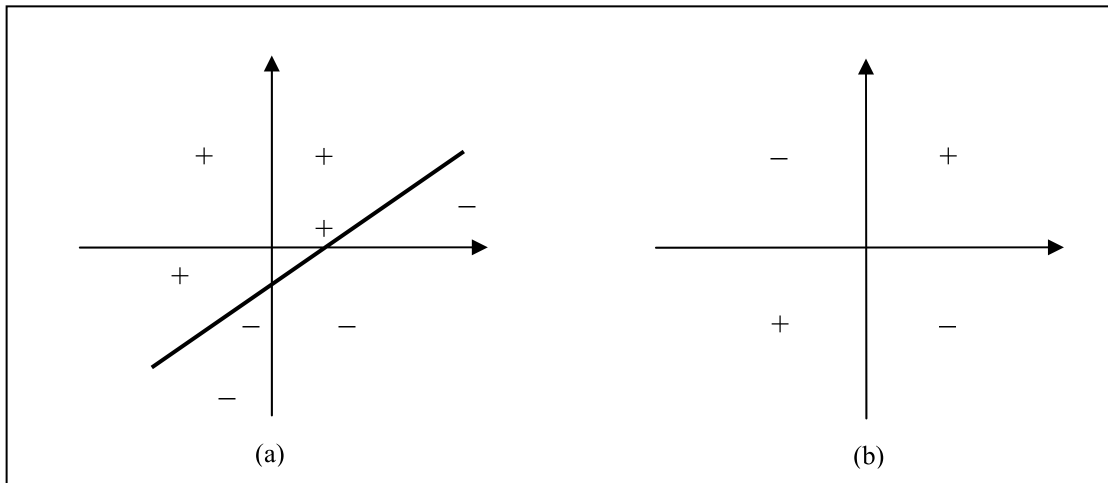


Figura 2.4 – Superfície de decisão representada por duas entradas
(a) Representação de um conjunto linearmente separável
(b) Conjunto não linearmente separável – não existe uma reta, no espaço bidimensional em questão, que separe as instâncias (+) das instâncias (–)

2.2.1 O Termo *Bias*

O termo *bias*, como mencionado anteriormente, é uma dimensão extra que é adicionada ao conjunto de treinamento, para o treinamento do Perceptron e de seus derivados, com o propósito de ajudar na determinação do vetor de pesos, durante o treinamento da TLU.

Essa dimensão adicional, com valor constante (geralmente igual a 1), aumenta a flexibilidade na determinação do vetor de pesos resultante do treinamento. Isso porque com a adição do termo *bias*, a capacidade do algoritmo de encontrar um hiperplano separador é aumentada. Sem o termo *bias* a busca por um hiperplano separador se limita à busca por um hiperplano que necessariamente passa pela origem. Muitas vezes um hiperplano que passa pela origem não consegue separar corretamente um conjunto linearmente separável.

A Figura 2.5 (a) mostra duas situações de aprendizados usando como conjunto de treinamento os seis pontos mostrados na Figura 2.5 com classes (+) e (–). Na Figura 2.5(a) o hiperplano foi obtido sem o uso do *bias*; já na Figura 2.5(b) o hiperplano foi obtido com o uso do *bias*. Note que, como o hiperplano em (a) passa pela origem ele não consegue separar completamente as instâncias. Já o hiperplano mostrado em (b) separa corretamente as instâncias de classe (+) das instâncias de classe (–).

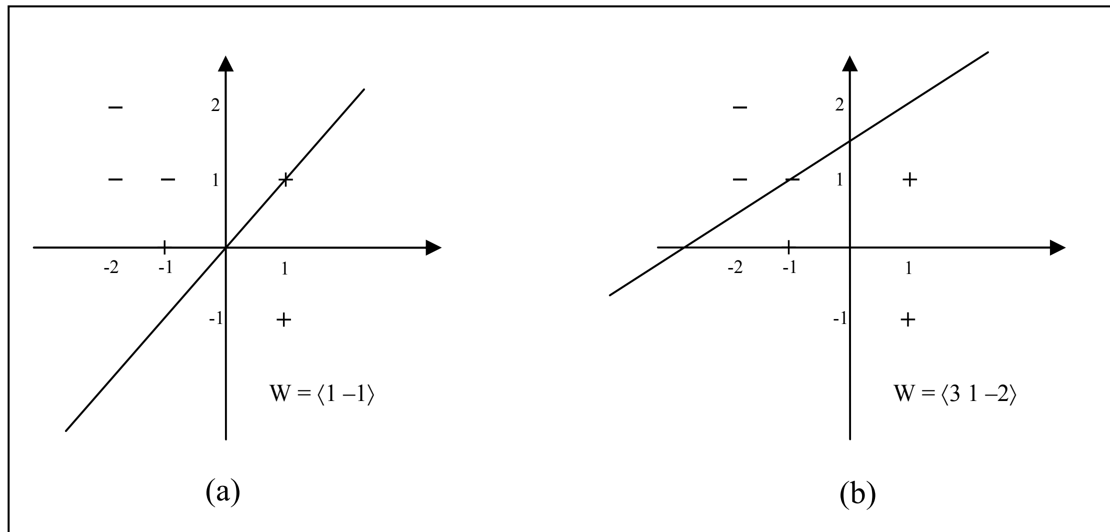


Figura 2.5 – Exemplos relativos ao termo *bias* - (a) hiperplano obtido sem o uso de *bias*; (b) hiperplano obtido com uso do *bias*

2.2.2 O Algoritmo Perceptron

A regra de aprendizado Perceptron é o algoritmo mais básico usado para treinar uma TLU [Rosenblatt *et al*. 1958].

Considere um conjunto de treinamento E tal que $|E| = n$, cujas instâncias pertencem à classe 1 ou –1. O conjunto E pode então ser abordado como $E = E1 \cup E2$, tal que $E1 = \{(E^p, C^p) \text{ e } C^p = 1\}$ e $E2 = \{(E^q, C^q) \text{ e } C^q = -1\}$, com p e q $\in \{1,..., n\}$.

O algoritmo Perceptron busca por um vetor de pesos W tal que para todo $E^p \in E1$, $WE^p > 0$ e para todo $E^q \in E2$, $WE^q \leq 0$. Durante o aprendizado o vetor de pesos W é modificado pelo algoritmo toda vez que não classificar corretamente a instância em questão. Suponha que a k-ésima instância, E^k, $1 \leq k \leq n$, tenha sido classificada incorretamente. O vetor W será atualizado de acordo com a equação (2.3).

$$W = W + C^k E^k \qquad (2.3)$$

O pseudocódigo do algoritmo de treinamento Perceptron, apresentado em Algoritmo 2.1, leva em consideração que o conjunto de treinamento E é percorrido seqüencialmente (uma outra possibilidade é o acesso randômico). Esse fato não terá relevância aqui, pois nenhum critério de parada foi estabelecido para achar um vetor de pesos quase ótimo. O algoritmo apresentado só termina se classificar corretamente todas as instâncias de

treinamento, i.e. se o conjunto de treinamento for linearmente separável. Desse modo a maneira como o conjunto de treinamento será percorrido é irrelevante.

```
Class Perceptron
begin
{Entrada: E - conjunto de treinamento com n instâncias da forma:
            (x^k_0, x^k_1, x^k_2,..., x^k_p, C^k), sendo que 1 ≤ k ≤ n}
{Saída: W}

method Perceptron (Matrix E)
  begin
    W ← ⟨0, 0, 0,...,0⟩ {W é iniciado como um vetor nulo com p+1 posições }
    correto ← false
    while not correto do
      begin
        correto ← true
        for k ← 1 to n do
          if (W.E^k ≤ 0 and C^k > 0) or (W.E^k > 0 and C^k ≤ 0) then  {erra}
            begin
              correto ← false
              W ← W + C^kE^k
            end
      end
  end
end
```

Algoritmo 2.1 – Pseudocódigo do algoritmo Perceptron

2.2.3 Exemplo de Execução do Algoritmo Perceptron

Nesse exemplo de execução foi usado o conjunto de treinamento que representa a função booleana *and*, representada na Tabela 2.1, com a dimensão associada ao *bias* incluída.

Tabela 2.1 – Conjunto de treinamento que representa da função *and*, com o *bias* incluído

Exemplos E^k (k = 1,..., 4)	**X_0 (*bias*)**	**X_1**	**X_2**	**Classe C^k (k = 1,..., 4)**
E^1	1	1	1	1
E^2	1	1	–1	–1
E^3	1	–1	1	–1
E^4	1	–1	–1	–1

Como a função *and* é linearmente separável, o Perceptron consegue achar um vetor de pesos W que classifica corretamente todas as instâncias de treinamento. Note na Tabela 2.2 que o algoritmo só termina quando percorrer o conjunto de treinamento todo (iteração 5 a 8) sem cometer erros.

Tabela 2.2 – Execução passo a passo do Perceptron para o conjunto de treinamento da Tabela 2.1

Iteração	Exemplo	W atual	Acerta	Ação
1	E^1	⟨0 0 0⟩	Não	$W \leftarrow W + E^1$
2	E^2	⟨1 1 1⟩	Não	$W \leftarrow W - E^2$
3	E^3	⟨0 0 2⟩	Não	$W \leftarrow W - E^3$
4	E^4	⟨–1 1 1⟩	Sim	Nenhuma
5	E^1	⟨–1 1 1⟩	Sim	Nenhuma
6	E^2	⟨–1 1 1⟩	Sim	Nenhuma
7	E^3	⟨–1 1 1⟩	Sim	Nenhuma
8	E^4	⟨–1 1 1⟩	Sim	Nenhuma

2.3 O Algoritmo Pocket

Quando o conjunto de treinamento não for linearmente separável o Perceptron não consegue classificar corretamente todo o conjunto de treinamento, pois não existe um vetor de pesos que classifique corretamente todas as instâncias de treinamento. Isso ocorre porque não existe um hiperplano que separa essas instâncias de treinamento usando como critério as suas classes.

Em casos como esse, onde não é possível achar um vetor de pesos que classifique todos os exemplos corretamente, é conveniente achar um vetor de pesos que classifique o maior número possível de instâncias de treinamento. Esse vetor de pesos é um vetor de pesos ótimo[4] para esse conjunto de treinamento.

Como citado em [Gallant 1994], "o problema do método de treinamento do Perceptron é que o mesmo usa apenas reforço negativo isto é, o Perceptron apenas modifica o vetor de pesos W se sua saída for diferente da saída esperada. Não existe nenhum meio de incentivar, reforçar ou recompensar vetores de pesos W que classifiquem corretamente as entradas de treinamento".

O problema do Perceptron em considerar somente o reforço negativo é solucionado pelo algoritmo Pocket, proposto em [Gallant 1986]. O Pocket é basicamente o Perceptron com um reforço positivo. Esse reforço é dado pelo número de instâncias consecutivas que o vetor de pesos corrente classifica corretamente. Toda vez que o algoritmo encontra um vetor de pesos W, que classifica um número maior de instâncias consecutivas, este vetor é salvo em um vetor de pesos extra[5] chamado WPoc.

O pseudocódigo do algoritmo Pocket é descrito em Algoritmo 2.2. Note que o vetor de pesos extra, Wpoc, é atualizado toda vez que o algoritmo acha um vetor de pesos W que

[4] Essa terminologia será usada no restante do texto.
[5] O algoritmo guarda o vetor extra "em seu bolso", daí o nome Pocket (bolso).

classifica corretamente um maior número de exemplos consecutivos. Para isso são usadas as variáveis consecW e consecWpoc. Essas variáveis armazenam o número máximo de instâncias de treinamento que W e Wpoc, respectivamente, classificam corretamente sem cometer erro.

O conjunto de treinamento desta vez é percorrido de forma aleatória, de maneira que o número de exemplos consecutivos classificados corretamente possa variar. Dessa maneira, a probabilidade de achar uma solução ótima é maior. No pseudocódigo, o método *seleciona*() é responsável por gerar, de modo aleatório, um número inteiro dentro do intervalo que é passado e este como parâmetro.

```
Class Pocket
begin
{Entradas: E - conjunto de treinamento com n instâncias da forma:
            (x^k_0, x^k_1, x^k_2,..., x^k_p, C^k), sendo que 1 ≤ k ≤ n
  MAX - número pré-determinado de iterações}
{Saída: Wpoc}

method Pocket(Matrix E)
  begin
    W ← ⟨0, 0, 0,..., 0⟩ {W é iniciado como um vetor nulo com p+1 posições }
    Wpoc ← W
    consecW ← 0
    consecWpoc ← 0
    it ← 0
    itMax ← MAX
    while (it < itMax) do
      begin
        k ← seleciona(1,n)
        if (WE^k > 0 and C^k > 0) or (WE^k ≤ 0 and C^k ≤ 0) then
          begin
            consecW ← consecW + 1
            if consecW > consecWpoc then
              begin
                consecWpoc ← consecW
                Wpoc ← W
              end
          end
        else
          begin
            consecW ← 0
            W ← W + C^kE^k
          end
        it ← it + 1
      end {while}
  end  {method}
end {Class}
```

Algoritmo 2.2 – Pseudocódigo do algoritmo Pocket

O Pocket tem alta probabilidade de encontrar um vetor de pesos "ótimo" se for executado por tempo suficiente, porém não existe garantia que o vetor será encontrado. Formalmente é possível demonstrar (consultar [Gallant 1990] para detalhes da prova) que

para um conjunto finito de exemplos e uma probabilidade P < 1, existe um valor *m* tal que após *m* iterações, o algoritmo chegará a um vetor com o mínimo de erros, com uma probabilidade maior que P.

2.3.1 Exemplo de Execução do Algoritmo Pocket

Para exemplificar o algoritmo Pocket será usado o conjunto de treinamento que representa o problema de paridade-3, cujas instâncias estão na Tabela 2.3, com a dimensão associada ao *bias* já incluída.

Tabela 2.3 – Conjunto de treinamento que representa a o problema paridade-3, com o temo *bias* incluído

Exemplos E^k (k = 1,...,8)	**X_0 (*bias*)**	**X_1**	**X_2**	**X_3**	**Classe C^k (k = 1,...,8)**
E^1	1	1	1	1	1
E^2	1	1	1	–1	–1
E^3	1	1	–1	1	–1
E^4	1	1	–1	–1	1
E^5	1	–1	1	1	–1
E^6	1	–1	1	–1	1
E^7	1	–1	–1	1	1
E^8	1	–1	–1	–1	–1

Tabela 2.4 – Exemplo de execução, passo a passo, do algoritmo Pocket utilizando como conjunto de treinamento os exemplos da Tabela 2.3

it	**Exemplo**	**W**	**consecW**	**Wpoc**	**consecWpoc**	**Acerta**	**Ação**
1	E^6	⟨0 0 0 0⟩	0	⟨0 0 0 0⟩	0	Não	consecW ← 0 $W \leftarrow W + E^6C^6$
2	E^1	⟨1 -1 1 -1⟩	0	⟨0 0 0 0⟩	0	Não	consecW ← 0 $W \leftarrow W + E^1C^1$
3	E^2	⟨2 0 2 0⟩	0	⟨0 0 0 0⟩	0	Não	consecW ← 0 $W \leftarrow W + E^2C^2$
4	E^7	⟨1 -1 1 1⟩	0	⟨0 0 0 0⟩	0	Sim	consecW ← consecW + 1 consecWpoc ← consecW Wpoc ← W
5	E^3	⟨1 -1 1 1⟩	1	⟨1 -1 1 1⟩	1	Sim	consecW ← consecW + 1 consecWpoc ← consecW Wpoc ← W
6	E^4	⟨1 -1 1 1⟩	2	⟨1 -1 1 1⟩	2	Não	consecW ← 0 $W \leftarrow W + E^4C^4$
7	E^6	⟨2 0 0 0⟩	0	⟨1 -1 1 1⟩	2	Sim	consecW ← consecW + 1
8	E^1	⟨2 0 0 0⟩	1	⟨1 -1 1 1⟩	2	Sim	consecW ← consecW + 1
9	E^7	⟨2 0 0 0⟩	2	⟨1 -1 1 1⟩	2	Sim	consecW ← consecW + 1 consecWpoc ← consecW Wpoc ← W
10	E^5	⟨2 0 0 0⟩	3	⟨2 0 0 0⟩	3	Não	consecW ← 0 $W \leftarrow W + E^5C^5$
11	E^8	⟨1 1 -1 -1⟩	0	⟨2 0 0 0⟩	3	Não	consecW ← 0 $W \leftarrow W + E^8C^8$

Note no exemplo de execução da Tabela 2.4, que o Pocket encontra um vetor de pesos ótimo na terceira iteração (it = 3) e que este vetor é salvo em Wpoc na quarta iteração. Como o Pocket, no entanto, não possui mecanismos para evitar uma situação de azar, o vetor ótimo é perdido na iteração it = 9.

2.4 O Algoritmo Pocket com Modificação Ratchet (PMR)

Muito embora o Pocket seja considerado um algoritmo que obtém bons resultados, não existem garantias que obterá um vetor de pesos ótimo ou mesmo bom. Apesar do Pocket implementar mecanismos de reforço positivo e negativo, o algoritmo não previne que um vetor ruim, eventualmente, substitua um bom.

Como as instâncias são selecionadas de maneira aleatória pelo algoritmo Pocket, duas situações de azar podem ocorrer:

(1) Como alguma(s) instância(s) pode(m) ser selecionada(s) várias vezes, o conjunto de treinamento usado pelo Pocket pode ser apenas um subconjunto do conjunto original de treinamento. Caso isso aconteça, o vetor de pesos gerado representa apenas o subconjunto em questão.

(2) Durante a execução o Pocket encontra um vetor de pesos ótimo para um determinado conjunto de treinamento e o armazena em Wpoc. Como a execução do algoritmo ainda está em andamento, pode acontecer que um vetor W, pior que Wpoc, classifique mais exemplos (situação que surge quando é feita a seleção repetitiva de alguns exemplos). Esse vetor W é então salvo em Wpoc, o que faz com que o vetor ótimo encontrado previamente seja perdido.

Uma modificação proposta em [Gallant 1986] para o algoritmo Pocket, chamada Ratchet, corrige os problemas citados anteriormente. Nessa modificação o vetor de pesos auxiliar (Wpoc) só é atualizado com o vetor de pesos corrente (W) se W, além de classificar corretamente mais instâncias consecutivas que Wpoc, também classificar corretamente mais instâncias de treinamento distintas que Wpoc.

Para isso o algoritmo armazena o número de instâncias do conjunto de treinamento que Wpoc classifica corretamente na variável corretosWpoc, e o número de instâncias de treinamento distintas que W classifica corretamente é armazenado na variável corretosW.

O Algoritmo 2.3 apresenta o pseudocódigo do algoritmo Pocket com Modificação Ratchet (ou PMR, como será chamado). O que muda no pseudocódigo do PMR em relação ao Pocket é, basicamente, a adição das variáveis corretosWpoc e corretosW. O método *nroCorretos*() determina quantas instâncias distintas, do conjunto de treinamento, são classificadas corretamente pelo vetor de pesos corrente W. Esse método é chamado sempre que W classificar corretamente mais exemplos consecutivos do que Wpoc, ou seja, sempre que consecW > consecWpoc.

```
Class PMR
begin
{Entradas: E - conjunto de treinamento com n instâncias da forma:
              (x^k_0, x^k_1, x^k_2,..., x^k_p, C^k), sendo que 1 ≤ k ≤ n
  MAX - número pré-determinado de iterações}
{Saída: Wpoc}

method PMR(Matrix E)
  begin
    W ← ⟨0, 0, 0,..., 0⟩ {W é iniciado como um vetor nulo com p+1 posições }
    Wpoc ← W
    consecW ←  0
    consecWpoc ← 0
    corretosW ← 0
    corretosWpoc ← 0
    it ← 0
    itMax ← MAX
    while it < itMax do
      begin
        k ← seleciona(1,n)
        if (WE^k > 0 and C^k > 0) or (WE^k ≤ 0 and C^k ≤ 0) then
          begin
            consecW ← consecW + 1
            if consecW > consecWpoc then
              begin
                corretosW ← nroCorretos()
                if corretosW > corretosWpoc then
                  begin
                    consecWpoc ← consecW
                    corretosWpoc ← corretosW
                    Wpoc ← W
                  end
              end
          end
        else
          begin
            consecW ← 0
            W ← W + C^kE^k
          end
        it ← it  + 1
      end {while}
  end {method}
end {Class}
```

Algoritmo 2.3 – Pseudocódigo do algoritmo Pocket com Modificação Ratchet (PMR)

O algoritmo PMR garante que um vetor ótimo é achado depois de um certo número de iterações. Como as instâncias são consideradas de maneira aleatória, este número pode variar de execução para execução com um mesmo conjunto de treinamento. Desse modo, assim como no Pocket, não é possível prever o número ideal de iterações. Mesmo assim, a modificação Ratchet melhora consideravelmente o desempenho do Pocket, pois mesmo que essa modificação nem sempre consiga achar um vetor ótimo em um tempo considerável, ela não permite que Wpoc piore durante o treinamento.

2.4.1 Exemplo de Execução do PMR

A Tabela 2.5 ilustra um exemplo de execução passo a passo do algoritmo Pocket com Modificação Ratchet. Na Tabela 2.5, as variáveis cW e cWpoc contabilizam o número de classificações consecutivas corretas feitas para W e Wpoc respectivamente. As variáveis aW e aWpoc o número de classificações distintas corretas feitas também para W e Wpoc respectivamente. O conjunto de treinamento não linearmente separável, usado neste exemplo, representa o problema de paridade-3 apresentado na Tabela 2.3.

Tabela 2.5 – Exemplo de execução passo a passo do algoritmo PMR, para o conjunto de treinamento da Tabela 2.3

it	Ex	W	cW	Wpoc	cWpoc	Acerta	aW	aWpoc	Ação
1	E^4	⟨0 0 0 0⟩	0	⟨0 0 0 0⟩	0	Não	0	0	consecW ← 0 W ← W + C^4E^4
2	E^6	⟨1 1 -1-1⟩	0	⟨0 0 0 0⟩	0	Não	0	0	consecW ← 0 W ← W + C^6E^6
3	E^8	⟨2 0 0 -2⟩	0	⟨0 0 0 0⟩	0	Não	0	0	consecW ← 0 W ← W + C^8E^8
4	E^1	⟨1 1 1 -1⟩	0	⟨0 0 0 0⟩	0	Sim	0	0	consecW ← consecW + 1 corretosW ← *nroCorretos*() corretosWpoc ← corretosW Wpoc ← W
5	E^1	⟨1 1 1 -1⟩	1	⟨1 1 1-1⟩	1	Sim	6	6	consecW ← consecW + 1 corretosW ← *nroCorretos*()
6	E^4	⟨1 1 1 -1⟩	2	⟨1 1 1-1⟩	1	Sim	6	6	consecW ← consecW + 1 corretosW ← *nroCorretos*()
7	E^5	⟨1 1 1 -1⟩	3	⟨1 1 1-1⟩	1	Sim	6	6	consecW ← consecW + 1 corretosW ← *nroCorretos*()
8	E^7	⟨1 1 1 -1⟩	4	⟨1 1 1-1⟩	1	Não	6	6	consecW ← 0 W ← W + C^7E^7
9	E^6	⟨2 0 0 0⟩	0	⟨1 1 1-1⟩	1	Sim	6	6	consecW ← consecW + 1 corretosW ← *nroCorretos*()
10	E^6	⟨2 0 0 0⟩	1	⟨1 1 1-1⟩	1	Sim	4	6	consecW ← consecW + 1 corretosW ← *nroCorretos*()

Note que a primeira vez que W classifica corretamente a instância de treinamento corrente ocorre na quarta iteração. Como a variável aWpoc tem valor 0 e aW > aWpoc, o vetor de pesos encontrado (W) é salvo em Wpoc na iteração it = 5.

Note que o vetor de pesos encontrado na iteração 4, ($W = \langle 1\ 1\ 1\ -1 \rangle$) é um vetor ótimo para este conjunto de treinamento; como o conjunto de treinamento não é linearmente separável, o vetor encontrado ainda comete erros de classificação. Este vetor, no entanto, não será substituído, pois não haverá nenhum outro que classifique mais que 6 instâncias de treinamento.

2.5 O Algoritmo Thermal Perceptron

Quando o conjunto de treinamento não for linearmente separável, o processo de atualização do vetor de pesos utilizado pelo Perceptron não garante que um vetor ótimo seja encontrado. De fato, o Perceptron não garante sequer que o vetor de pesos tenha o seu desempenho melhorado ao longo do treinamento.

O Thermal Perceptron, proposto em [Frean 1992], é uma variação do Perceptron que controla a atualização do vetor de pesos durante a fase de treinamento e previne mudanças drásticas no seu valor. Mudanças drásticas no vetor de pesos ocorrem no treinamento do Perceptron porque esse algoritmo procede da mesma maneira toda vez que comete um erro. Dessa forma, um novo vetor de pesos pode classificar incorretamente um exemplo que estava sendo corretamente classificado pelo vetor de pesos anterior.

Considere a instância de treinamento $E^k = \langle x_0,\ldots,x_p \rangle$, e o vetor de pesos $W = \langle w_0,\ldots,w_p \rangle$. O potencial de ativação associado à instância E^k e notado por v^k é o valor do produto interno entre o vetor de pesos e a instância E^k, mostrado na equação (2.4).

$$v^k = \sum_{i=0}^{p} w_i x_i \tag{2.4}$$

O fato do Perceptron atualizar o vetor de pesos com o valor de v^k causa flutuações no aprendizado em conjuntos de treinamento não linearmente separáveis. Um erro de classificação com um alto valor de v^k associado é mais difícil de ser corrigido do que um erro com um baixo valor v^k associado. Isso acontece porque corrigir um erro com um alto valor de

v^k exige uma mudança substancial no vetor de pesos o que, certamente, vai provocar a classificação incorreta de instâncias previamente classificadas corretamente.

A idéia do Thermal é corrigir primeiro os erros que possuem v^k próximos de zero. Uma maneira simples de fazer isso é modificar o vetor de pesos W segundo a equação (2.5).

$$W = W - \alpha\left(C^k - O^k\right)E^k e^{-|v^k|/T} \tag{2.5}$$

na qual a função exponencial *e* controla a intensidade da mudança no vetor de pesos, pois $e^{-|v^k|/T}$ varia entre 0 e 1 de acordo com a variação de $|v^k|/T$. Se $|v^k|/T$ é alto o valor da função exponencial $e^{-|v^k|/T}$ é próximo de 0 e o vetor de pesos praticamente não muda. Por outro lado, se $|v^k|/T$ é pequeno o valor de $e^{-|v^k|/T}$ é próximo de 1.

Portanto se o parâmetro T (chamado temperatura) for decrescido linearmente de um valor inicial T_0, relativamente grande, até 0 em um número pré-determinado de iterações MAX, o vetor de pesos se estabilizará naturalmente. Uma maneira descrita em [Amaldi 1994] e [Amaldi *et al.* 1998] para fazer com que T varie dessa forma é dada nas equações (2.6).

$$T = \gamma T_0 \qquad \text{e} \qquad \gamma = 1 - it / MAX \tag{2.6}$$

nas quais it é o número da iteração atual e MAX o número total de iterações, definido previamente. Se os valores do parâmetro α variarem de maneira decrescente no intervalo [0,1], a partir de 1, ao mesmo tempo em que T é reduzido de T_0 a 0, o algoritmo terá melhores resultados [Frean 1992]. Para que α varie dessa maneira α deve ser igual a T/T_0.

O algoritmo Thermal é uma boa variação do Perceptron para conjuntos de exemplos não linearmente separáveis; seu desempenho, no entanto, depende totalmente da escolha inicial de T_0 e MAX. O Algoritmo 2.4 apresenta o pseudocódigo do algoritmo Thermal Perceptron.

```
Class Thermal
begin
{Entrada: E - conjunto de treinamento com n instâncias da forma:
              (xk0, xk1, xk2,..., xkp, Ck), sendo que 1 ≤ k ≤ n
 MAX - número máximo de ciclos, pré-determinado
 MaxTemp - Temperatura inicial pré-determinada }
{Saída: W}

method Thermal(Matrix E)
  begin
    W ← ⟨0, 0, 0,..., 0⟩ {W é iniciado como um vetor nulo com p+1 posições }
    T ← 1
    T0 ← MaxTemp
    it ← 0
    itMax ← MAX

    while T > 0 do
      begin
        k ← seleciona(1,n)
        vk ← WEk
        if (vk > 0) then
          O = 1
        else
          O = –1
        T ← 1 – (it / itMax)
        W ← W – T/T0(Ck – O)Ek e-|vk|/T
        it ← it + 1
      end
  end
end
```

Algoritmo 2.4 – Pseudocódigo do algoritmo Thermal Perceptron

2.5.1 Exemplo de Execução do Algoritmo Thermal Perceptron

Para exemplificar o processo de funcionamento do Thermal Perceptron e para permitir a visualização da mudança no vetor de pesos foi usado o conceito de paridade-3, cujo conjunto de treinamento está na Tabela 2.3. Na Tabela 2.6 é apresentada a execução passo a passo do Thermal Perceptron.

Tabela 2.6 – Exemplo de execução passo a passo do algoritmo Thermal Perceptron

Iteração	Exemplo	W	v^k	Acerta	T
1	E^3	⟨0 0 0 0⟩	0	Sim	1
2	E^4	⟨0 0 0 0⟩	0	Não	0.96
3	E^6	⟨2,57 –2,57 2,57 2,57⟩	5,15	Sim	0.93
4	E^5	⟨2,57 –2,57 2,57 2,57⟩	10,31	Não	0.9
5	E^4	⟨2,57 –2,57 2,57 2,57⟩	–5,15	Não	0.86
6	E^8	⟨2,57 –2,58 2,58 2,58⟩	–0,01	Sim	0.83

Note na execução do Thermal na Tabela 2.6, que a partir da iteração 3 o vetor de pesos começa a estabilizar, pois o vetor encontrado na iteração 2 (que comete apenas dois erros no conjunto de treinamento) é um vetor "ótimo" para o exemplo em questão. Esse vetor erra

somente as instâncias de treinamento E^4 e E^5, o fator de correção, portanto, altera o vetor de pesos de maneira irrelevante, pois só o fará quando for escolhidas uma dessas instâncias e a alteração será cada vez menor devido à diminuição do valor da temperatura.

2.5.2 O Algoritmo Thermal Perceptron Modificado

Os principais problemas associados ao algoritmo Thermal Perceptron estão relacionados à determinação de um valor conveniente para a temperatura inicial T_0 e ao estabelecimento de um limite para o número máximo de iterações MAX. Esse fato apresentado na literatura (ver [Burgess 1994] por exemplo) foi constatado durante a avaliação empírica dos algoritmos investigados nesse trabalho, no Capítulo 8. O valor dessas variáveis são deixados a cargo do usuário.

Uma proposta apresentada em [Burgess 1994] trata o problema do estabelecimento do valor da temperatura inicial T_0. A proposta consiste em iniciar o parâmetro T_0 com o valor 1 e atualizá-lo ao final de cada iteração, com base no valor anterior de T_0 e na média (λ) dos valores de v^k ($1 \leq k \leq n$), gerados pelo vetor de pesos corrente. Uma justificativa para esta atualização é a de que o valor de T_0 deve ter magnitude da ordem dos valores de v^k (como comentado em [Frean 1990b]).

A regra para a atualização de T_0 proposta em [Burgess 1994] é descrita pela equação (2.7). O fato de T_0 variar de acordo com a equação (2.7) previne que sua variação oscile.

$$T_0 = (2T_0 + 2\lambda) / 3 \qquad (2.7)$$

A versão do Thermal Perceptron que implementa a regra para a atualização de T_0, como proposta em [Burgess 1994], apresentada em (2.7) é referenciada nesta dissertação como Thermal Modificado. Testes realizados durante o desenvolvimento desse trabalho, entretanto, evidenciaram que o Thermal Modificado tem um desempenho pior que o Thermal, quando o Thermal é usado com valores ótimo para T_0 e MAX.

O ajuste dos valores de T_0 durante a execução, permite ao Thermal Modificado obter, em muitos casos, vetores de pesos bons ou mesmo ótimos sem que o parâmetro T_0 tenha que ser 'adivinhado' previamente (como acontece com o Thermal). O pseudocódigo do Thermal Modificado é mostrado em Algoritmo 2.5.

```
Class ThermalModificado
begin
{Entrada: E - conjunto de treinamento com n instâncias da forma:
            (x^k_0, x^k_1, x^k_2,..., x^k_p, C^k), sendo que 1 ≤ k ≤ n
  MAX - número máximo de iterações, pré-determinado}
 {Saída: Wpoc}

method ThermalModificado(Matrix E)
  begin
    W ← ⟨0, 0, 0,..., 0⟩ {W é iniciado como um vetor nulo com p+1 posições}
    T ← 1;  T0 ← 1{Temperatura inicial}
    it ← 0;  itMax ← MAX
    corretosWpoc ← 0

    while T > 0 do
      begin
        k ← seleciona(1,n)
        vk ← WE^k
        if (vk > 0) then
          O = 1
        else
          O = –1
        T ← 1 – (it / itMax)
        W ← W – T/T0(C^k – O)E^k e^(-|vk|/T)
        it ← it + 1
        mediaVk ← mediaVks()
        T0 ← (2T0 + 2mediaVk) / 3
        corretosW ← nroCorretos()
        if corretosW > corretosWpoc then
          begin
            corretosWpoc ← corretosW
            Wpoc ← W
          end
      end
  end

method Float mediaVks()
  begin
    soma ← 0
    for i ← 1 to n do
      soma ← soma + WE^i
    return soma / n
  end
end {Class}
```

Algoritmo 2.5 – Pseudocódigo do algoritmo Thermal Modificado

Apesar do Thermal Modificado eliminar a necessidade de se determinar um valor para T_0 antes do início do treinamento, ela não resolve o problema associado ao valor do parâmetro MAX (número máximo de iterações), responsável pelo número de iterações do algoritmo e pela maneira com que a temperatura T decresce até 0.

2.5.3 Exemplo de Execução do Algoritmo Thermal Modificado

Para este exemplo de execução considere o conjunto de treinamento apresentado na Tabela 2.3. A Tabela 2.7 mostra os valores das variáveis T, T_0, vk, bem como o vetor de

pesos W e o número de instâncias corretamente classificadas por W. O valor pré-determinado para MAX para este exemplo foi 10, a fim de mostrar a variação do parâmetro T de 1 à 0.

Tabela 2.7 – Exemplo de execução do Algoritmo Thermal modificado com o conjunto de treinamento da Tabela 2.3

ciclo	W	Instância	T_0	T	vk	Acertos
0	⟨0 0 0 0⟩	6	1	1	0	4
1	⟨2 –2 2 –2⟩	1	2	0,9	0	2
2	⟨2,9 –1,1 2,9 –1,1⟩	7	3,2	0,8	0	3
3	⟨3,4 –1,5 2,4 –0,6⟩	6	4,4	0,7	8	5
4	⟨3,4 –1,5 2,4 –0,6⟩	3	5,2	0,6	–1,2	5
5	⟨3,4 –1,5 2,4 –0,6⟩	5	5,7	0,5	6,7	5
6	⟨3,4 –1,5 2,4 –0,6⟩	1	6,1	0,4	3,5	5
7	⟨3,4 –1,5 2,4 –0,6⟩	6	6,3	0,3	7,9	5
8	⟨3,4 –1,5 2,4 –0,6⟩	4	6,7	0,19	4,5	5
9	⟨3,4 –1,5 2,4 –0,6⟩	7	6,5	0,09	1,9	5
10	⟨3,4 –1,5 2,4 –0,6⟩	2	6,6	0	4,8	5

Note na Tabela 2.7 que o desempenho do Thermal Modificado foi pior que o desempenho do algoritmo original, pois esta versão obteve um vetor de pesos que classifica 5 instâncias de treinamento e a versão original obteve um vetor de pesos ótimo (ver Tabela 2.6).

Como mencionado anteriormente, o Thermal original encontra vetores ótimos quando os valores de T_0 e de MAX são adequados para o conjunto de treinamento em questão, como é o caso do exemplo da Tabela 2.6.

2.6 O Algoritmo MinOver

Um outro algoritmo usado para o treinamento de TLUs baseado no Perceptron é o algoritmo MinOver, proposto em [Krauth & Mezard 1987]. Apesar do MinOver ser um algoritmo bastante estável, na maioria das vezes não encontra um vetor ótimo.

O algoritmo MinOver inicia o treinamento da TLU com um vetor de pesos W, inicializado com valores aleatórios do intervalo [0,1]. Para a atualização de W, a cada iteração o algoritmo determina o valor de S calculado pelas equações em (2.8).

$$S_k = C^k \sum_{i=0}^{p} w_i x_i^k \text{, } \forall\, k \in \{1,...,n\} \quad \text{e} \quad S = \min_{1 \le k \le n} \{S_k\} \qquad (2.8)$$

Suponha que $S = S_j$. Caso o valor S_j seja menor ou igual ao valor do parâmetro de estabilidade Δ, a instância E^j é usada para atualizar o vetor W, como mostrado em (2.9).

$$W \leftarrow W + C^j E^j \tag{2.9}$$

Se, no entanto, S_j for maior que o parâmetro de estabilidade então nada é feito. Note no pseudocódigo do MinOver, mostrado em Algoritmo 2.6, que quando a iteração corrente it for par, o valor de S é tomado em módulo. Isto não faz parte do algoritmo original, porém mostrou-se mais eficiente que o original nos testes realizados.

O grande problema do algoritmo MinOver é ajustar o parâmetro de estabilidade Δ, uma vez que é impossível fixar um valor para este parâmetro que seja bom o suficiente para um conjunto de treinamento específico. Duas alternativas podem ser adotadas; a primeira é iniciar Δ com um valor relativamente grande e, decrementá-lo ao longo do treinamento. A segunda é iniciar Δ com um valor pequeno e incrementá-lo durante o treinamento.

Nos experimentos conduzidos a segunda alternativa mostrou-se um pouco superior à primeira, pelo fato de que não acontece um afunilamento no parâmetro que comprometa a regra de atualização do algoritmo. A versão implementada no pseudocódigo do Algoritmo 2.6 é, portanto, a segunda alternativa. Nesta implementação do algoritmo o parâmetro de estabilidade Δ é iniciado com 0,1 e tem seu valor multiplicado por 1,1 a cada duas iterações.

Ainda com relação ao pseudocódigo, o método *aleatorio*(), recebe como parâmetro dois valores inteiros e retorna um valor real, e aleatório, dentro deste intervalo. Já o método *nroCorretos*() retorna o número de exemplos distintos classificados corretamente por W. Este método é usado para que o melhor vetor de pesos encontrado possa ser identificado e salvo em Wpoc. Note que na versão original [Krauth & Mezard 1987] não é mencionado o uso de um vetor de pesos auxiliar (Wpoc), no entanto, como observado nos testes realizados, Wpoc é necessário para evitar que um vetor de pesos bom ou mesmo ótimo seja perdido.

```
Class MinOver
begin
{Entrada: E - conjunto de treinamento com n instâncias da forma:
              (xk0, xk1, xk2,..., xkp, Ck), sendo que 1 ≤ k ≤ n
  MAX - número de iterações, pré-determinado}
{Saída: Wpoc}

method MinOver(E)
  begin
    itMax ← MAX
    Δ ← 0,1 {parâmetro de estabilidade}
    corretosWpoc ← 0
    for i ← 0 to p do
      Wi ← aleratorio(0,1)
    while it < itMax do
      begin
        menor ← (WE1)C1
        idMenor ← 1
        if it mod 2 = 0 then
          menor ← |menor|
        for k ← 2 to n do
          begin
            soma ← (WEk)Ck
            if it mod 2 = 0 then
              soma ← |soma|
            if soma < menor then
              begin
                menor ← soma
                idMenor ← k
              end
          end  {for}
        if menor ≤ Δ then
          W ← W + EidMenorCidMenor
        if it mod 2 = 0 then
          Δ ← 1,1Δ
        it ← it + 1
        corretosW ← nroCorretos()
        if corretosW > corretosWpoc then
          begin
            corretosWpoc ← corretosW
            Wpoc ← W
          end
      end  {while}
  end {method}
end {Class}
```

Algoritmo 2.6 – Pseudocódigo do algoritmo MinOver

2.6.1 Exemplo de Execução do Algoritmo MinOver

Assim como nos outros exemplos de execução, será usado o conjunto que representa o problema paridade-3, apresentado na Tabela 2.3. O exemplo de execução mostrado na Tabela 2.8 apresenta um '*trace*' do algoritmo até a décima iteração. A cada iteração, representada por it, é mostrado o vetor de pesos corrente W, o vetor de pesos auxiliar Wpoc, o menor valor para v^k seguido do índice da instância que o gerou (representado por k), o parâmetro de estabilidade Δ, que neste exemplo é iniciado com 0,1 e tem seu valor incrementado durante a

execução (como mostra o pseudocódigo), e o número de instâncias classificadas corretamente por W.

Tabela 2.8 – Exemplo de execução do algoritmo MinOver usando o conjunto de treinamento dado na Tabela 2.3

it	**W**	**Wpoc**	**Menor v^k**	**k**	**Δ**	**Corretos**
0	⟨0,52 0,04 0,13 0,08⟩	⟨0,52 0,04 0,13 0,08⟩	0,28	8	0,010	4
1	⟨0,52 0,04 0,13 0,08⟩	⟨0,52 0,04 0,13 0,08⟩	–0,69	5	0,011	2
2	⟨–0,48 1,04 –0,86 –0,92⟩	⟨0,52 0,04 0,13 0,08⟩	0,28	8	0,011	2
3	⟨–0,48 1,04 –0,86 –0,92⟩	⟨0,52 0,04 0,13 0,08⟩	–1,6	7	0,012	4
4	⟨0,52 0,04 –1,87 0,08⟩	⟨0,52 0,04 0,13 0,08⟩	1,23	7	0,012	4
5	⟨0,52 0,04 –1,87 0,08⟩	⟨0,52 0,04 0,13 0,08⟩	–2,5	3	0,013	4
6	⟨–0,48 –0,96 –0,87 –0,92⟩	⟨–0,48 –0,96 –0,87 –0,92⟩	0,35	4	0,013	6
7	⟨–0,48 –0,96 –0,87 –0,92⟩	⟨–0,48 –0,96 –0,87 –0,92⟩	–3,23	4	0,014	6
8	⟨0,52 0,04 –1,86 –1,92⟩	⟨–0,48 –0,96 –0,87 –0,92⟩	0,43	7	0,014	4
9	⟨0,52 0,04 –1,86 –1,92⟩	⟨–0,48 –0,96 –0,87 –0,92⟩	–4,3	8	0.016	4
10	⟨–0,48 1,03 –0,86 –0,92⟩	⟨–0,48 –0,96 –0,87 –0,92⟩	0,28	8	0,016	2

Note, na Tabela 2.8, que os valores do menor v^k intercalam-se entre positivos e negativos, e que o mesmo só consegue ser menor que o parâmetro de estabilidade (Δ) quando é negativo. Por essa razão o vetor de pesos só é atualizado quando o menor valor de v^k for negativo. Isto acontece somente no início da iteração, pois como o valor de Δ é incrementado durante a iteração, chegará um momento em que os valores positivos do menor v^k serão menores que Δ, e as instâncias que os geraram também contribuirão na atualização do vetor de pesos.

3 capítulo O Algoritmo BCP – Uma Abordagem Geométrica para o Treinamento de TLUs

3.1 Introdução

O Barycentric Correction Procedure (BCP) [Poulard 1995] é um algoritmo eficiente para o treinamento de uma TLU que não pertence à família Perceptron. O BCP baseia-se em conceitos geométricos e a proposta do algoritmo, para a classificação em duas classes, é calcular iterativamente o baricentro das regiões delimitadas pelos exemplos de treinamento relativos a cada uma das classes. O baricentro ou combinação convexa é definido como a média ponderada dos exemplos de treinamento de uma determinada classe e seu conjunto de coeficientes de pesos.

Além da abordagem geométrica, o algoritmo diferencia-se de outros por calcular o vetor de pesos e o valor do *bias* separadamente. O BCP define o vetor de pesos W como o vetor que conecta dois pontos, b_1 e b_2, tal que b_1 é o baricentro da região convexa[6] (*convex hull*) [de Berg *et al.* 2000] definida pelas instâncias de treinamento de classe positiva e b_2 é o baricentro da região convexa definida pelas instâncias de treinamento de classe negativa. Dessa forma, o cálculo do vetor de pesos W é dado pela equação (3.1).

$$W = b_1 - b_2 \qquad (3.1)$$

A busca por um vetor de pesos W é mais controlada no BCP do que no Perceptron, uma vez que todas as instâncias classificadas incorretamente são levadas em consideração em toda modificação no vetor de pesos, sem desconsiderar as instâncias classificadas corretamente, o que faz com que a modificação de W não seja local.

Nas próximas seções são apresentadas duas variações do algoritmo BCP ([Poulard & Labreche 1995]) que têm objetivos diferentes: o BCPMin e o BCPMax. O BCPMin tem como objetivo encontrar um vetor de pesos que define um hiperplano que separa o maior número de

[6] Região convexa (*convex hull*) de um conjunto de pontos é a menor região convexa que inclui todos os pontos.

instâncias corretamente. Já o BCPMax procura por um hiperplano que separa o maior número possível de instâncias de uma determinada classe do restante do conjunto de treinamento.

A diferença entre o BCPMin e o BCPMax está apenas na forma como o termo *bias* é determinado. Quando, porém, o conjunto de treinamento for linearmente separável, o cálculo do *bias* será o mesmo para ambas as variações, pois neste caso um hiperplano que separa corretamente todos os exemplos do conjunto de treinamento irá satisfazer os objetivos de ambas as variações.

No que segue será apresentada inicialmente a proposta geral do algoritmo BCP, que corresponde à parte comum a ambas as variações, isto é, do cálculo do vetor de pesos W e do possível cálculo do *bias* se o conjunto de treinamento for linearmente separável. Na seqüência são apresentadas as variações BCPMin e BCPMax respectivamente.

3.2 A Proposta Geral do Algoritmo BCP

Sejam W e θ o vetor de pesos e o termo *bias*, respectivamente, aprendidos pelo BCP. Considere o hiperplano H(W,θ) descrito pela equação (3.2).

$$H(W, \theta): \varepsilon WX + \theta = 0 \tag{3.2}$$

na qual $\varepsilon \in \{-1,1\}$ é o parâmetro que será usado para variar entre o hiperplano encontrado e o hiperplano seu ortogonal. A justificativa para o uso deste parâmetro está no fato que, em um determinado momento do treinamento, o hiperplano ortogonal ao hiperplano corrente pode classificar corretamente mais exemplos de treinamento.

Considere o conjunto de treinamento com n instâncias dado por $E = \{(E^i,C^i) \mid i = 1,2,...,n \text{ e } C^i \in \{1,-1\}\}$. Para definição dos baricentros e do vetor de pesos W, o conjunto de treinamento E é separado em dois: um conjunto com todas as instâncias de classe positiva $E1 = \{(E^p,C^p) \text{ e } C^p = 1\}$, e o outro com todas as instâncias de classe negativa $E2 = \{(E^q,C^q) \text{ e } C^q = -1\}$, com p e $q \in \{1,...,n\}$. Como $|E| = n$ e $E = E1 \cup E2$, seja $|E1| = n_1$ e $|E2| = n_2$.

Considere $I_1 = \{1,...,n_1\}$ e $I_2 = \{1,...,n_2\}$ conjuntos de índices dos respectivos conjuntos E1 e E2. Seja b_1 o vetor que define o baricentro do conjunto E1 e b_2 o vetor que define o baricentro do conjunto E2, definidos pelas equações em (3.3).

$$b_1 = \frac{\sum_{i=1}^{n_1} \alpha_i E1^i}{\sum_{i=1}^{n_1} \alpha_i} \quad \text{e} \quad b_2 = \frac{\sum_{i=1}^{n_2} \mu_i E2^i}{\sum_{i=1}^{n_2} \mu_i} \qquad (3.3)$$

nas quais os conjuntos $\alpha = \{\alpha_1,..., \alpha_{n_1}\}$ e $\mu = \{\mu_1,..., \mu_{n_2}\}$ são conjuntos de coeficientes de pesos relativos aos conjuntos E1 e E2 respectivamente. Esses coeficientes têm por objetivo ponderar cada instância de treinamento; quanto maior o peso de uma instância, mais próxima ela estará do baricentro.

Em geral, esses coeficientes poderiam ser inicializados com 1 porém, para alguns conjuntos de treinamento, o algoritmo não teria bons resultados. Um exemplo dessa situação é a de conjuntos de treinamento simétricos, como por exemplo, os conjuntos que representam as funções *xor* e paridade-3. Por essa razão é conveniente inicializar os conjuntos de coeficientes com pesos aleatórios, dentro de um intervalo [1,*a*], com *a* igual a 2 por exemplo (como sugerido em [Pourlard 1995]).

A cada vez que um novo hiperplano é encontrado o algoritmo atualiza os pesos em α e μ. No processo de atualização é adicionado um valor aleatório do intervalo (0,1) às instâncias classificadas incorretamente. Isso faz com que o baricentro dos exemplos incorretos varie numa situação em que os exemplos incorretos permanecem os mesmos durante várias iterações.

A atualização de α e μ, no entanto, não é suficiente para encontrar um vetor de pesos W ótimo. Uma maneira de fazer com que W varie de modo a encontrar um vetor ótimo é atualizá-lo de acordo com o procedimento descrito a seguir.

Suponha uma situação durante o aprendizado, na qual o hiperplano definido pelo vetor de pesos (obtido até o momento) não separa devidamente as instâncias, dando origem aos chamados erros de positivo (instâncias positivas que são classificadas como negativas) e erros de negativo (instâncias negativas que são classificadas como positivas).

Seja b_{e_1} o baricentro da região delimitada pelos exemplos de treinamento de classe positiva que foram classificados incorretamente e b_{e_2} o baricentro da região delimitada pelos exemplos de treinamento de classe negativa que foram classificadas incorretamente. Considere os vetores e_1 e e_2 dados pelas equações (3.4).

$$e_1 = b_{e_1} - b_1 \quad \text{e} \quad e_2 = b_{e_2} - b_2 \qquad (3.4)$$

Sejam os vetores b_1' e b_2',[7] os novos baricentros, como mostram as equações em (3.5) e W' o novo vetor de pesos, definido por (3.6).

$$b_1' = b_1 + re_1 \quad \text{e} \quad b_2' = b_2 + se_2 \tag{3.5}$$

$$W' = W + b_1' - b_2' \tag{3.6}$$

tal que r e $s \in [0,1]$ e são gerados randomicamente. A Figura 3.1 ilustra a interpretação geométrica da atualização de W. Note, na Figura 3.1, que o hiperplano H definido por W, em uma iteração it, classifica incorretamente os exemplos das regiões de cor cinza (ambas tonalidades). Observe como os baricentros b_{e_1} e b_{e_2}, ajudam na determinação dos vetores e_1 e e_2 (equação 3.4) e conseqüentemente na busca de W'. O hiperplano H', definido por W' na iteração it + 1, classifica incorretamente apenas os exemplos na região cinza escuro da Figura 3.1.

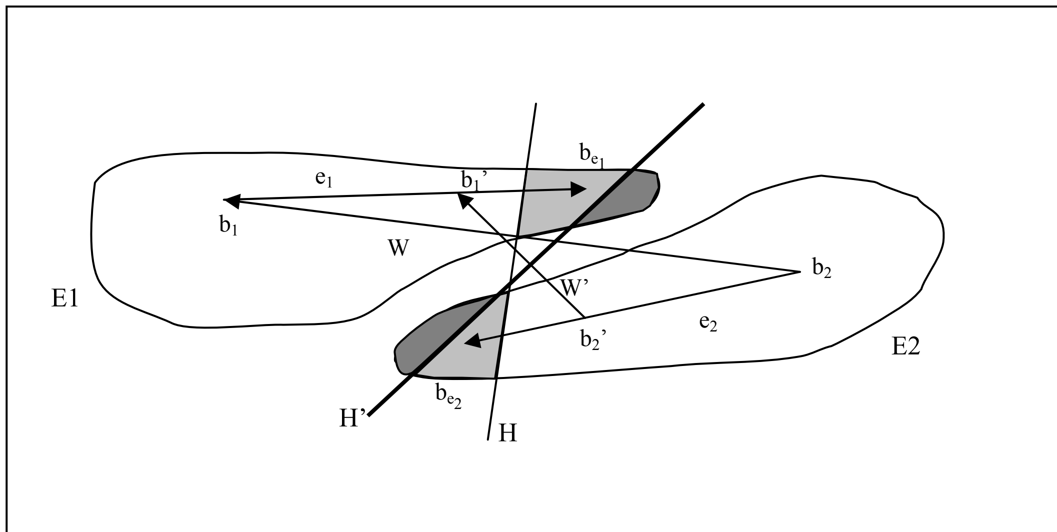


Figura 3.1 – Exemplo de atualização de W por meio do cálculo dos baricentros dos exemplos classificados incorretamente

A atualização do vetor de pesos W, descrita na equação (3.6), pressupõe que a separação das instâncias pelo hiperplano atual provoque o surgimento de erros de positivo e de erros de negativo. A proposta original [Poulard 1995] não deixa claro como deve ser feita a atualização de W no caso de existir apenas um tipo de erro. Uma maneira trivial de adequar a

[7] Considere o apóstrofo como indicador da iteração it + 1.

proposta original ao tratamento de situações em que ocorre apenas um tipo de erro é descrito a seguir.

Considere o caso em que o hiperplano atual provoque erros somente em algumas instâncias positivas (pertencentes a E1). Como nenhuma instância negativa foi classificada incorretamente, b_{e_2} não existirá. Dessa forma, de acordo com as equações referentes ao cálculo de e_1 e b_1 em (3.4) e (3.5) respectivamente, e de acordo com a equação (3.6); o vetor de pesos seria atualizado como: $W' = W + b_1' - b_2$, sendo b_2 o baricentro da classe negativa. Esta abordagem, no entanto, não obteve resultados satisfatórios nos testes realizados.

No que segue é apresentada a abordagem proposta neste trabalho para a atualização de W quando existir apenas um tipo de erro. Na proposta os baricentros b_1' e b_2' são determinados da seguinte maneira: encontra-se o baricentro dos exemplos que foram classificados incorretamente, neste caso só haverá erros em uma das classes. Em seguida, encontra-se o baricentro dos exemplos que estão na área de sobreposição (no conjunto P_{ov}, Seção 3.3) pertencentes à classe que não apresentou erros.

O vetor (e_1 ou e_2) é então calculado, para a classe que não possui erro, como a diferença entre o antigo baricentro e o baricentro dos exemplos desta classe que pertencem a P_{ov}. Note que o vetor (e_1 ou e_2) só é calculado para a classe que não possui erros; o baricentro da outra classe será o baricentro dos exemplos que foram classificados incorretamente.

Suponha a situação que o hiperplano atual cometa somente alguns erros de positivo, ou seja, algumas instâncias positivas são classificadas como negativas. Neste caso encontra-se o baricentro dos exemplos positivos classificados incorretamente, b_{e_1}. Como na situação considerada não acontecem erros de negativos, o próximo passo é encontrar o baricentro dos exemplos negativos presentes na área de sobreposição, denotado por b_{s2}. Neste caso, os baricentros b_1' e b_2' são determinados segundo as equações (3.7) e (3.8).

$$b_1' = b_{e_1} \tag{3.7}$$

$$e_2 = b_{s2} - b_2 \quad \text{e} \quad b_2' = b_2 + s e_2 \tag{3.8}$$

Se, no entanto, os exemplos errados fossem somente pertencentes à classe negativa, os baricentros b_1' e b_2' seriam dados pelas equações (3.9) e (3.10).

$$b_2' = b_{e_2} \tag{3.9}$$

$$e_1 = b_{s1} - b_1 \quad e \quad b_1' = b_1 + re_1 \tag{3.10}$$

O cálculo do vetor de pesos W', dado pela equação (3.6), permanece o mesmo para todos os casos. Depois que um determinado vetor de pesos W foi aprendido é necessário encontrar um *bias* para completar a definição de um hiperplano. Seja p o número de atributos do conjunto de treinamento e n o número de instâncias no conjunto de treinamento, considere a função v: $R^p \rightarrow R$ definida em (3.11).

$$v(E^k) = -WE^k, \ \forall \ k \in \{1,...,n\} \tag{3.11}$$

Considere agora os conjuntos E1 e E2 definidos no começo desta seção tais que E = E1 $\cup$ E2. Seja o conjunto V = $\{v(E^k)|\ E^k \in E \text{ e } 1 \leq k \leq n\}$ e os conjuntos V1 = $\{v(E^p)|\ E^p \in E1 \text{ e } 1 \leq p \leq n_1\}$ e V2 = $\{v(E^q)|\ E^q \in E2 \text{ e } 1 \leq q \leq n_2\}$, desse modo V = V1 $\cup$ V2. Se o conjunto E for linearmente separável, o vetor W aprendido pelo BCP será tal que a desigualdade max(V1) < min(V2) é verificada. Nesse caso, para que o hiperplano H classifique corretamente todo o conjunto de treinamento E, o termo *bias* deve ser escolhido dentro do intervalo max(V1) < θ < min(V2). Uma boa escolha para o *bias* é dada pela equação (3.12).

$$\theta = \frac{\max(V1) + \min(V2)}{2} \tag{3.12}$$

No entanto, se a desigualdade max(V1) < min(V2) não for verificada, então o conjunto de treinamento E não é linearmente separável e, dessa forma, o termo *bias* deve ser encontrado por alguma das versões, BCPMin (Seção 3.4) ou BCPMax (Seção 3.5), de acordo com o objetivo desejado.

3.3 Considerações Preliminares sobre as Versões BCPMin e BCPMax

Antes da apresentação de cada uma das versões, entretanto, esta seção discute duas situações (identificadas por (A) e (B)) que ocorrem quando do uso de qualquer uma das versões.

SITUAÇÃO (A) - Considere o conjunto dos extremos de V1 e de V2, que é dado por CE = {min(V1), max(V1), min(V2), max(V2)}. Considere uma renomeação dos elementos de CE como ext_1, ext_2, ext_3 e ext_4 feita de tal forma que a seguinte desigualdade se verifique: $ext_1 \leq ext_2 \leq ext_3 \leq ext_4$ (ou seja, ext_1 será o menor elemento de CE e ext_4 o maior).

Como esses valores representam valores associados a um conjunto não linearmente separável, a desigualdade max(V1) ≥ min(V2) é verificada e as configurações apresentadas em (3.13) podem acontecer.

1. $\min(V1) \leq \min(V2) \leq \max(V1) \leq \max(V2)$
   $t_- = 1, t_+ = 0$
2. $\min(V1) \leq \min(V2) \leq \max(V2) \leq \max(V1)$
   $t_- = 1, t_+ = 1$
3. $\min(V2) \leq \min(V1) \leq \max(V1) \leq \max(V2)$
   $t_- = 0, t_+ = 0$
4. $\min(V2) \leq \min(V1) \leq \max(V2) \leq \max(V1)$
   $t_- = 0, t_+ = 1$

(3.13)

Agora considere os conjuntos P_-, P_+ e P_{ov} definidos em (3.14).

$$\begin{aligned} P_- &= [ext_1, ext_2) \cap V \\ P_+ &= (ext_3, ext_4] \cap V \\ P_{ov} &= [ext_2, ext_3] \cap V \end{aligned} \qquad (3.14)$$

O conjunto P_- possui n_- instâncias de mesma classe notada por t_-, analogamente P_+ possui n_+ instâncias com a mesma classe identificada por t_+ e diferente de t_-. O conjunto P_{ov} possui n_{ov} instâncias pertencentes a qualquer uma das duas classes.

Os conjuntos P_- e P_+ são chamados áreas de exclusão, pois em cada conjunto há somente instâncias pertencentes a uma das duas classes. O conjunto P_{ov} é chamado de área de sobreposição, pois neste conjunto estão presentes instâncias pertencentes a ambas as classes.

SITUAÇÃO (B) - Nas duas versões do BCP existe a possibilidade do algoritmo encontrar uma situação de impasse, por exemplo: no BCPMin, quando o número de erros cometidos pelo hiperplano atual for igual ao número de erros cometidos pelo hiperplano anterior, o algoritmo precisará de um outro parâmetro para decidir qual o melhor. Já no

BCPMax um caso de impasse ocorrerá quando as cardinalidades dos conjuntos P_- e P_+ forem iguais.

Nesses casos, um parâmetro que justifique a escolha de uma determinada solução para o impasse deve ser introduzido. Considere o parâmetro *gap* (notado como G) representando uma medida de distância entre a instância E^k e o hiperplano H. Quanto maior o *gap*, menor será a sensibilidade à ruídos adicionados ao conjunto de treinamento e melhor será a capacidade de generalização da solução. Como o *gap* é definido de forma distinta para os algortimos BCPMin e BCPMax, este será definido em suas respectivas Seções.

3.4 BCPMin – Minimizando o Número de Instâncias Classificadas Incorretamente

A idéia deste método é calcular o número de erros referentes a possíveis valores para o *bias*, apenas na área que os conjuntos E1 e E2 intercalam-se (i.e. a área de sobreposição entre as duas classes). Esse método é justificado pelo fato de que o número de erros é uma função monotônica nas áreas de exclusão [Poulard & Labreche 1995].

Considere o conjunto Vs, dado pela equação (3.15), obtido por meior da união entre o conjunto P_{ov} (equação (3.14)) e os elementos adjacentes à área de sobreposição (ou seja, $\max(P_-)$ e $\min(P_+)$).

$$Vs = P_{ov} \cup \{\max(P_-), \min(P_+)\} \qquad (3.15)$$

A proposta deste método consite em fazer uma iteração com os valores do conjunto Vs (em ordem crescente) e sucessivamente calcular os erros cometidos pelo *bias* corrente. O termo *bias* é calculado a cada iteração como a média entre dois elementos adjacentes em Vs.

A Figura 3.2 exprime a relação entre a variação do termo *bias* e a classificação das instâncias de treinamento. Considere o conjunto V (dado em (3.11)) ordenado de maneira crescente e representado na Figura 3.2 pelo segmento que contém os extremos ext_1, ext_2, ext_3 e ext_4. Toda instância E^i, para $1 \leq i \leq n$, cujo valor $v(E^i)$ for menor que o *bias* corrente, será classificada como positiva. Com base na equação (3.11) e no fato do conjunto V estar ordenado, pode-se verificar facilmente a desigualdade: $WE^i + \theta > 0$. De maneira análoga verifica-se que as instâncias cujo valor de V é maior que o *bias* são classificadas como negativas.

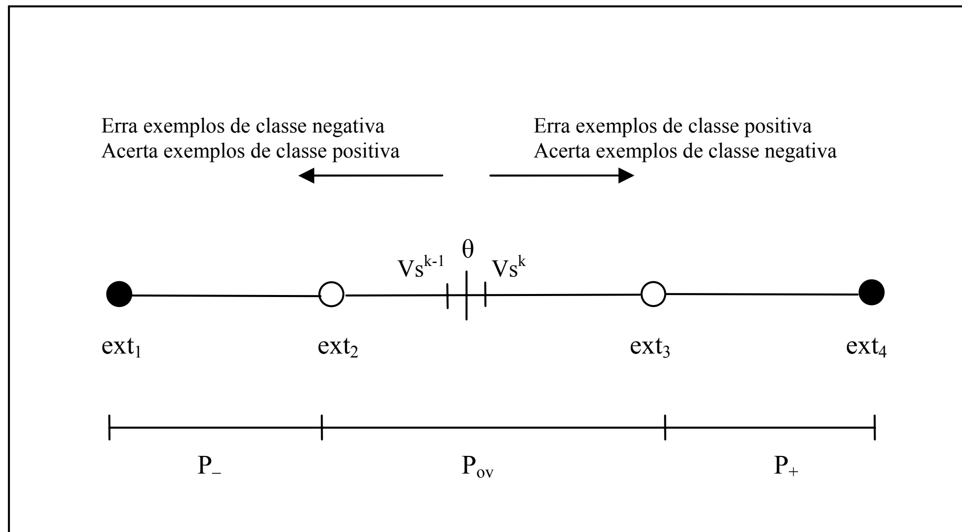


Figura 3.2 – Representação da relação entre o valor do *bias* e suas classificações

Como a iteração é realizada somente no conjunto Vs, o número de erros (ne) associado ao primeiro valor do *bias* deve ser calculado considerando as instâncias pertencentes a P_{-} e P_{+}. Lembrando que o termo t_{-} indica a classe de P_{-}, considere as situações (A) e (B).

(A) $t_{-} = 0$, de acordo com as situações 3 e 4 de (3.13) segue que a classe das instâncias em P_{-} é negativa. Dessa forma o número de erros (ne) será o número de instâncias positivas mais o número de instâncias em P_{-}, ou seja, $ne = n1 + n_{-}$.

(B) $t_{-} = 1$, de acordo com as situações 1 e 2 de (3.13) segue que a classe das instâncias em P_{-} é positiva. Dessa forma o número de erros (ne) será o número de instâncias positivas menos o número de instâncias em P_{-}, ou seja, $ne = n1 - n_{-}$.

A partir da situação inicial, citada anteriormente, o cálculo do termo *bias* prossegue seqüencialmente para todo par de valores em Vs. Como se pode notar na Figura 3.2, toda instância de classe negativa, cujo valor em V é menor que o valor do *bias*, é classificada incorretamente pelo *bias* atual. Dessa forma, se a classe associada à instância que gerou Vs^{k-1} (ver Figura 3.2) for negativa, o *bias* atual irá classificá-la incorretamente, então ne é incrementado de 1. De maneira análoga, se a classe associada à instância que gerou Vs^{k-1} for positiva, o *bias* irá classificá-la corretamente, e ne é decrementado de 1.

Cada vez que o algoritmo percorre os elementos em Vs, o melhor *bias* encontrado é armazenado em θc. O melhor *bias* é aquele que, junto com o vetor de pesos corrente, define

um hiperplano que classifica corretamente mais instâncias de treinamento ou, no caso de empate, o que possuir o maior *gap*, definido pela equação (3.16).

$$G = \frac{Vs^{k-1} - Vs^{k}}{\|W\|} \tag{3.16}$$

Se o hiperplano definido pelo vetor de pesos atual W e θc classificar corretamente mais instâncias de treinamento que o último hiperplano salvo, então W é armazenado em Wpoc e θc é armazenado em θpoc. O pseudocódigo do algoritmo BCPMin é apresentado nos Algoritmos 3.1, 3.2 e 3.3.

No Algoritmo 3.1, após o conjunto E ser separado em dois (E1 e E2) com relação às classes das instâncias de treinamento (E1 contendo instâncias de classe positiva e E2 instâncias de classe negativa), os métodos *geraPesos*() e *baricentro*() são chamados. O método *geraPesos*() recebe como parâmetro o número de instâncias que determinada classe possui e retorna um vetor, de dimensão dada pelo número passado como parâmetro, com pesos de valores aleatórios no intervalo [1,2].

Já o método *baricentro*() que está descrito em Algoritmo 3.3, recebe como parâmetros um conjunto de treinamento e um vetor de coeficientes de pesos e retorna o vetor baricentro referente a este conjunto. Os métodos *baricentroErrado*() e *baricentroPov*() não estão descritos no pseudocódigo porém, são bastante semelhantes ao método *baricentro*(), apresentado em Algoritmo 3.3. A principal diferença entre eles é o conjunto de instâncias sobre o qual o baricentro será obtido.

No caso do método *baricentroErrado*() os parâmetros que este recebe são os mesmos que no método *baricentro*(), porém o baricentro é calculado usando apenas as instâncias pertencentes a este conjunto que apresentaram erro durante a iteração anterior. De maneira análoga o método *baricentroPov*() recebe os mesmos parâmetros, porém usa apenas as instâncias que pertencem ao conjunto passado como parâmetro e também ao conjunto P_{ov}.

O método *baricentro*() é chamado apenas duas vezes durante toda execução, uma para encontrar o baricentro da classe positiva e a outra para encontrar o baricentro da classe negativa. Note, no entanto, que esses baricentros serão usados durante toda a execução do algoritmo, sempre que o vetor de pesos W for atualizado.

```
Class BCPMin
begin
{Entradas: E - conjunto de treinamento com n instâncias da forma: (x^k_1, x^k_2,..., x^k_p, C^k), sendo que 1 ≤ k ≤ n
  MAX - número máximo de iterações, pré-determinado}
{Saída: Wpoc e biasPoc}
tMenos ← 0; nPMenos   ← 0

method BCPMin(Matrix E)
  begin
    exCorretos ← 0
    it ← 0; itMax ← MAX
    a ← 0 ; b ← 0
    nc ← 0; gapPoc ← 0
    for i ← 1 to n do
      begin
        if C^i > 0 then
          begin
            a ← a + 1
            E1^a ← E^i
          end
        else if C^i ≤ 0 then
            begin
              b ← b + 1
              E2^b ← E^i
           end
      end
    alfa ← geraPesos(a)
    mi ← geraPesos(b)
    b1 ← baricentro(alfa, E1)
    b2 ← baricentro(mi, E2)
    W ← b_1 – b_2
    while it < itMax do
      begin
        if it > 1 then
          begin
            r ← aleatorio(0,1)           {r ∈ [0,1] e s ∈ [0,1]}
            s ← aleatorio(0,1)
            if (existeErro(E1) and existeErro(E2)) then
              begin
                be1 ← baricentroErrado(alfa, E1)
                be2 ← baricentroErrado(mi, E2)
                e1 ← be1 – b1
                e2 ← be2 – b2
                W ← W + (b1 + re1) – (b2 + se2)
              end
            else if existeErro(E1) then
                begin
                  be1 ← baricentroErrado(alfa, E1)
                  bs2 ← baricentroPov(mi, E2)
                  e2 ← bs2 – b2
                  W ← W + be1 – (b2 + se2)
               end
            else if existeErro(E2) then
                begin
                  bs1 ← baricentroPov(alfa, E1)
                  be2 ← baricentroErrado(mi, E2)
                  e1 ← bs1 – b1
                  W ← W + (b1 + re1) – be2
                end
          end   {if it > 1}
```

Algoritmo 3.1 – Pseudocódigo do algoritmo BCPMin

```
        for i ← 1 to n1 do   {n1 - número de instâncias em E1}
          V1^i ← –WE1^i
        for i ← 1 to n2 do   {n2 - número de instancias em E2}
          V2^i ← –WE2^i
        minV2 ← menor(V2)
        maxV1 ← maior(V1)
        if (maxV1 < minV2) then
          biasPoc ← (maxV1 + minV2) / 2
        else                                   {inicio else}
          begin
            Vs ← determinaVs(V1,V2)
            if tMenos = 1 then    {tMenos é atualizaddo em determinaVs()}
              ne ← n1 –  nPMenos   { nPMenos é o número de instâncias em P_ }
            else
              ne ← n1 + nPMenos
            if(n – ne ≥ ne) then
              begin
                nc ← ne; biasC ← bias;  gapC ← gap; eps ← 1
              end
            else
              begin
                nc ← n – ne;  biasC ← –bias; gapC ← gap; eps ←  –1
              end
            nVs ← |Vs|
            for i ← 3 to nVs do  {nVs é o número de instâncias em Vs}
              begin
                for j ←1 to n1 do
                  if Vs^(i – 1) = V1^j then
                    ne ← ne – 1
                for k ←1 to n2 do
                  if Vs^(i – 1) = V2^k  then
                ne ← ne +1
                bias ← (Vs^i + Vs^(i – 1)) / 2
                gap ← (Vs^i – Vs^(i – 1)) / modulo(W)

                if((ne < nc) or (ne = nc and gap > gapC) then
                  begin
                    nc ← ne; biasC ← bias; gapC ← gap; eps ← 1
                  end
                if((n – ne < nc) or (n – ne = nc and gap > gapC) then
                  begin
                    nc ← n – ne; biasC ← –bias; gapC ← gap; eps ← –1
                  end
              end                {for}
            nroCorretos ← nroCorretos()
            if(nroCorretos > exCorretos or (nroCorretos = exCorretos and gapC > gapPoc)) then
              begin
                exCorretos ← nroCorretos
                gapPoc ← gapC
                biasPoc ← biasC
                if eps = –1 then
                  WPoc ← –W
                else
                  WPoc ← W
              end
          end                                      {fim else}
        atualizaPesos(W, biasC, eps)
        it ← it +1
      end                              {fim while  it < itMax}
  end   {fim method }
```

Algoritmo 3.2 – Pseudocódigo do algoritmo BCPMin (Continuação)

```
method Vector baricentro(Vector V, Matriz M)
  begin
    somab ← 0
    b ← ⟨0,...,0⟩; auxb ← ⟨0,...,0⟩  {b e auxb são vetores de dimensão p}
    for i ← 1 to nVec do  {nVec - número de elementos no vetor V}
      for j ← 1 to p do   {p - número de atributos do conjunto de treinamento}
        auxb[j] ← auxb[j] + V[i]M[i][j]
    for j ← 1 to nVec do
      somab ← somab + V[j]
    for i ← 1 to p do
      b[i] ← auxb[i] / somab
    return b
  end   {fim method}

method Vector determinaVs(vector V1, vector V2)
  begin
    minV1 ← menor(V1);  minV2 ← menor(V2); maxV1 ← maior(V1);  maxV2 ← maior(V2)
    if(minV1 ≤ minV2 and minV2 ≤ maxV1 and maxV1 ≤ maxV2) then
      begin
        ext1 ←  minV1; ext2 ← minV2; ext3 ←  maxV1; ext4 ← maxV2; tMenos ← 1
      end
    else if(minV1 ≤ minV2 and minV2 ≤ maxV2 and maxV2 ≤ maxV1) then
        begin
          ext1 ←  minV1; ext2 ← minV2; ext3 ←  maxV2; ext4 ← maxV1; tMenos ← 1
        end
    else if(minV2 ≤ minV1 and minV1 ≤ maxV1 and maxV1 ≤ maxV2) then
        begin
          ext1 ←  minV2; ext2 ← minV1; ext3 ←  maxV1; ext4 ← maxV2; tMenos ← 0
        end
    else if(minV2 ≤ minV1 and minV1 ≤ maxV2 and maxV2 ≤ maxV1) then
        begin
          ext1 ←  minV2; ext2 ← minV1; ext3 ←  maxV2; ext4 ← maxV1; tMenos ← 0
        end
    for j ← 1 to n do
      V^j ← –WE^j

    for i ← 1 to n do
      begin
        if V^i < ext2 then
          P_- ← V^i
        else if V^i > ext3 then
          P_+ ← V^i
        else
          P_ov ← V^i
      end

    Vs ← maior(P_-)
    Vs ← menor(P_+)
    Vs ← P_ov
    Vs ← ordena(Vs)
    return Vs
  end  {fim method}
end {fim Class}
```

Algoritmo 3.3 – Pseudocódigo do algoritmo BCPMin (Continuação)

A cada iteração o método *existeErro*() é chamado, este método recebe como parâmetro um conjunto de treinamento e retorna verdadeiro se neste conjunto existir erro e falso caso contrário. Este método é usado para saber em qual classe o último hiperplano obtido comete erro e assim determinar como o vetor de pesos será atualizado. O método *aleatorio*(), em Algoritmo 3.1, retorna um valor real e aleatório dentro do intervalo passado como parâmetro.

O método *modulo*(), em Algoritmo 3.2, retorna o módulo do vetor de pesos passado como parâmetro. Os métodos *maior*() e *menor*(), em Algoritmos 3.2 e 3.3, retornam o maior e o menor elemento do conjunto passado como parâmetro, respectivamente. O método *ordena*(), retorna o conjunto que foi passado como parâmetro ordenado de forma crescente. E o método *nroCorretos*() retorna o número de instâncias que foram classificadas corretamente pelo hiperplano atual.

O método *determinaVs*() recebe como parâmetros os conjuntos V1 e V2, e além de criar os conjuntos P_+, P_- e P_{ov}, retorna o conjunto Vs já ordenado. O método *atualizaPesos*() incrementa, de um valor aleatório do intervalo [0,1], os pesos referentes às instâncias que foram classificadas incorretamente durante a última iteração, dos vetores de coeficientes de pesos alfa[] e mi[].

O teorema de convergência para conjuntos linearmente separáveis e a descrição de um modelo determinístico para minimizar os erros em um conjunto de treinamento não linearmente separável pode ser encontrado em [Poulard & Estèves 1995].

3.4.1 Aprendizado usando o BCPMin em um Conjunto de Treinamento Linearmente Separável

Esta seção apresenta o '*trace'* do BCPMin quando do aprendizado utilizando um conjunto de treinamento linearmente separável, no caso o conjunto da Tabela 3.1.

Tabela 3.1 – Conjunto Linearmente Separável com entradas reais

Exemplos E^k (k = 1,..., 12)	**X_1**	**X_2**	**Classe C^k (k = 1,..., 12)**	**Exemplos E^k (k = 13,..., 23)**	**X_1**	**X_2**	**Classe C^k (k = 13,..., 23)**
E^1	1	7	1	E^{13}	9	2	–1
E^2	2	6	1	E^{14}	13	4	–1
E^3	2	9	1	E^{15}	16	3	–1
E^4	3	4	1	E^{16}	16	6	–1
E^5	3	7	1	E^{17}	18	4	–1
E^6	4	6	1	E^{18}	18	6	–1
E^7	4	9	1	E^{19}	19	3	–1
E^8	5	4	1	E^{20}	19	7	–1
E^9	6	6	1	E^{21}	20	5	–1
E^{10}	9	8	1	E^{22}	21	4	–1
E^{11}	12	7	1	E^{23}	22	7	–1
E^{12}	18	8	1				

O primeiro passo do algoritmo é encontrar os baricentros das instâncias positivas e negativas, notados aqui por b_1 e b_2 respectivamente. Na primeira iteração (it = 1) o vetor de pesos W é definido como a diferença dos baricentros b_1 e b_2.

Sejam os baricentros encontrados na primeira iteração $b_1 = \langle 5,79\ 6,79 \rangle$ e $b_2 = \langle 17,44\ 4,52 \rangle$. Note que esses baricentros foram calculados com o uso dos coeficientes de pesos dos vetores α e μ, sem o uso desses vetores os baricentros seriam $b_1 = \langle 5,75\ 6,75 \rangle$ e $b_2 = \langle 17,36\ 4,63 \rangle$ respectivamente.

Como mostrado na equação (3.1) a primeira atualização do vetor de pesos é a diferença entre os baricentros, desse modo $W = \langle -11,65\ 2,27 \rangle$. Encontrado o vetor de pesos W, resta encontrar o valor do *bias* para definir o hiperplano $H(W,\theta)$. O próximo passo é encontrar os conjuntos V1 e V2, como definido na equação (3.11). A Tabela 3.2 apresenta os dois conjuntos (V1 e V2) obtidos na primeira iteração.

Tabela 3.2 – Conjuntos V1 e V2 obtidos na primeira iteração do BCPMin. O mínimo e o máximo de cada conjunto estão assinalados com '*' e '**' respectivamente

Exemplos E^k (k = 1,..., 12)	**V1**	**Exemplos E^k (k = 13,..., 23)**	**V2**
E^1	–4,24*	E^{13}	100,31*
E^2	9,68	E^{14}	142,37
E^3	2,87	E^{15}	179,59
E^4	25,87	E^{16}	172,78
E^5	19,06	E^{17}	200,62
E^6	32,98	E^{18}	196,08
E^7	26,17	E^{19}	214,54
E^8	49,17	E^{20}	205,46
E^9	56,28	E^{21}	221,65
E^{10}	86,69	E^{22}	235,57
E^{11}	123,91	E^{23}	240,41**
E^{12}	191,54**		

Como mencionado anteriormente se o conjunto de treinamento for linearmente separável e o vetor de pesos for tal que o hiperplano definido por ele separa corretamente todas as instâncias, o maior elemento de V1 deverá ser menor que o menor elemento de V2. Como pode ser verificado na Tabela 3.2, este fato não acontece nesta iteração, pois o vetor de pesos não está corretamente ajustado para encontrar o hiperplano separador.

O próximo passo é encontrar os extremos dos conjuntos V1 e V2 e determinar os valores para ext_1, ext_2, ext_3 e ext_4. Neste exemplo tem-se $\min(V1) = -4,24$; $\max(V1) = 191,54$; $\min(V2) = 100,31$; $\max(V2) = 240,41$. Assim: $ext_1 = -4,24$; $ext_2 = 100,23$; $ext_3 = 191,54$; $ext_4 = 240,41$. Esta configuração permite verificar as seguintes desigualdades: $\min(V1) \leq \min(V2) \leq \max(V1) \leq \max(V2)$. Dessa forma, de acordo com (3.13), segue que $t_+ = 0$ e $t_- = 1$.

Uma vez definidos os extremos, são determinados os conjuntos P_-, P_+ e P_{ov}, como mostrado nas equações em (3.14). Sejam os conjuntos $P_- = \{-4,24; 9,28; 2,87; 25,87; 19,06;$

32,98; 26,17; 49,17; 56,28; 86,69}, P_+ = {200,62; 196,08; 214,54; 205,46; 221,65; 235,57; 240,41} e P_{ov} = {123,91; 191,54; 100,31; 142,37; 179,59; 172,78}.

O próximo passo do algoritmo é determinar o conjunto Vs segundo a equação (3.15). Neste exemplo, o conjunto Vs será $P_{ov} \cup$ {86,96; 196,08}, portanto o conjunto ordenado dos valores de Vs, onde deverá ser realizada a iteração para encontrar os possíveis valores para o *bias* é: Vs = {86,69; 100,31; 123,91; 142,37; 172,78; 179,59; 191,54; 196,08}.

Como $t_– = 1$, sabe-se que $P_–$ possui instâncias positivas, então o primeiro valor para o *bias* só errará as instâncias que não estão em $P_–$ (ver Figura 3.2), dessa forma ne = n1 – $n_–$. Se, no entanto, o valor de $t_–$ fosse 0, $P_–$ iria conter instâncias negativas, e, portanto o primeiro *bias* erraria todos os exemplos em $P_–$ mais os exemplos positivos, pois todas as instâncias de classe positiva possuiria valor em V maior que o *bias* atual, assim ne seria ne = n1 + $n_–$.

Depois de encontrado o número de erros ne, o algoritmo começa a iteração em Vs para encontrar o melhor *bias*. Nesta iteração o *bias* que erra menos instâncias (duas) está entre os valores 123,91 e 142,37, o melhor valor para o *bias* é calculado, então, como a média entre esses valores, dessa forma θ = 133,14, assim a equação do hiperplano H é: $H(W,\theta) = -11,65X + 2,27Y + 133,14$. A Figura 3.3 resume graficamente a primeira iteração (it =1).

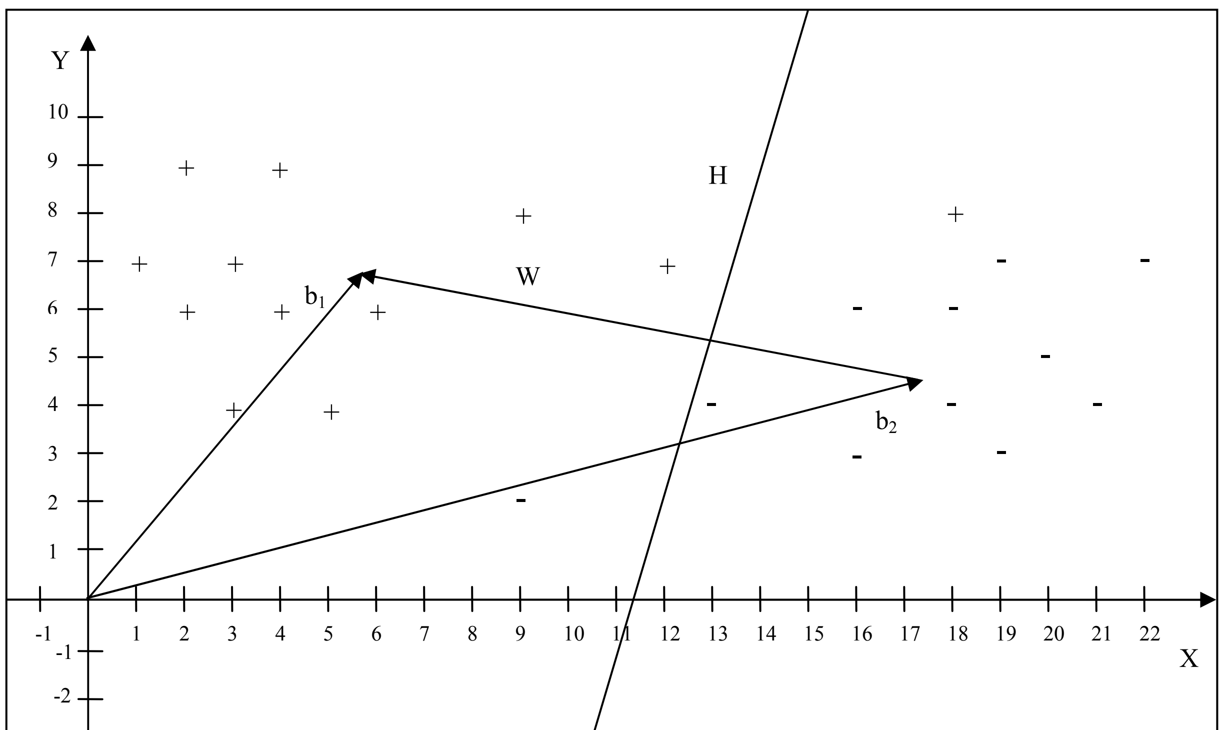


Figura 3.3 – Primeira iteração do algoritmo BCPMin sobre o conjunto da Tabela 3.1

Na segunda iteração (it = 2), o primeiro passo do algoritmo é verificar onde o hiperplano anterior H cometeu erro. Neste exemplo, o hiperplano H comete dois erros, um erro de positivo (instância E^{12}) e um erro de negativo (instância E^{13}).

Como existe apenas um erro em cada classe, o baricentro dos exemplos errados de cada classe é a própria instância, assim o baricentro dos exemplos positivos errados é $b_{e_1} = \langle 18\ 8 \rangle$ e o baricentro dos exemplos negativos errados é $b_{e_2} = \langle 9\ 2 \rangle$.

Uma vez calculados os baricentros dos exemplos errados, os vetores e_1 e e_2 são obtidos segundo a equação (3.4), neste exemplo $e_1 = \langle 12,21\ 1,21 \rangle$ e $e_2 = \langle -8,44\ -2,52 \rangle$. Para que os novos baricentros, b_1' e b_2', possam ser determinados, é necessário multiplicar os vetores e_1 e e_0 por escalares (*r* e *s*) de valores aleatórios no intervalo [0,1]. Sejam $r = 0,6$ e $s = 0,74$ os valores aleatórios obtidos nesta iteração. A Figura 3.4 ilustra os passos da segunda iteração citados até aqui.

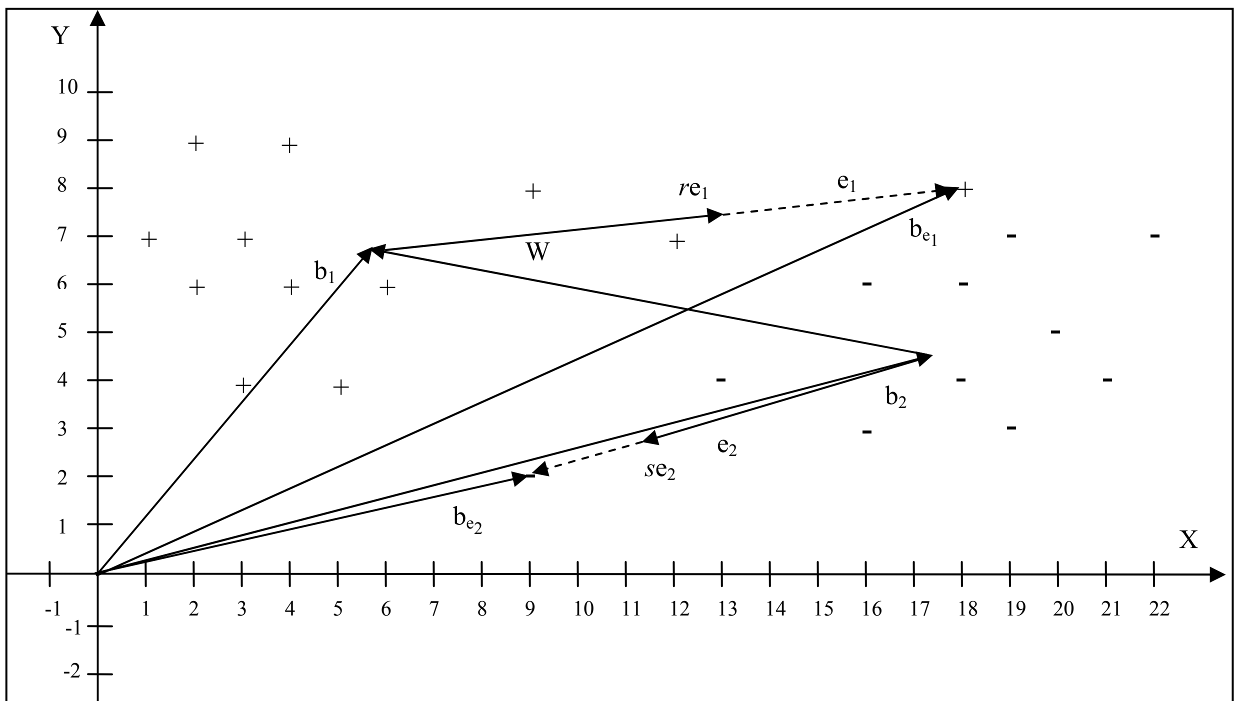


Figura 3.4 – Obtendo os novos baricentros b1' e b2'

O próximo passo é calular os novos baricentros, b_1' e b_2', de acordo com as equações em (3.5), assim $b_1' = \langle 13,1\ 7,52 \rangle$ e $b_2' = \langle 11,19\ 2,65 \rangle$. Uma vez calculados b_1' e b_2', o algoritmo calcula o novo vetor de pesos W', como mostrado na equação (3.6). Para exemplificar o cálculo de W' considere o vetor auxiliar A sendo o vetor obtido da diferença entre os novos baricentros, desse modo $A = b_1' - b_2'$, portanto $A = \langle 1,91\ 4,87 \rangle$. Agora basta

somar o vetor auxiliar A ao antigo vetor de pesos (W = ⟨–11,65 2,27⟩) para obter o novo vetor de pesos W'; portando W' = ⟨–9,74 7,14⟩.

Uma vez achado W' o algoritmo recalcula os conjuntos V1 e V2 e verifica se o conjunto é linearmente separável. Se não for o processo descrito anteriormente se repete; primeiro são calculados os extremos, depois os conjuntos P_+, P_- e P_{ov} e em seguida o conjunto Vs para a determinação do novo termo *bias*. Seja θ' = 69,79 o termo *bias* obtido nesta iteração (it = 2), portanto a equação do hiperplano H' é: H' (W', θ') = –9,74X + 7,14Y + 69,79. A Figura 3.5 demonstra o cálculo realizado para encontrar o vetor W', note que alguns vetores envolvidos foram omitidos da representação para facilitar a visualização.

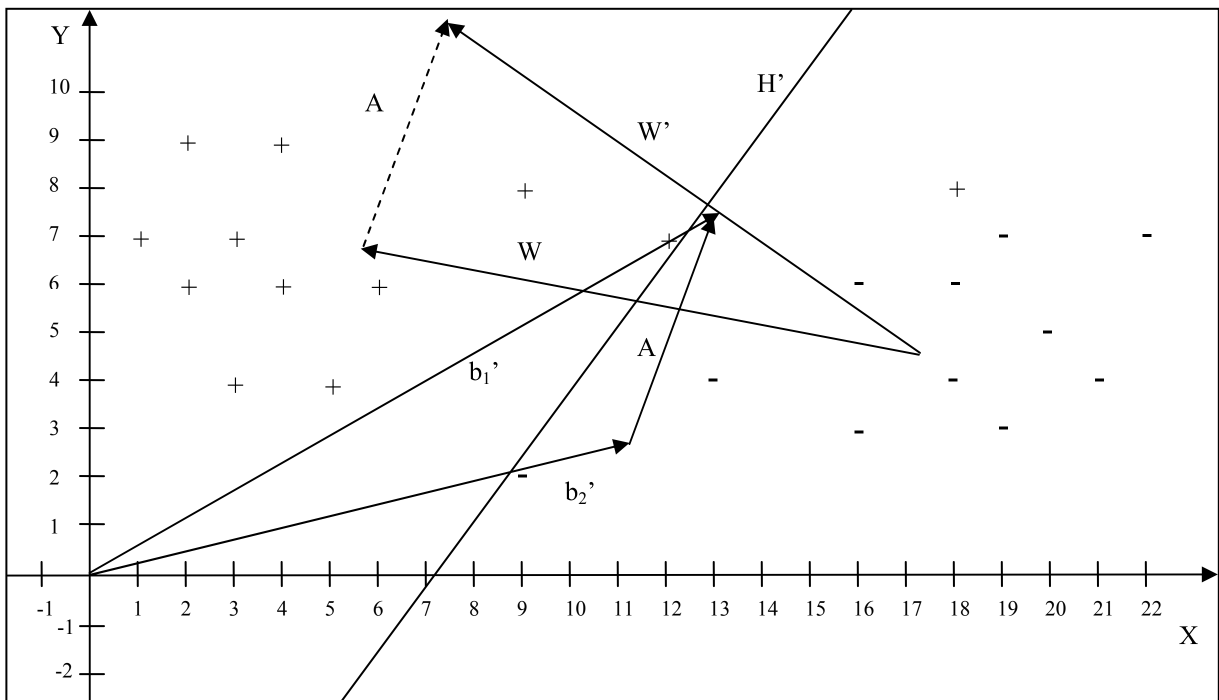


Figura 3.5 – Cálculo do vetor W', segunda iteração do BCPMin

Como pode ser notado na Figura 3.5, o hiperplano H', encontrado na segunda iteração, ainda comete um erro de positivo (instância E^{12}), desse modo o algoritmo atualiza os pesos dos exemplos errados, nesse caso somente desta instância e começa uma nova iteração.

A terceira iteração começa verificando onde o hiperplano anterior cometeu erros, no caso, o hiperplano obtido anteriormente H', comete somente um erro na classe positiva. Dessa forma essa instância é o baricentro dos exemplos positivos classificados incorretamente, assim b_{e_1} = ⟨18 8⟩.

Como não existem erros na classe negativa, o baricentro b_{s2} deverá ser obtido com as instâncias negativas que estão na área de sobreposição, i.e. no conjunto P_{ov} obtido na iteração

anterior. Seja I = {E^{12}, E^{13}, E^{14}, E^{16}}, o conjunto das instâncias cujo valor no conjunto V (ver equação (3.11)) pertence ao conjunto P_{ov} obtido na iteração anterior. Dentre as instâncias pertencentes ao conjunto I, as instâncias de classe negativas são E^{13}, E^{14} e E^{16}; o baricentro dessas instâncias é $b_{s2} = \langle 13{,}27\ 4{,}38 \rangle$.

O próximo passo é calcular e_2, de acordo com a equação em (3.8) (note que e_1 não existirá neste caso), assim $e_2 = \langle -4{,}17\ -0{,}14 \rangle$. Considerando s = 0,95 para esta iteração, o novo baricentro b_2'', dado por $b_2'' = b_2 + se_2$, será $b_2'' = \langle 13{,}47\ 4{,}39 \rangle$.

De acordo com a equação (3.7), o vetor $b_1'' = b_{e_1}$, assim, considerando novamente o vetor auxiliar $A = b_1'' - b_2''$, tem-se $A = \langle 4{,}53\ 3{,}61 \rangle$, e como $W'' = W' + A$, segue que $W'' = \langle -5{,}21\ 10{,}75 \rangle$. Com este vetor de pesos o algoritmo encontra o *bias* θ'' = 12,71, que define a equação para o hiperplano: $H''\ (W'', \theta'') = -5{,}21X + 10{,}75Y + 12{,}71$. Este hiperplano, por fim, classifica todas as instâncias corretamente. A Figura 3.6 ilustra a terceira e última iteração do algoritmo BCPMin.

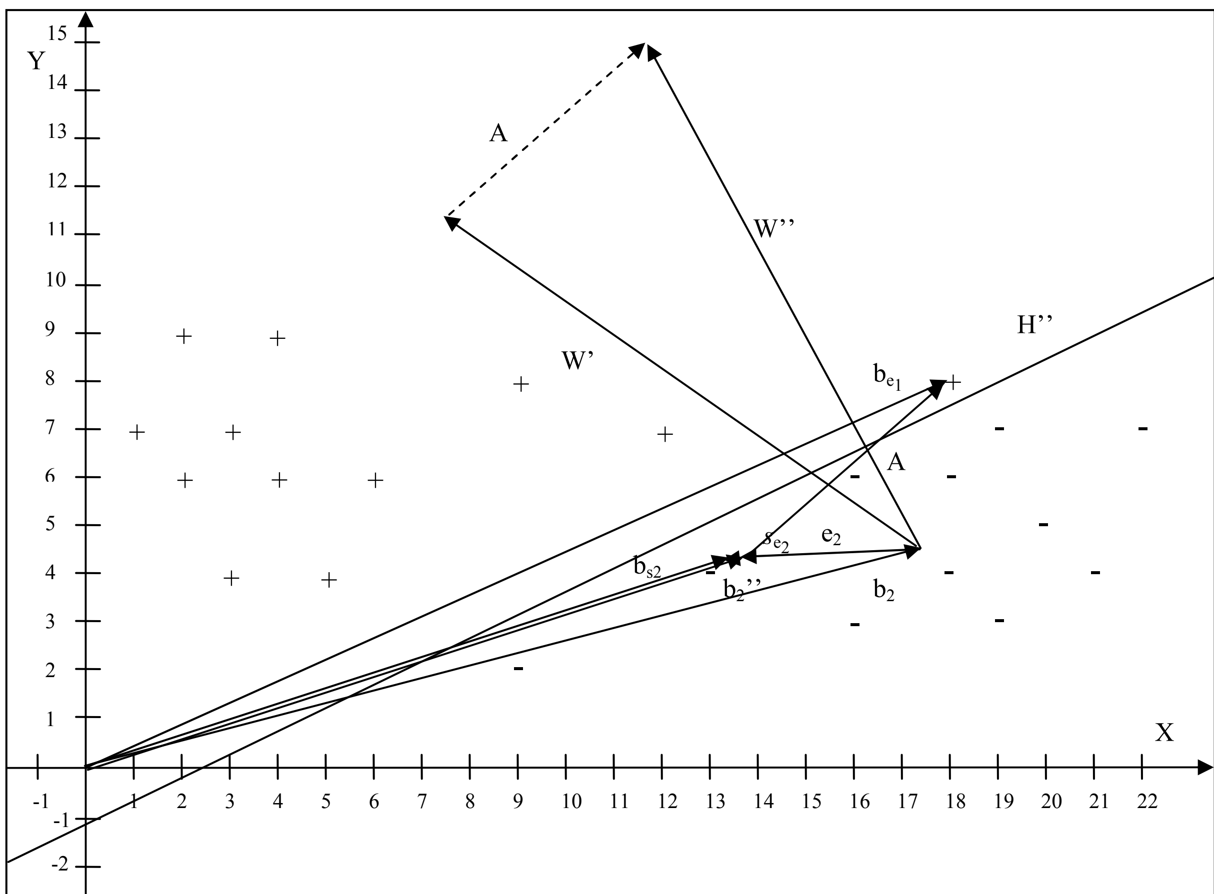


Figura 3.6 – Terceira iteração do BCPMin, separando linearmente o conjunto

3.4.2 Aprendizado usando o BCPMin em um Conjunto de Treinamento Não Linearmente Separável

Neste exemplo de execução será usado o conjunto não linearmente separável da Tabela 3.3, este conjunto também será usado para exemplificar a execução do BCPMax, na próxima seção, a fim de evidenciar os diferentes resultados obtidos pelas duas variações.

Tabela 3.3 – Conjunto não linearmente separável com instâncias reais

Exemplos E^k (k = 1,..., 11)	**X_1**	**X_2**	**Classe C^k (k = 1,..., 11)**	**Exemplos E^k (k = 12,..., 21)**	**X_1**	**X_2**	**Classe C^k (k = 12,..., 21)**
E^1	–3	3	1	E^{12}	–1	–3	1
E^2	1	2	1	E^{13}	–2	–1	1
E^3	2	2	–1	E^{14}	–3	–2	1
E^4	3	1	1	E^{15}	–2	2	–1
E^5	3	2	1	E^{16}	–2	4	–1
E^6	3	4	1	E^{17}	–3	2	–1
E^7	2	5	1	E^{18}	–4	1	–1
E^8	1	–1	1	E^{19}	–4	5	–1
E^9	2	–1	–1	E^{20}	–5	3	–1
E^{10}	3	–2	1	E^{21}	–6	5	–1
E^{11}	–1	–2	–1				

O primeiro passo do algoritmo é separar o conjunto de treinamento original em dois, um com as instâncias positivas e o outro com as negativas. Suponha que, uma vez separados, os baricentros desses conjuntos sejam: $b_1 = \langle 0{,}5\ 0{,}7\rangle$ e $b_2 = \langle -2{,}39\ 2{,}1\rangle$, respectivamente. Desse modo, de acordo com a equação (3.1), o primeiro vetor de pesos encontrado é: $W = \langle 2{,}88\ -1{,}38\rangle$.

Nessa iteração o melhor *bias* encontrado é $\theta = 7{,}2$; definindo assim a equação do hiperplano: $H(W, \theta) = 2{,}88X - 1{,}38Y + 7{,}2$. Esse hiperplano acerta 17 das 21 instâncias e é uma solução ótima para esse conjunto de treinamento, pois não existe um hiperplano que separe mais instâncias nesse conjunto. A Figura 3.7 apresenta o conjunto no plano, bem como a solução encontrada pelo BCPMin.

Note que este hiperplano só será atualizado durante a execução se for encontrado outro hiperplano ótimo que possua um *gap* maior que o *gap* deste hiperplano que é 0,83. A justificativa para substituir um hiperplano ótimo por outro de maior gap é que intuitivamente o hiperplano que possui maior *gap* também tem maior poder de generalização uma vez que o *gap* representa a distância entre o hiperplano e a instância mais próxima.

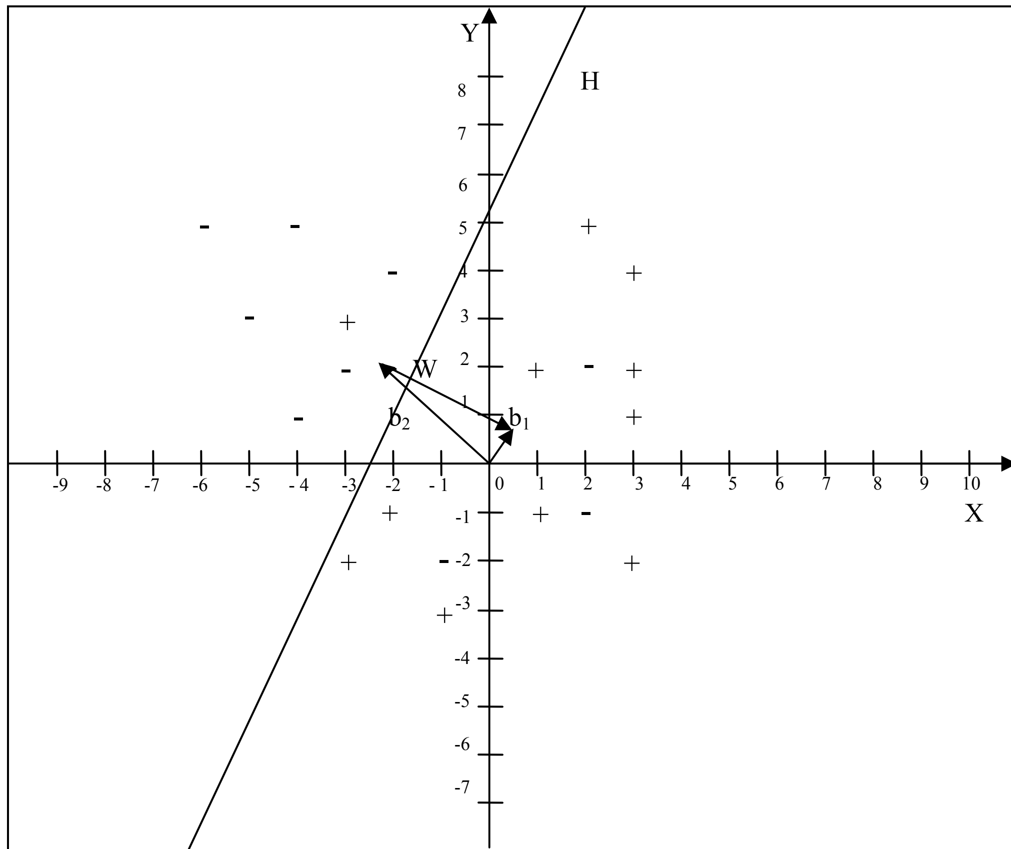


Figura 3.7 – Hiperplano ótimo encontrado pelo BCPMin

3.5 BCPMax – Maximizando o Número de Instâncias Excluídas

A outra versão do BCP é o algoritmo BCPMax, este algoritmo tem objetivo diferente do algoritmo BCPMin, discutido na Seção 3.4. O objetivo agora é encontrar um hiperplano que separe o maior conjunto de instâncias pertencentes a uma mesma classe do resto do conjunto de treinamento.

Algoritmos como este são usados como algoritmos base para alguns algoritmos neurais construtivos que criam a rede através da eliminação gradativa dos exemplos de treinamento, como é o caso do algoritmo Sequential (Capítulo 5, Seção 5.6). O termo instâncias excluídas refere-se ao conjunto linearmente separável encontrado pelo BCPMax.

O maior conjunto de instâncias pertencentes a uma mesma classe, linearmente separável do resto do conjunto de treinamento, será o maior conjunto entre P_{-} e P_{+} (consultar Seção 3.3, equações em (3.14)), pois esses conjuntos possuem as instâncias de mesma classe que estão na área de exclusão.

Se o conjunto de maior cardinalidade for o conjunto P_{-} então o termo *bias* e o termo *gap* são determinados de acordo com as equações em (3.17).

$$\theta = \frac{ext_2 + \max(P_-)}{2} \quad e \quad G_- = \frac{ext_2 - \max(P_-)}{\|W\|} \tag{3.17}$$

Se, no entanto, o conjunto de maior cardinalidade for o conjunto P_+ então o *bias* e o gap são determinados de acordo com as equações em (3.18).

$$\theta = \frac{ext_3 + \min(P_+)}{2} \quad e \quad G_+ = \frac{\min(P_+) - ext_3}{\|W\|} \tag{3.18}$$

Se, por acaso os conjuntos possuírem mesma cardinalidade, $n_- = n_+$, deve-se escolher o que apresenta o maior *gap* G, e se $n_- = n_+ = 0$, todas as instâncias de treinamento estão na área de sobreposição. Uma solução para esse problema é recalcular W fazendo com que os pesos dos exemplos E^k, tais que $WE^k = 0$, aumentem temporariamente.

O pseudocódigo do algoritmo BCPMax está descrito nos Algoritmos 3.4, 3.5 e 3.6. Note que o Algoritmo 3.4 é semelhante ao Algoritmo 3.1 referente ao algoritmo BCPMin; essa parte, comum a ambos os algoritmos, é relativa ao cálculo do baricentro. Os métodos apresentados em Algoritmo 3.4 são, portanto, idênticos aos métodos apresentados em Algoritmo 3.1.

Em Algoritmo 3.5 o método *determinaP*() é chamado, este método, descrito em Algoritmo 3.6, define os conjuntos P_+, P_- e P_{ov}; define também os valores das variáveis tMenos e tMais. Essas variáveis indicam qual a classe das instâncias em P_- e P_+ respectivamente, e são usadas para definir o sinal do *bias*. Considere o caso em que o conjunto P_- tenha mais instâncias que o conjunto P_+. Se as instâncias do conjunto P_- pertencerem à classe positiva, o valor do *bias* dado pela equação (3.17) classificará todas elas corretamente. Se, no entanto, essas instâncias pertencerem à classe negativa, o *bias* dado pela equação (3.17) irá classificá-las incorretamente. Para que essas instâncias possam ser separadas do conjunto de treinamento, no entanto, elas devem ser classificadas corretamente, para isso o *bias* tem seu sinal invertido sem que o vetor de pesos W seja alterado, dessa forma o hiperplano classifica corretamente todas as instâncias em P_-. O caso em que o conjunto P_+ é maior é tratado de maneira análoga.

```
Class BCPMax
begin
{Entradas: E - conjunto de treinamento com n instâncias da forma: (x^k_1, x^k_2,..., x^k_p, C^k), sendo que 1 ≤ k ≤ n
  MAX - número máximo de iterações, pré-determinado}
{Saída: Wpoc e biasPoc}
tMenos ← 0; tMais ← 0
ext2; ext3; P_–; P_+
method BCPMax(Matrix E)
  begin
    exCorretos ← 0
    it ← 0; itMax ← MAX
    a ← 0 ; b ← 0
    nc ← 0; gapPoc ← 0
    for i ← 1 to n do
      begin
        if C^i > 0 then
          begin
            a ← a + 1
            E1^a ← E^i
          end
        else if C^i ≤ 0 then
            begin
              b ← b + 1
              E2^b ← E^i
           end
      end
    alfa ← geraPesos(a)
    mi ← geraPesos(b)
    b1 ← baricentro(alfa, E1)
    b2 ← baricentro(mi, E2)
    W ← b_1 – b_2
    while it < itMax do
      begin
        if it > 1 then
          begin
            r ← aleatorio(0,1)          {r ∈ [0,1] e s ∈ [0,1]}
            s ← aleatorio(0,1)
            if (existeErro(E1) and existeErro(E2)) then
              begin
                be1 ← baricentroErrado(alfa, E1)
                be2 ← baricentroErrado(mi, E2)
                e1 ← be1 – b1
                e2 ← be2 – b2
                W ← W + (b1 + re1) – (b2 + se2)
              end
            else if existeErro(E1) then
                begin
                  be1 ← baricentroErrado(alfa, E1)
                  bs2 ← baricentroPov(mi, E2)
                  e2 ← bs2 – b2
                  W ← W + be1 – (b2 + se2)
               end
            else if existeErro(E2) then
                begin
                  bs1 ← baricentroPov(alfa, E1)
                  be2 ← baricentroErrado(mi, E2)
                  e1 ← bs1 – b1
                  W ← W + (b1 + re1) – be2
                end
          end   {if it > 1}
```

Algoritmo 3.4 – Pseudocódigo do algoritmo BCPMax

```
        for i ← 1 to n1 do    {n1 – número de instâncias em E1}
          V1^i ← –WE1^i
        for i ← 1 to n2 do    {n2 – número de instâncias em E2}
          V2^i ← –WE2^i
        minV2 ← menor(V2)
        maxV1 ← maior(V1)
        if (maxV1 < minV2) then
          biasPoc ← (maxV1 + minV2) / 2
        else                                        {inicio else}
          begin
            determinaP(V1,V2)
            n_ ← |P_|
            n+ ← |P+|
            G_ ← (ext2 – maior(P_)) / modulo(W)
            G+ ← (menor(P+) – ext3) / modulo(W)

            if ((n_ > n+) or (n_ = n+ and G_ > G+)) then
              begin
                bias ← (ext2 + maior(P_)) / 2
                G ← G_
                Exc ← n_
                if tMenos = 1 then
                  eps ← 1
                else
                  begin
                    eps ← –1
                    biasC ← –bias
                  end
              end

            if ((n_ < n+) or (n_ = n+ and G_ < G+ )) then
              begin
                bias ← (ext3 + menor(P+)) / 2
                G ← G+
                Exc ← n+
                if tMais = 0 then
                  eps ← 1
                else
                  begin
                    eps ← –1
                    bias ← –bias
                  end
              end

            if maiorExc < Exc then
              begin
                maiorExc ← Exc
                Gpoc ← G
                biasPoc ← bias
                epsPoc ← eps
                if eps = 1 then
                  Wpoc ← W
                else
                  Wpoc ← –W
              end
          end { else}
        it ← it + 1
      end {whille}
  end {method}
```

Algoritmo 3.5 – Pseudocódigo do algoritmo BCPMax (Continuação)

```
method determinaP(Vector V1, Vector V2)
  begin
    minV1 ← menor(V1);  minV2 ← menor(V2); maxV1 ← maior(V1);  maxV2 ← maior(V2)

    if(minV1 ≤ minV2 and minV2 ≤ maxV1 and maxV1 ≤ maxV2) then
      begin
        ext1 ←  minV1; ext2 ← minV2; ext3 ←  maxV1; ext4 ← maxV2; tMenos ← 1; tMais ← 0
      end
    else if(minV1 ≤ minV2 and minV2 ≤ maxV2 and maxV2 ≤ maxV1) then
        begin
          ext1 ←  minV1; ext2 ← minV2; ext3 ←  maxV2; ext4 ← maxV1; tMenos ← 1; tMais ← 1
        end
    else if(minV2 ≤ minV1 and minV1 ≤ maxV1 and maxV1 ≤ maxV2) then
        begin
          ext1 ←  minV2; ext2 ← minV1; ext3 ←  maxV1; ext4 ← maxV2; tMenos ← 0; tMais ← 0
        end
    else if(minV2 ≤ minV1 and minV1 ≤ maxV2 and maxV2 ≤ maxV1) then
        begin
          ext1 ←  minV2; ext2 ← minV1; ext3 ←  maxV2; ext4 ← maxV1; tMenos ← 0; tMais ← 1
        end
    for j ← 1 to n do
      V^j ← –WE^j

  for i ← 1 to n do
      begin
        if V^i < ext2 then
          P_- ← V^i
        else if V^i > ext3 then
            P_+ ← V^i
        else
            P_ov ← V^i
      end
  end  {fim method}
end {fim Class}
```

Algoritmo 3.6 – Pseudocódigo do algoritmo BCPMax (Continuação)

3.3.6 Aprendizado usando o BCPMax em um Conjunto Não Linearmente Separável

Para exemplificar o funcionamento do algoritmo BCPMax e destacar a diferença com relação ao BCPMin, será usado como conjunto de treinamento o conjunto não linearmente separável da Tabela 3.3. Como apenas a escolha do *bias* determina a diferença entre as variações alguns passos, por serem idênticos a ambas as versões, serão omitidos.

O algoritmo começa a execução separando o conjunto de treinamento em dois, depois de separdos, o próximo passo é encontrar os baricentros das instâncias de classe positiva e negativa, desse modo seja $b_1 = \langle 0{,}66\ 0{,}74\rangle$ e $b_2 = \langle -2{,}11\ 2\rangle$, e de acordo com (3.1), tem-se $W = \langle 2{,}77\ -1{,}26\rangle$.

Achado o vetor de pesos W, calcula-se os conjuntos V1 e V2, e com os extremos desses conjuntos, min(V1) = –10,83; max(V1) = 12,09; min(V2) = –6,8; max(V2) = 22,92; calcula-se ext_2 = –6,8 e ext_3 = 12,09. Determinados os extremos, ext_2 e ext_3 pode-se determinar os conjuntos P_+, P_- e P_{ov}; considerando $1 \leq k \leq n$, todas as instâncias E^k cujos

valores de $v(E^k)$ forem menor que ext_2 estarão em P_- e, todas as instâncias E^k cujos valores de $v(E^k)$ forem maior que ext_3 estará em P_+.

Para este exemplo, seja $IP_+ = \{E^{18}, E^{19}, E^{20}, E^{21}\}$ o conjunto das instâncias cujos respectivos valores em V pertencem também à P_+ e, $IP_- = \{E^4, E^{10}\}$ o conjunto das instâncias cujos respectivos valores no conjunto V pertencem também à P_-. Desse modo, as cardinalidades dos conjuntos P_+ e P_- são $n_+ = 4$ e $n_- = 2$ respectivamente. Como a desigualdade $n_+ > n_-$ é verificada e o termo *bias* é dado pela equação (3.18); desse modo $\theta = 12{,}23$. Como também se verifica a desigualdade $\min(V1) < \min(V2) < \max(V1) < \max(V2)$, segue que $t_- = 1$ e $t_+ = 0$. Como $t_+ = 0$, tem-se que a classe das instâncias em P_+ é negativa e dessa forma o hiperplano atual classifica corretamente essas instâncias, portanto $\varepsilon = 1$.

O hiperplano obtido, $H(W, \theta) = 2{,}77X - 1{,}26Y + 12{,}23$, é ótimo se considerado o objetivo do algoritmo, pois não existe um hiperplano que separe um conjunto maior de instâncias de mesma classe do resto do conjunto de treinamento. A Figura 3.8 mostra os baricentros b_1 e b_2, o vetor de pesos W e o hiperplano H, note que as quatro instâncias de IP_+ são separadas das demais instâncias.

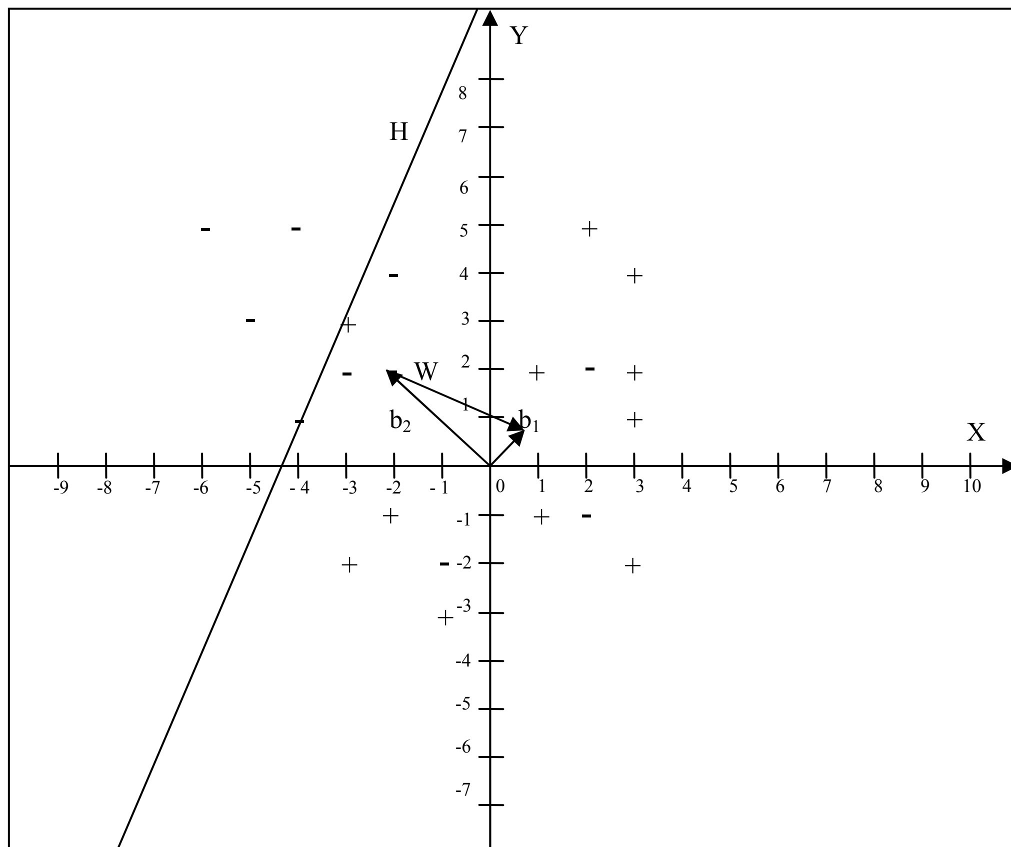


Figura 3.8 – Hiperplano obtido pelo algoritmo BCPMax

4 capítulo

Aprendizado Neural Construtivo – Revisão dos Algoritmos Tower, Pyramid, Tiling e Upstart

4.1 Introdução

Os algoritmos apresentados e discutidos neste capítulo, a saber, os algoritmos Tower e Pyramid [Gallant 1986], o Tiling [Mezard & Nadal 1989] e o Upstart [Frean 1990a], foram originalmente propostos para tarefas de aprendizado em domínios de conhecimento que descrevem apenas duas classes. Essas versões já foram discutidas e analisadas em [Chen *et al.* 1995], [Palma Neto 2004] e [Palma Neto & Nicoletti 2005].

O objetivo desse capítulo é apresentar uma revisão das principais idéias relativas a esses algoritmos bem como apresentar o pseudocódigo de cada um deles, de maneira a fornecer os subsídios necessários para a apresentação e discussão de suas versões estendidas para domínios multiclasses, no Capítulo 7.

Além de viabilizar a pesquisa das extensões para tratar problemas multiclasses, a revisão desses algoritmos é necessária para que no Capítulo 8 seus desempenhos possam ser comparados aos desempenhos dos algoritmos neurais construtivos investigados neste trabalho e descritos no Capítulo 5. Outra contribuição importante é poder avaliá-los com outros algoritmos para o treinamento de TLUs.

As próximas seções apresentam as revisões seguidas dos pseudocódigos dos algoritmos Tower, Pyramid, Tiling e Upstart, nesta ordem. As informações contidas neste capítulo foram extraídas das respectivas referências originais, bem como de [Palma Neto 2004] e [Palma Neto & Nicoletti 2005]. É importante lembrar que as descrições dos algoritmos apresentados nesta dissertação são originais, uma vez que adotam a abordagem orientada a objeto.

4.2 O Algoritmo Tower

O algoritmo Tower é um algoritmo neural construtivo que cria uma rede neural com apenas um neurônio em cada camada intermediária. Em uma rede Tower[8] todo neurônio intermediário é conectado a todos os neurônios da camada de entrada e ao neurônio intermediário criado no passo anterior. Esse procedimento constrói uma rede neural com uma arquitetura semelhante a uma torre (ver Figura 4.1), daí o seu nome.

Note na Figura 4.1 que o primeiro neurônio de entrada, neurônio de cor cinza claro, difere-se dos demais. Este neurônio representa o termo *bias* que é adicionado ao conjunto de treinamento a fim de ajudar na classificação (ver Capítulo 2, Subseção 2.2.1) quando o algoritmo usado para treinar os neurônios é da família Perceptron. Como o termo *bias* só é usado quando alguma variação do Perceptron é usada, no que segue este neurônio será diferenciado para evidenciar o fato de que o termo *bias* nem sempre é adicionado ao conjunto de treinamento.

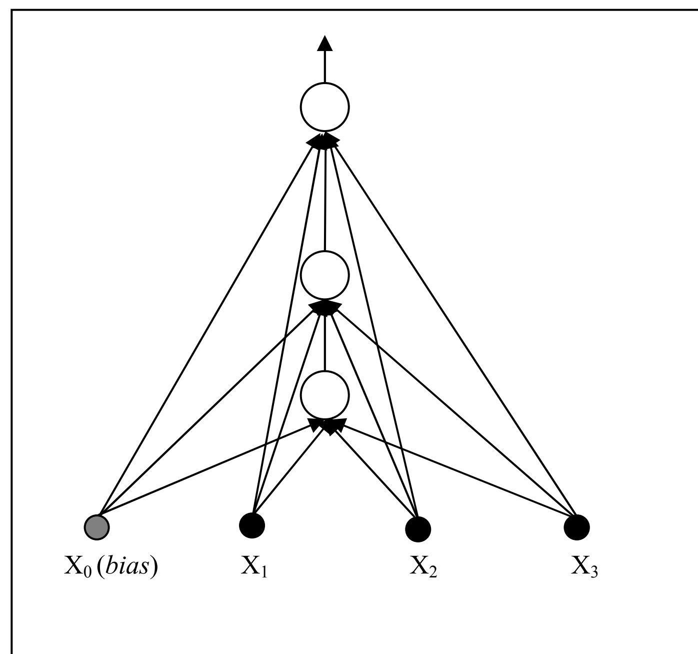


Figura 4.1 – Possível arquitetura de uma rede Tower

A idéia no Tower para a classificação do conjunto de treinamento é bastante simples. A cada iteração é adicionado um novo neurônio em uma nova camada intermediária. Considerando que na iteração 1 o algoritmo adiciona o primeiro neurônio na primeira camada intermediária, na iteração i o algoritmo adicionará o i-ésimo neurônio na i-ésima camada.

[8] Por “rede Tower”, entende-se uma rede neural criada pelo algoritmo Tower.

Com a adição de neurônios ao longo do treinamento, o algoritmo tende a classificar mais instâncias de treinamento corretamente. O processo de inserção de neurônios à rede segue até que um dos três critérios de parada seja satisfeito, a saber: (1) a rede converge, i.e. todos os exemplos de treinamento são classificados corretamente; (2) a adição do último neurônio não ajuda mais na classificação das instâncias de treinamento; (3) o número máximo de neurônios intermediários atingiu o limite pré-determinado.

Para exemplificar o funcionamento do algoritmo considere E o conjunto de treinamento e p o número de atributos de cada instância em E. O primeiro passo do algoritmo é criar um neurônio na primeira camada; esse neurônio será treinado com o conjunto de treinamento original E. Enquanto a rede atual não satisfazer algum critério de parada, o algoritmo seguirá adicionando neurônios à rede.

Os neurônios intermediários, a partir da segunda camada, são treinados com o conjunto de treinamento original E, acrescido de uma dimensão. Esta dimensão representa a saída do neurônio da camada anterior para todas as instâncias de treinamento. O Algoritmo 4.1 apresenta uma versão orientada a objeto do pseudocódigo do algoritmo Tower.

A classe[9] *Tower* possui um vetor de objetos *Camadas*[]; esse vetor representa as camadas intermediárias da rede. Cada camada intermediária é formada por um único neurônio representado aqui pela classe *Neuronio*. A Classe *Neuronio* representa um algoritmo de treinamento de TLU, tal como o PMR ou o BCPMin. O método *saida*() da classe *Neuronio* retorna a saída do neurônio para a instância de treinamento cujo índice foi passado como parâmetro.

No pseudocódigo, o algoritmo adiciona neurônios até que algum critério de parada seja satisfeito. Dois dos critérios de parada são baseados no método *precisao*(); este método retorna a precisão da rede atual, que varia entre 0 e 1. Note que, quando o critério de parada satisfeito for aquele em que a precisão anterior supera ou se iguala à precisão atual, é necessário retirar o último neurônio adicionado à rede para que esta retome sua última configuração bem sucedida. No pseudocódigo o método *apaga*() retira o último neurônio adicionado e é chamado quando o caso descrito anteriormente ocorre.

O conjunto de treinamento para treinar cada neurônio intermediário é criado a cada passo pelo método *criaTreinamento*(). Como descrito no pseudocódigo, este método adiciona uma única dimensão extra ao conjunto de treinamento original; esta dimensão é a saída do último neurônio adicionado, para todas as instâncias do conjunto de treinamento original.

[9] O termo ‘classe’ quando aplicado à uma descrição de algoritmo, diz respeito ao conceito de classe no contexto de programação orientada a objetos.

Note que no Algoritmo 4.1 a entrada correspondente ao termo *bias* (X_0) está entre aspas simples (‘’). No que segue essa notação será usada para evidenciar o fato de que o termo *bias* pode não ser usado (quando não empregado algum variante do Perceptron).

```
Class Tower
begin
{Entradas: E - conjunto de treinamento com n instâncias da forma:
              (‘xk0’, xk1, xk2,..., xkp, Ck), sendo que 1 ≤ k ≤ n
  MAX - número máximo de camadas, pré-determinado}
{Saída: objeto tower que representa a rede neural criada}
camada ← 0{ número de camadas intermediárias da rede}
camadaMax ← MAX
Camadas[camadaMax]  {vetor de neurônios, camada intermediária da rede}

method Tower(Matrix E)
  begin
    E1 ← E
    precisao ← 0
    precisaoAnterior ← 0
    repeat
      begin
        camada ← camada + 1
        Camadas[camada] ← new Neuronio(E1)
        precisaoAnterior ← precisao
        precisao ← precisao(E1)
        E1 ← criaTreinamento(E)
      end
    until (precisaoAnterior ≥ precisao or precisao = 1 or camada = camadaMax)

    if precisaoAnterior ≥ precisao then
      begin
        apaga(Camadas[camada])
        camada ← camada – 1
      end
  end   {method}

method Matrix criaTreinamento(Matrix E)
  begin
    A ← E
    for k ← 1 to n do
      A[k][p+1] ← Camadas[camada].saida(k)
    return A
  end
end  {Class}
```

Algoritmo 4.1 – Pseudocódigo do algoritmo Tower

4.3 O Algoritmo Pyramid

Proposto em [Gallant 1986], o Pyramid é um algoritmo neural construtivo semelhante ao Tower (Seção 4.2). A única diferença está no fato que, cada neurônio adicionado a uma rede Pyramid[10] recebe conexões de todos os neurônios intermediários previamente adicionados, bem como dos neurônios da camada de entrada (ver Figura 4.2).

[10] Por “rede Pyramid”, entende-se uma rede neural criada pelo algoritmo Pyramid.

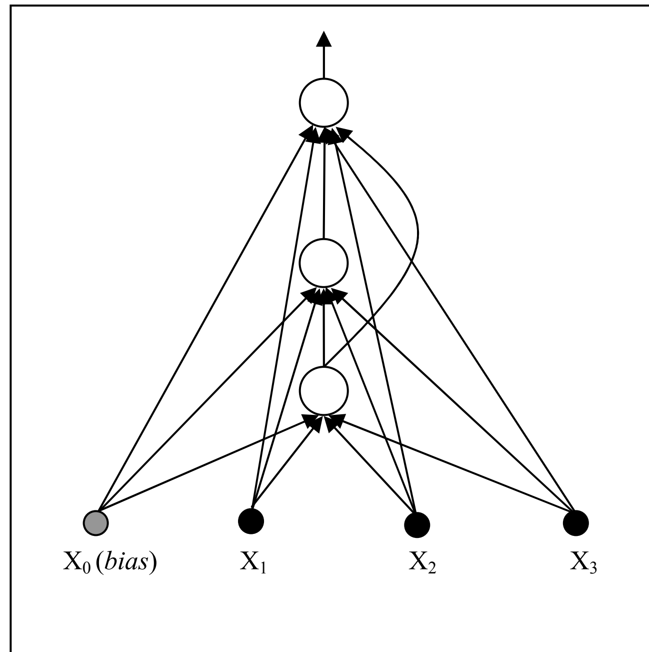


Figura 4.2 – Possível arquitetura de rede Pyramid

O processo de construção da rede usado pelo algoritmo Pyramid é semelhante ao processo utilizado pelo Tower. No Pyramid, entretanto, a dimensão dos vetores de peso, W, aumenta conforme aumenta o número de camadas da rede, uma vez que um neurônio intermediário recebe conexões de todos os neurônios previamente adicionados. Dessa forma, se as instâncias do conjunto de treinamento original, E, possuírem p atributos, o i-ésimo neurônio de uma rede Pyramid terá um vetor de pesos de dimensão p + (i – 1), ou p + i caso o termo *bias* tenha sido adicionado.

O fato do vetor de pesos W aumentar linearmente conforme a rede é construida, não influencia, necessariamente, seu poder de aprendizado. Segundo [Gallant 1994] "Embora pareça que as conexões e pesos extras do algoritmo Pyramid aumentam o poder de aprendizado, isso não é verdade necessariamente. Simulações realizadas foram inconclusivas com relação ao poder de aprendizado, quando as redes envolvidas são Tower e Pyramid".

Assim como no Tower, o Pyramid adiciona um neurônio intermediário por camada até que algum critério de parada seja satisfeito. Os critérios de parada são os mesmos que os utilizados pelo algoritmo Tower, a saber: a rede classifica corretamente todo o conjunto de treinamento; a adição de neurônios à rede não contribui mais na classificação das instâncias de treinamento ou o número máximo de neurônios intermediários foi atingido.

O pseudocódigo do Pyramid, descrito em Algoritmo 4.2, é bastante parecido com o pseudocódigo do Tower. A única diferença entre ambos é como o método *criaTreinamento*()

é usado. Note que o corpo do método *criaTreinamento*() da classe *Pyramid* é idêntico ao do método *criaTreinamento*() da classe *Tower*. A diferença está em como este método é usado durante a construção da rede. Na classe *Tower* o conjunto passado como parâmetro é sempre o conjunto de treinamento original, dessa forma o método *criaTreinamento*() sempre atualiza (ou adiciona, no caso da primeira chamada) o atributo relativo à dimensão p + 1. Já no caso da classe *Pyramid*, o conjunto usado para treinar o neurônio anterior é passado como parâmetro para o método *criaTreinamento*(); desse modo, o novo conjunto de treinamento sempre terá uma dimensão a mais que o conjunto de treinamento anterior.

```
Class Pyramid
begin
{Entradas: E - conjunto de treinamento com n instâncias da forma:
            (‘x^k_0’, x^k_1, x^k_2,..., x^k_p, C^k), sendo que 1 ≤ k ≤ n
  MAX - número máximo de camadas, pré-determinado}
{Saída: objeto pyramid que representa a rede neural criada}
camada ← 0 { número de camadas intermediárias da rede }
camadaMax ← MAX
Camadas[camadaMax]  {vetor de neurônios, camada intermediária da rede}

method Pyramid(Matrix E)
  begin
    precisao ← 0
    precisaoAnterior ← 0
    repeat
      begin
        camada ← camada + 1
        Camadas[camada] ← new Neuronio(E)
        precisaoAnterior ← precisao
        precisao ← precisao(E)
        E ← criaTreinamento(E)
      end
    until (precisaoAnterior ≥ precisao or precisao = 1 or camada = camadaMax)

    if precisaoAnterior ≥ precisao then
      begin
        apaga(Camadas[camada])
        camada ← camada – 1
      end
  end  {method}

method Matrix criaTreinamento(Matrix E)
  begin
    A ← E
    for k ← 1 to n do
      A[k][p+1] ← Camadas[camada].saida(k)
    return A
  end
end  {Class}
```

Algoritmo 4.2 – Pseudocódigo do algoritmo Pyramid

4.4 O Algoritmo Tiling

O algoritmo Tiling, proposto em [Mézard & Nadal 1989], é um algoritmo neural construtivo que cria uma rede neural multicamadas, na qual a camada c recebe conexões somente da camada c – 1 (exceto pelo termo *bias*, quando este é usado). Cada camada intermediária possui um neurônio mestre e alguns neurônios auxiliares e a camada de saída é formada por um único neurônio mestre (ver Figura 4.3).

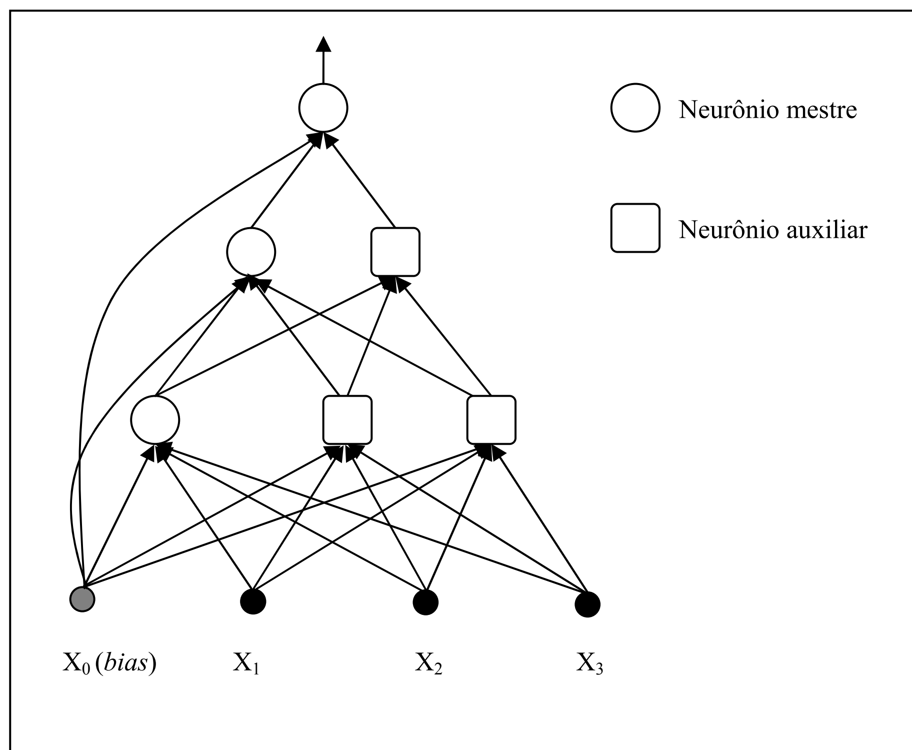


Figura 4.3 – Possível arquitetura de uma rede Tiling

Um neurônio mestre é o primeiro neurônio a ser adicionado em uma nova camada e é responsável pela classificação das instâncias de treinamento; já os neurônios auxiliares têm a função de tornar cada camada confiável. Uma camada é confiável se nenhum par de exemplos de treinamento com classes diferentes tiver a mesma *representação interna* nesta camada.

A representação interna de uma determinada instância de treinamento em uma camada é definida como as saídas dos neurônios desta camada para a instância de treinamento em questão. Considerando que o conjunto de treinamento usado na criação da camada c possua n_c instâncias, esta camada poderá gerar q representações internas, sendo $1 \leq q \leq n_c$. Cada representação interna é formada por j_c elementos, na qual j_c é o número de neurônios da camada c. O conjunto das representações internas de uma determinada camada forma a *matriz de representação interna* desta camada.

Para a construção da rede o primeiro passo do algoritmo é treinar o primeiro neurônio mestre com o conjunto de treinamento original e adiciona-lo à primeira camada. Se, com a adição deste neurônio, a precisão da rede for diferente de 1, o algoritmo torna esta camada confiável adicionando neurônios auxiliares. A camada que acabou de se tornar confiável gera uma matriz de representação interna confiável, i.e. livre de inconsistências. Essa matriz será usada como conjunto de treinamento para o neurônio mestre da segunda camada.

Como já mencionado, para tornar uma camada confiável o Tiling adiciona alguns neurônios auxiliares à camada em questão. Os neurônios auxiliares são adicionados um a um até que a camada se torne confiável. Para a compreensão do processo de criação dos neurônios auxiliares considere que o neurônio mestre da camada c acaba de ser adicionado e agora esta camada precisa se tornar confiável.

O conjunto de treinamento usado para treinar o primeiro neurônio auxiliar será o menor conjunto de instâncias da matriz de representação interna da camada $c - 1$ (ou o conjunto de treinamento original se $c = 1$) que gerar uma representação interna não confiável na camada c. Note que para a inclusão do primeiro neurônio auxiliar as possíveis representações internas da camada c serão 1 e -1. Este procedimento é realizado até que a camada corrente se torne confiável.

O Tiling segue adicionando camadas até que algum critério de parada seja satisfeito. Os critérios de parada são quatro: (1) a convergência da rede; (2) o neurônio mestre adicionado piora o desempenho da rede; (3) o número máximo de camadas é atingido; ou (4) o número máximo de neurônios auxiliares em uma camada atinge o limite, sem que esta tenha se tornado confiável.

Se o processo de construção da rede terminou por ter satisfeito o critério (2), o Tiling precisa recuperar a última configuração bem sucedida. Para isso, o último neurônio mestre bem como todos os neurônios auxiliares da camada anterior são retirados da rede, deixando o mestre da camada anterior como neurônio de saída. Se, no entanto, o critério satisfeito tenha sido o critério (4), o Tiling retira todos os neurônios auxiliares desta camada, deixando o mestre como neurônio de saída.

O pseudocódigo do algoritmo é apresentado em Algoritmo 4.3. A rede é criada camada por camada por meio da chamada recursiva do método *criaCamada*(), que utiliza os métodos *criaTreinamento*() e *criaAuxiliares*(). O método *criaTreinamento*() é responsável por criar o conjunto de treinamento que será usado para treinar o novo neurônio mestre. Considere que a camada $c - 1$ acaba de torna-se confiável, e que esta camada gera q representações internas distintas, o conjunto para treinar o neurônio mestre da camada c será formado por

essas q representações internas. Se o algoritmo de treinamento de TLU usar o termo *bias*, este deve ser inserido no conjunto de treinamento de todo neurônio mestre.

```
Class Tiling
begin
{Entradas: E - conjunto de treinamento com n instâncias da forma:
            (‘x^k_0’, x^k_1, x^k_2,..., x^k_p, C^k), sendo que 1 ≤ k ≤ n
  MAX_CAM - número máximo de camadas, pré-determinado
  MAX_AUX - número máximo de neurônios auxiliares por camada, pré-determinado}
{Saída: objeto tiling que representa a rede neural criada}
camada ← 0 {número de camadas da rede}
camadaMax  ← MAX_CAM
auxiliarMax ← MAX_AUX
Camadas[camadaMax][auxiliarMax] {matriz de neurônios, onde cada linha representa uma camada
intermediária da rede }
precisaoAnterior ← 0

method Tiling(Matrix E)
  begin
    criaCamada(E)
    if precisao ≤ precisaoAnterior then
    begin
      apaga(Camadas[camada][1])
      camada ← camada – 1
      for i ← 2 to auxiliarMax do
        apaga(Camadas[camada][i])
    end
  end

method criaCamada(Matrix E)
  begin
    camada ← camada + 1
    neuronio ← new Neuronio(E)
    Camadas[camada][1] ← neuronio {mestre}
    precisao ← precisao()
    if( precisao > precisaoAnterior and camada < camadaMax and precisao ≠ 1) then
      begin
        precisaoAnterior ← precisao
        criaAuxiliares(E)
        if confiavel() then
          criaCamada(criaTreinamento(E))
        else
          for i ← 2 to auxiliarMax do
            apaga(Camadas[camada][i])
      end
  end  {method}

method criaAuxiliares(Matrix E)
  begin
    auxiliares ← 1
    while (not confiavel() and auxiliares < auxililaresMax) then
      begin
        novoNeuronio  ← new Neuronio(criaTreinamentoAux(E))
        auxiliares ← auxiliares + 1
        Camadas[camada][auxiliares] ← novoNeuronio  {auxiliares}
      end
  end
end {Class}
```

Algoritmo 4.3 – Pseudocódigo do algoritmo Tiling

O método *criaAuxiliares*() cria todos os neurônios auxiliares necessários para tornar a camada em questão confiável (verificado pelo método *confiavel*()). Os neurônios auxiliares são treinados com o conjunto de treinamento criado pelo método *criaTreinamentoAux*(); este método verifica as saídas dos neurônios da camada que está sendo construída em busca de uma representação interna não confiável. Dentre as representações não confiáveis o método retorna o menor conjunto de instâncias que gerou uma das representações internas não confiáveis.

4.4.1 Utilizando o PMR e o BCP na Construção de uma Rede Tiling – O Algoritmo Tiling Híbrido

Considerando os algoritmos PMR e BCPMin para o treinamento de TLUs, foi notado experimentalmente que, esses algoritmos possuem vantagens e desvantagens quanto ao uso em certos algoritmos construtivos e quanto ao conjunto de treinamento utilizado.

Uma vantagem que o PMR parece ter sobre o BCPMin é com relação ao uso deste no treinamento com conjuntos booleanos que representam problemas de paridade. Já o BCPMin leva vantagem para criar redes multicamadas onde são usados mais de um neurônio por camada (tais como no Tiling e PTI). Considerando que essas diferenças entre esses dois algoritmos podem ser complementares, é proposto neste trabalho uma versão do algoritmo Tiling que utiliza ambos os algoritmos para a construção da rede, essa versão será chamada Tiling Híbrido.

Considere o processo de criação da rede usada pelo Tiling, apresentado na Seção 4.4, no Tiling Híbrido sempre que o algoritmo precisar adicionar um neurônio mestre ele cria dois neurônios, com o mesmo conjunto de treinamento, um utilizando o PMR e o outro o BCPMin. Uma vez criados os dois neurônios, verifica-se qual dos neurônios comete menos erros de classificação no conjunto de treinamento que foi usado para criá-lo. O neurônio que comete menos erros é então inserido na rede como mestre, e o outro é descartado.

O processo de escolha entre os algoritmos não se aplica à construção da camada com neurônios auxiliares. Sempre que for necessário tornar uma camada confiável é usado o algoritmo BCPMin, pois este se mostrou muito mais eficiente na tarefa de tornar uma camada confiável.

Note que a utilização de dois algoritmos com abordagens diferentes para o treinamento de TLUs pode trazer alguns problemas, como o uso do termo *bias* por exemplo. Nesta versão o *bias* é usado somente pelo algoritmo PMR, portanto não é possível adicionar o termo *bias*

ao conjunto de treinamento original antes de iniciar o processo de treinamento do algoritmo. Este problema é resolvido adicionando o *bias* somente aos conjuntos de treinamento usados para treinar os neurônios com o PMR.

A avaliação desta versão do algoritmo Tiling bem como dos outros algoritmos apresentados neste capítulo em vários domínios de conhecimento seguido de uma análise entre esses algoritmos, será apresentado no Capítulo 8.

4.5 O Algoritmo Upstart

O algoritmo Upstart [Frean 1990a] é um algoritmo neural construtivo que cria uma rede neural com o formato de uma árvore binária (ver Figura 4.4). A construção da rede no Upstart é feita no sentido saída – entrada, ou seja, o algoritmo começa treinando o neurônio de saída e segue adicionando neurônios filhos à medida que o treinamento progride.

A idéia básica no Upstart para classificar as instâncias de treinamento, é criar neurônios filhos ao longo do treinamento com o objetivo de que estes neurônios corrijam os erros cometidos por seus pais. Se os filhos também não conseguirem classificar as instâncias de treinamento, o processo se repete. Desse modo, o último neurônio corrigirá os erros de seu pai que, por sua vez, corrigirá os erros de seu pai e assim sucessivamente, até o neurônio de saída.

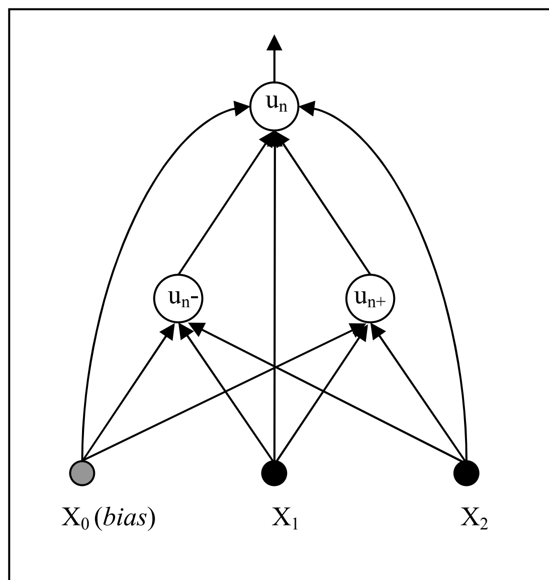


Figura 4.4 – Uma possível arquitetura de uma rede Upstart

Um neurônio Upstart pode ter até dois filhos (daí a razão de uma rede Upstart[11] ter a arquitetura de uma árvore binária), um para corrigir os erros de positivo e o outro para corrigir os erros de negativo. Erros de positivo ocorrem quando um neurônio u_n classifica um exemplo de treinamento E^k, com classe C^k positiva, como de classe negativa. Analogamente, erros de negativo ocorrem quando um neurônio u_n classifica um exemplo E^k, com classe C^k negativa, como de classe positiva, como mostra a Figura 4.5.

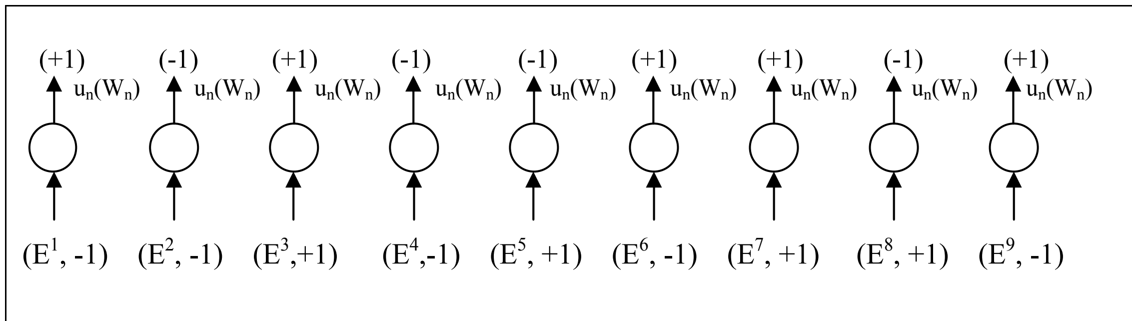


Figura 4.5 – O neurônio u_n classifica corretamente os exemplos de treinamento: E^2, E^3, E^4 e E^7 Comete erro de positivo em E^5 e E^8 e erro de negativo em E^1, E^6 e E^9

Quando um neurônio não classifica corretamente todas as instâncias de treinamento, ele pode ter cometido erros de positivo, erros de negativo ou ambos. Supondo que o neurônio u_n tenha cometido somente erros de positivo, o Upstart alocará um neurônio filho u_{n+} que terá como objetivo corrigir os erros cometidos por u_n (pai) e ao mesmo tempo, manter inalteradas as classificações corretas feitas pelo pai.

Para corrigir os erros de positivo cometidos pelo neurônio u_n, o neurônio u_{n+} deverá ser ativado para as instâncias que fizeram que u_n cometesse erros de positivo e deve ficar inativo para as instâncias de classe negativa. Considere C^k a classe da instância E^k do conjunto de treinamento do neurônio u_n, e $O^k_{u_n}$ a saída deste neurônio para a instância E^k. Seja A o conjunto de treinamento que será usado para treinar u_{n+}, este conjunto será formado por toda instância E^k, para $1 \leq k \leq n$, com as classes C^k_A dadas como em (4.1).

$$\begin{aligned} &(1)\ \text{Se } C^k = -1 \text{ então } C^k_A = -1 \\ &(2)\ \text{Se } C^k = 1 \text{ e } O^k_{u_n} = 0, \text{ então } C^k_A = 1 \\ &(3)\ \text{Se } C^k = 1 \text{ e } O^k_{u_n} = 1, \text{ então } C^k_A = -1 \end{aligned} \qquad (4.1)$$

[11] Por “rede Upstart” entende-se uma rede neural criada pelo algoritmo Upstart.

Se, no entanto, os erros cometidos por u_n forem erros de negativo, a forma de corrigí-los é análoga. Considere um neurônio u_n que comete erros de negativo, e que o algoritmo crie um neurônio filho u_{n-} para corrigí-los. Para que u_{n-} não interfira nas instâncias de treinamento que u_n classifica corretamente e também nas que u_n comete erros de positivo, as saídas das conexões devem ser 1 somente para as instâncias que devem ser corrigidas (neste caso, as instâncias de treinamento com erros de negativo) e 0 para as demais. Dessa forma o conjunto de treinamento A usado para treinar u_{n-} deve ser composto por toda instância E^k, tal que $1 \le k \le n$, com suas classe C_A^k trocadas acordo com (4.2).

$$
\begin{aligned}
&(1)\ \text{Se } C^k = 1 \text{ então } C_A^k = -1 \\
&(2)\ \text{Se } C^k = -1 \text{ e } O_{u_n}^k = 1, \text{ então } C_A^k = 1 \\
&(3)\ \text{Se } C^k = -1 \text{ e } O_{u_n}^k = 0, \text{ então } C_A^k = -1
\end{aligned}
\tag{4.2}
$$

Note que em ambos os casos ((4.1) e (4.2)) as instâncias que se encaixam na descrição do item (3) poderiam possuir $C_A^k = 1$, ou ainda poderiam nem ser adicionadas ao conjunto de treinamento. A troca das classes para positiva nestes casos, faria com que a saída já correta do neurônio u_n fosse reforçada. Fazer com que essas instâncias possuam classe 1, no entanto, além de não ser necessário podem atrapalhar na classificação correta dos exemplos que necessitam ser corretamente classificados, i.e. os exemplos que geram erros de positivo ou de negativos. Como mencionado anteriormente estas instâncias podem não ser adicionadas, no entanto, adicioná-las com classe 0 permite que o vetor de pesos seja mais bem treinado.

Para que um neurônio que foi adicionado para corrigir um determinado tipo de erro não atrapalhe na classificação das instâncias que não apresentam esse erro, é preciso que esse novo neurônio altere somente os potenciais de ativação para as instâncias que apresentaram este tipo de erro. Por essa razão a saída inativa dos neurônios deve ter valor 0 e não –1.

Como a rede é construída de maneira recursiva, a sub-árvore (quando existir) correspondente ao filho gerado para corrigir erros de positivo de cada neurônio é construída primeiro[12]. Se este neurônio também cometeu erros de negativo então a sub-árvore correspondente ao filho gerado para corrigir erros de negativo é criada. Depois de criadas ambas as sub-árvores (ou apenas uma delas se a outra não existir), o neurônio é retreinado

[12] Pelo fato do erro de positivo ser tratado primeiro no pseudocódigo.

com as saídas dessas sub-árvores, e este neurônio por sua vez, se não for a raiz, irá gerar a saída para retreinar seu pai, assim sucessivamente até a raiz.

Quando um neurônio é folha ele não precisa ser retreinado, ele apenas gera a saída para todas as instâncias de treinamento que servirão para treinar o seu pai. Um neurônio é folha quando não possui nenhum filho. Para que um neurônio não tenha filhos, uma de duas situações pode ocorrer: (1) O neurônio classifica corretamente todas as instâncias de seu conjunto de treinamento. (2) O neurônio comete algum tipo de erro (eventualmente ambos) e, quando um filho seu for treinado para corrigir esse tipo de erro, o filho comete um número de erros, desse tipo, maior que o cometido pelo pai. Desse modo o filho não é inserido na rede.

O Algoritmo 4.4 mostra o pseudocódigo orientado a objeto do algoritmo Upstart. O primeiro passo do algoritmo é treinar a raiz e passá-la para o método *criaRede*() juntamente com o conjunto de treinamento original para que a rede seja construída. O método *criaRede*() é chamado recursivamente até que a rede seja criada. Os métodos *erroPositivo*() e *erroNegativo*() recebem um neurônio como parâmetro e retornam, para este neurônio, o número de erros de positivo e de negativo respectivamente.

Se o neurônio atual cometer algum erro, este criará um neurônio filho para corrigi-lo. O neurônio filho será treinado com um conjunto de treinamento criado por *criaTreinamentoPos*() ou *criaTreinamentoNeg*(), dependendo do tipo do erro que este deverá corrigir. Os métodos que criam os conjuntos de treinamento recebem o conjunto de treinamento do neurônio atual para que, a partir deste, seja criado o novo conjunto de treinamento.

Caso o neurônio recém criado não cometer mais erros, de mesmo tipo, que seu pai, ele é adicionado à rede através do método *insere*(). O método *insere*() cria a rede neural com a estrutura de uma árvore binária. Depois de construída, para que a rede classifique uma nova instância, basta que esta seja percorrida em pós-ordem.

Toda vez que um neurônio tem suas sub-árvores completadas o método *modificaEntrada*() é chamado, este método adiciona as saídas das sub-árvores (uma ou duas) do neurônio atual ao conjunto de treinamento original para que o neurônio atual seja retreinado.

```
Class Upstart
begin
{Entradas: E - conjunto de treinamento com n instâncias da forma:
            ('x^k_0', x^k_1, x^k_2,..., x^k_p, C^k), sendo que 1 ≤ k ≤ n
 MAX - número máximo de neurônios intermediarios da rede, pre-determinado}
{Saída: objeto upstart que representa a rede neural criada}
neuronioMax ← MAX
nroNeuronio ← 0

method Upstart(Matrix E)
  begin
    raiz ← new Neuronio(E)
    criaRede(E,raiz)
  end

method criaRede(Matrix E, Neuronio neuronioAtual)
  begin
    erroP ← erroPositivo(neuronioAtual)
    if (erroP > 0 and nroNeuronio ≤  neuronioMax) then
      begin
        EPos ← criaTreinamentoPos(E,neuronioAtual)
        novoNeuronio ← new Neuronio(Epos)
        novoErroP ← erroPositivo(novoNeuronio)
        if novoErroP < erroP then
          begin
            insere(novoNeuronio)
            nroNeuronio ← nroNeuronio + 1
            criaRede(Epos, novoNeuronio)
          end
      end

    erroN ← erroNegativo(neuronioAtual)
    if (erroN > 0 and nroNeuronio ≤ neuronioMax) then
      begin
        ENeg ← criaTreinamentoNeg(E,neuronioAtual)
        novoNeuronio ← new Neuronio(ENeg)
        novoErroN ← erroNegativo(novoNeuronio)
        if novoErroN < erroN then
          begin
            insere(novoNeuronio)
            nroNeuronio ← nroNeuronio + 1
            criaRede(ENeg, novoNeuronio)
           end
      end

    if not folha(neuronioAtual) then
      begin
        novoE ← modificaEntrada( E, neuronioAtual)
        neuronioAtual ← Neuronio(novoE)
      end
  end
end   {Class}
```

Algoritmo 4.4 – Pseudocódigo do algoritmo Upstart

5 capítulo Aprendizado Neural Construtivo – Algoritmos Shift, Offset, PTI, Perceptron Cascade e Sequential

5.1 Introdução

Este Capítulo investiga cinco algoritmos neurais construtivos para o tratamento de problemas de classificação que envolvem duas classes, a saber: Shift, Offset, PTI, Perceptron Cascade e Sequential. Todos os algoritmos discutidos neste Capítulo, com exceção do Sequential, podem usar qualquer algoritmo de treinamento de TLU para duas classes.

Os algoritmos Shift, PTI e Perceptron Cascade são considerados em duas versões, uma utilizando o algoritmo PMR (Seção 2.4) e outra utilizando o algoritmo BCPMin (Seção 3.4), para o treinamento das TLUs. O Sequential, por apresentar uma maneira diferente de construção da rede, poderá usar dois algoritmos para treinar seus neurônios: o IncLp (Subseção 5.6.1) e o BCPMax (Seção 3.5). Na seção referente ao Offset é também apresentada uma variante que combina características do Offset com as do Tiling, em um algoritmo que é proposta deste trabalho, chamado OffTiling.

Para os algoritmos construtivos Shift, PTI e Perceptron Cascade são apresentados dois exemplos de execução, um utilizando o PMR e o outro o BCPMin. Os exemplos usam como conjunto de treinamento as instâncias que representam o conceito de Paridade-3 ímpar; vale lembrar que este conjunto sofre a adição do termo *bias* (Subeção 2.2.1) sempre que o PMR for utilizado. A execução do Offset é exemplificada no domínio Paridade-3 usando o PMR e a sua variante híbrida, OffTiling é exemplificada usando o BCPMin.

5.2 O Algoritmo Shift

O algoritmo Shift [Amaldi & Guenin 1998], assim como o Upstart (Capítulo 4, Seção 4.5), cria a rede neural começando pelo neurônio de saída da rede. Diferente do Upstart, entretanto, o Shift cria uma rede com uma única camada intermediária.

Assim como o Upstart, o algoritmo Shift distingue dois tipos de erros de classificação: erros de positivo e erros de negativo. Como visto anteriormente (Capítulo 4, Figura 4.5), erros

de positivo ocorrem quando um neurônio u_n classifica uma instância de treinamento E^k, de classe positiva, como de classe negativa. Erros de negativo ocorrem quando um neurônio u_n classifica uma instância de treinamento E^k, de classe negativa, como de classe positiva.

Se durante o treinamento a rede Shift[13] atual não conseguir classificar corretamente as instâncias de treinamento, os erros de classificação cometidos por ela são calculados e, então, um novo neurônio é criado, na camada intermediária. Esse novo neurônio será ligado diretamente ao neurônio de saída por meio de uma conexão com um peso Δ de modo a corrigir os erros de maior freqüência, sejam eles erros de positivo ou erros de negativo.

Para exemplificar o processo de correção de erros, considere v^k como o somatório das entradas do neurônio de saída para a instância de treinamento E^k. A Figura 5.1 ilustra a adição de um novo neurônio (u_2) em uma rede Shift. Neste exemplo, para a adição desse neurônio, o valor de v^k é calculado[14] como na equação (5.1), para cada instância E^k do conjunto de treinamento:

$$v^k = W_0 E^k + O_1^k \Delta_1 \tag{5.1}$$

na qual W_0 é o vetor de pesos do neurônio de saída u_0, O_1^k é a saída do neurônio u_1 para a instância E^k e Δ_1 é o peso da conexão entre os neurônios u_0 e u_1. A saída dos neurônios intermediários é dada pela a função de ativação $\varphi(\,)$, na qual $y \in R$, representa o resultado do produto interno entre o vetor de pesos e a instância corrente, como na equação (5.2).

$$\varphi(y) = \begin{cases} 1 \text{ se } y > 0 \\ 0 \text{ se } y \le 0 \end{cases} \tag{5.2}$$

Seja O_2^k a saída do novo neurônio (u_2) relativa à instância E^k. Uma das duas (ou ambas) situações, identificadas como (A) erro de positivo, (B) erro de negativo, discutidas a seguir podem acontecer.

(A) TRATAMENTO DE ERROS DE POSITIVO - O neurônio u_2 adicionado à camada intermediária da rede deverá ser conectado ao neurônio de saída por meio de uma conexão com um peso Δ_2 suficientemente grande de modo que sua saída fique ativa (O_2^k = 1) quando

[13] Por "rede Shift" entende-se uma rede neural criada pelo algoritmo Shift.
[14] O exemplo considera que u_2 ainda não foi adicionado, portando v^k é calculado sem o uso de u_2.

$v^k \leq 0$ e $C^k = 1$ (erro de positivo) e fique inativa ($O_2^k = 0$) quando $v^k \leq 0$ e $C^k = -1$. Para fazer isso, as instâncias de treinamento que devem ser usadas para treinar u_2 serão aquelas instâncias do conjunto de treinamento, para as quais o correspondente valor de v^k seja tal que $v^k \leq 0$.

Uma vez treinado u_2 (e determinado o vetor de pesos W_2), o Shift determina o valor de Δ_2 de acordo com o seguinte procedimento:

(1) Identifica quais instâncias do conjunto original de treinamento ativam u_2 (ou seja, tornam $O_2^k = 1$).

(2) Para cada uma dessas instâncias identificadas em (1) coleciona o correspondente valor v^k.

(3) Determina menor valor dos v^k colecionados, notado por $\underline{v}^k$.

(4) Calcula o valor do peso Δ_2 usando a equação (5.3)

$$\Delta_2 = -1(\underline{v}^k - \delta) \tag{5.3}$$

na qual δ é um número real pequeno e positivo que é adicionado à $\underline{v}^k$, para que a instância E^k que gerou $\underline{v}^k$ também possa ser classificada corretamente.

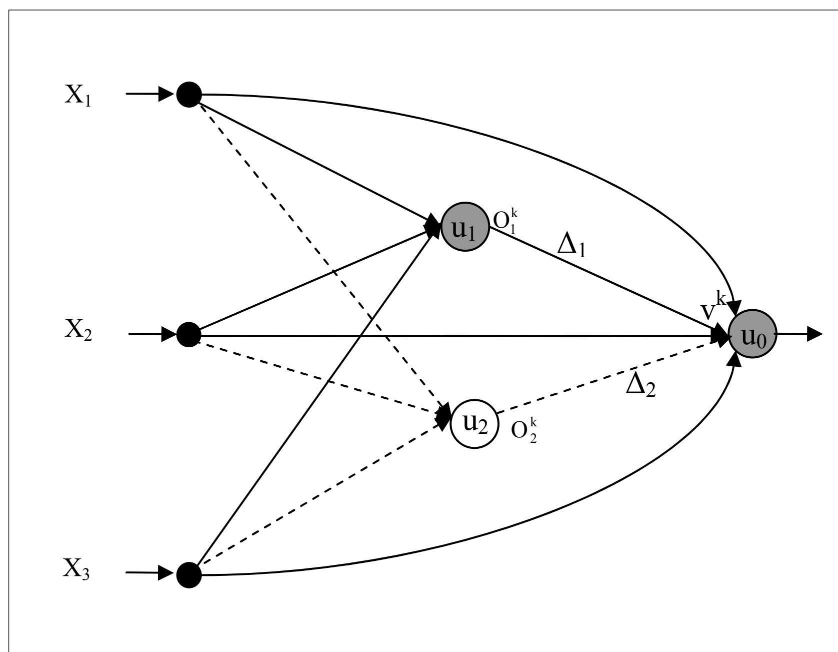


Figura 5.1 – Adicionando o neurônio u_2 (em branco) a uma rede Shift. Os demais neurônios (em cinza) foram adicionados em fases anteriores. As conexões tracejadas farão parte da rede após a inclusão de u_2

Como na situação de erro de positivo os v^ks são negativos o menor dos v^ks ($\underline{v}^k$), será o maior valor positivo (quando multiplicado por -1), o que permite corrigir todos os erros de positivo.

(B) TRATAMENTO DE ERROS DE NEGATIVO - O neurônio u_2 alocado para corrigir erros de negativo, é adicionado à camada intermediária da rede e conectado ao neurônio de saída por meio de uma conexão com um peso Δ_2 suficientemente pequeno (negativo).

Para o tratamento de erros de negativo é necessário fazer com que o novo neurônio u_2 fique ativo ($O_2^k = 1$) quando $v^k > 0$ e $C^k = -1$ (erro de negativo), e fique inativo ($O_2^k = 0$) quando $v^k > 0$ e $C^k = 1$. Sendo assim, o conjunto de treinamento do novo neurônio deve conter as instâncias E^k que geram $v^k > 0$, porém com suas respectivas classes invertidas. O procedimento para a determinação de Δ_2:

(1) Identifica quais instâncias do conjunto original de treinamento ativam u_2 (ou seja, tornam $O_2^k = 1$).
(2) Para cada uma dessas instâncias identificadas em (1) coleciona o correspondente valor v^k.
(3) Determina maior valor dos v^k colecionados, notado por $\underline{\underline{v}}^k$.
(4) Calcula o valor do peso Δ_2 usando a equação (5.4)

$$\Delta_2 = -1(\underline{\underline{v}}^k + \delta) \quad (5.4)$$

No pseudocódigo descrito em Algoritmo 5.1 o primeiro passo do algoritmo é criar o neurônio de saída (com o nome ‘saida’ no pseudocódigo). O neurônio de saída é um objeto da classe *Neuronio* que implementa um algoritmo de treinamento de TLU (e.g. PMR ou BCPMin). Uma vez criado o neurônio de saída, os métodos *erroPositivo*() e *erroNegativo*() são chamados. Esses métodos retornam o número de erros de positivo e de negativo, respectivamente, que a rede (por enquanto apenas o neurônio de saída) comete. O tipo de erro mais freqüente determinará a tarefa a ser realizada pelo próximo neurônio criado.

Se o tipo de erro mais freqüente for o erro de positivo então o método *criaTreinamentoPositivo*() é chamado. Esse método retorna o conjunto de treinamento para os neurônios que foram alocados para reparar erros de positivo. Esse conjunto é formado pelas instâncias de treinamento que produzem v^k negativo ou nulo. De modo similar o método

criaTreinamentoNegativo() é responsável por criar o conjunto treinamento para os neurônios alocados para corrigir erros de negativo. Porém agora o método seleciona as instâncias que geram v^k positivo e, além disso, todo exemplo tem sua classe trocada de 1 para -1 e vice-versa.

O novo neurônio, seja ele criado para corrigir erros de positivo ou erros de negativo, é treinado com o conjunto recém criado e então inserido no vetor de neurônios *Camada*[]. Os métodos *corrigePositivo*() e *corrigeNegativo*() calculam o valor de Δ e o inserem no vetor de pesos do neurônio de saída, na posição referente à saída do neurônio em questão.

```
Class Shift
begin
{Entradas: E - conjunto de treinamento com n instâncias da forma:
            (‘x^k_0’, x^k_1, x^k_2,..., x^k_p, C^k), sendo que 1 ≤ k ≤ n
  MAX - número máximo de neurônios intermediarios da rede, pre-determinado}
{Saída: objeto shift que representa a rede neural criada}
camadaMax ← MAX
nroNeuronios ← 0 {conta número de neurônios na camada intermediária}
Camada[camadaMax]  {vetor formado por neurônios intermediários}

method Shift (Matrix E)
  begin
    precisaoAnteiror ← 0
    saida ← new Neuronio(E)
    precisao ← precisao()
    erroPositivo ← erroPositivo()
    erroNegativo ←  erroNegativo()
    while (precisao > precisaoAnterior  and precisao ≠ 1 and nroNeuronios ≤ camadaMax) do
      begin
        if erroPositivo  ≥ erroNegativo then
          begin
            Epos ← criaTreinamentoPositivo()
            Camada[nroNeuronios] ← new Neuronio(Epos)
            corrigePositivo()
          end
        else if erroNegativo > erroPosivivo then
              begin
                Eneg ← criaTreinamentoNegativo()
                Camada[nroNeuronio] ← new Neuronio(Eneg)
                corrigeNegativo()
              end
        precisao ← precisao()
        nroNeuronio ← nroNeuronio +1
        erroPositivo ← erroPositivo()
        erroNegativo ←  erroNegativo()
      end
  end  {method}
end  {Class}
```

Algoritmo 5.1 – Pseudocódigo do algoritmo Shift

5.2.1 Exemplo de Execução do Algoritmo Shift usando o Algoritmo PMR

Como comentado na Seção 5.1, para exemplificar o funcionamento do algoritmo Shift, será usado o conjunto de treinamento que representa o conceito de Paridade-3 ímpar, mostrado na Tabela 5.1 com a adição do termo *bias*.

O primeiro passo do algoritmo é treinar o neurônio de saída. Como o conjunto não é linearmente separável o vetor de pesos W_0 não classifica corretamente todos os exemplos do conjunto de treinamento. Seja $W_0 = \langle 1\ 1\ 1\ -1 \rangle$, o vetor de pesos obtido no treinamento do neurônio de saída. Esse vetor comete dois erros, um erro de negativo em E^2 e um erro de positivo em E^7. Como não houve um tipo de erro predominante, o erro de positivo é tratado primeiro. A Tabela 5.1 apresenta o conjunto de treinamento e os resultados dos v^k para $1 \le k \le n$ da rede atual.

Tabela 5.1 – Conjunto que representa o problema Paridade-3 e os valores de v^k (k = 1,...,8) da rede atual

Exemplos E^k (k = 1,...,8)	**X_0(*bias*)**	**X_1**	**X_2**	**X_3**	**Classe**	**v^k (k = 1,...,8)**
E^1	1	1	1	1	1	2
E^2	1	1	1	–1	–1	4
E^3	1	1	–1	1	–1	0
E^4	1	1	–1	–1	1	2
E^5	1	–1	1	1	–1	0
E^6	1	–1	1	–1	1	2
E^7	1	–1	–1	1	1	–2
E^8	1	–1	–1	–1	–1	0

Como neste caso o erro de positivo será tratado primeiro, o algoritmo deve inserir um neurônio intermediário u_1 para corrigir os erros de positivo. Como mencionado anteriormente, para treinar um neurônio que deve corrigir erros de positivo, são usados os exemplos que geram $v^k \le 0$ na rede atual.

O conjunto de treinamento do novo neurônio, u_1 é mostrado na Tabela 5.2. O treinamento de u_1 resulta no vetor de pesos $W_1 = \langle -1\ -1\ -1\ 1 \rangle$. Tal vetor de pesos classifica corretamente todos os exemplos de treinamento da Tabela 5.2.

Tabela 5.2 – Conjunto de treinamento do neurônio alocado para corrigir erros de positivo

Exemplos E^k (k = 3, 5, 7, 8)	**X_0 (*bias*)**	**X_1**	**X_2**	**X_3**	**Classe C^k (k = 3, 5, 7, 8)**
E^3	1	1	–1	1	–1
E^5	1	–1	1	1	–1
E^7	1	–1	–1	1	1
E^8	1	–1	–1	–1	–1

O próximo passo do algoritmo é chamar o método *corrigePositivo*() que tem como função escolher um peso Δ que minimize os erros da rede. Para a escolha de Δ, é necessário identificar os v^ks, com $1 \leq k \leq n$, para os quais a instância correspondente E^k produza $O_1^k = 1$ (ver Tabela 5.3). Nesse caso a única opção de valores de v^k é $\underline{v}^7 = -2$. Como o neurônio a ser adicionado é para corrigir erros de positivo, $\Delta_1 = -1(v^k - \delta)$. Para este exemplo foi usado $\delta = 0{,}5$ e, portanto, $\Delta_1 = 2{,}5$.

Depois de adicionado um neurônio, a rede verifica se ainda comete erro ou se algum critério de parada foi satisfeito, por exemplo, se a rede comete mais erros que cometia antes da inserção do último neurônio. Se nenhum critério de parada for satisfeito, o conjunto v^k é recalculado e o treinamento continua. A Tabela 5.3 mostra as saídas do neurônio u_1 e os novos valores de v^k para $1 \leq k \leq n$.

Tabela 5.3 – Conjunto com as classes originais saída de O_1 e novos valores de v^k (k = 1,..., 8)

Exemplos E^k (k = 1,..., 8)	**O_1**	**Classe C^k (k = 1,..., 8)**	**v^k (k = 1,... 8)**
E^1	–1	1	2
E^2	–1	–1	4
E^3	–1	–1	0
E^4	–1	1	2
E^5	–1	–1	0
E^6	–1	1	2
E^7	1	1	0,5
E^8	–1	–1	0

Como a rede melhora sua precisão, o processo de inserção de neurônios intermediários continua. A rede atual comete um erro de negativo, na instância E^2. Para corrigir este erro o algoritmo cria o neurônio u_2. Como este neurônio foi alocado para corrigir erros de negativo, as instâncias usadas para treiná-lo serão todas E^k instâncias que geram $v^k > 0$ para $1 \leq k \leq n$ com suas classes invertidas, como mostra a Tabela 5.4.

Tabela 5.4 – Conjunto de treinamento do neurônio alocado para corrigir erros de negativo

Exemplos E^k (k = 1, 2, 4, 6, 7)	**X_0 (*bias*)**	**X_1**	**X_2**	**X_3**	**Classe C^k (k = 1, 2, 4, 6, 7)**
E^1	1	1	1	1	–1
E^2	1	1	1	–1	1
E^4	1	1	–1	–1	–1
E^6	1	–1	1	–1	–1
E^7	1	–1	–1	1	–1

O treinamento do neurônio u_2 com o conjunto de treinamento da Tabela 5.4 resulta no vetor de pesos $W_2 = \langle -1\ 1\ 1\ -1 \rangle$. Como feito na adição do neurônio anterior, o próximo passo

é calcular o valor de Δ_2. Para o caso de erros de negativo o Δ escolhido deve ser o maior dentre os valores de v^k com $1 \leq k \leq n$ associados a E^k, que ativem o último neurônio adicionado, i.e. $O_2^k = 1$.

Neste caso a única instância que ativa o neurônio u_2 é a instância E^2, assim o único valor que pode ser escolhido é $\underline{\underline{v}}^2 = 4$. Como o Δ para erros de negativo é dado por (5.4), tem-se $\Delta_2 = -4,5$. A configuração da rede recém obtida, mostrada na Figura 5.2, classifica corretamente todos os exemplos de treinamento.

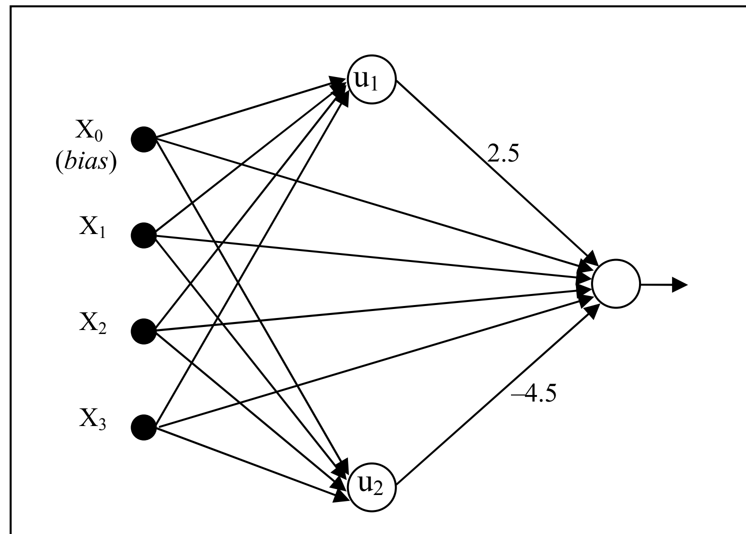


Figura 5.2 – Conceito de Paridade-3 aprendido por uma rede Shift que usa o PMR para treinar TLUs

Note que a solução para o problema de Paridade-3 resolvida pelo Shift envolveu dois neurônios intermediários. De fato, em [Amaldi & Guenin 1998] pode ser encontrada a prova de que todo problema de Paridade-n pode ser resolvido pelo algoritmo Shift por meio da criação de uma rede com $\left\lceil \frac{n+1}{2} \right\rceil - 1$ neurônios intermediários.

5.2.2 Exemplo de Execução do Algoritmo Shift usando o Algoritmo BCPMin

Considere o problema de Paridade-3, apresentado anteriormente e o conjunto de treinamento, porém sem a adição do termo *bias*, mostrado na Tabela 5.5. Suponha que o treinamento do neurônio de saída pelo BCPMin com o conjunto da Tabela 5.5 resulte no vetor de pesos $W_0 = \langle 10\ -9{,}1\ 11{,}4 \rangle$ e *bias* $\theta_0 = 0$. O hiperplano definido por esse vetor e este *bias*

comete dois erros de classificação, um erro de positivo em E^6 e um erro de negativo em E^3. A Tabela 5.5 mostra também os valores de v^k para $1 \leq k \leq 8$ da rede atual.

Tabela 5.5 – Conjunto de treinamento paridade-3 com os valores de v^k (k = 1,..., 8) para a rede atual

Exemplos E^k (k = 1,..., 8)	**X_1**	**X_2**	**X_3**	**Classe C^k (k = 1,..., 8)**	**v^k (k = 1,..., 8)**
E^1	1	1	1	1	12,3
E^2	1	1	–1	–1	–10,4
E^3	1	–1	1	–1	30,4
E^4	1	–1	–1	1	7,7
E^5	–1	1	1	–1	–7,7
E^6	–1	1	–1	1	–30,4
E^7	–1	–1	1	1	10,4
E^8	–1	–1	–1	–1	–12,3

Como a rede comete dois erros, um de cada tipo, o erro de positivo é tratado primeiro. O conjunto de treinamento usado para treiná-lo será formado pelas instâncias do conjunto de treinamento original que geram $v^k \leq 0$, para $1 \leq k \leq n$. Desse modo, as instâncias usadas no treinamento do novo neurônio u_1, serão as instâncias E^2, E^5, E^6 e E^8.

Suponha que o treinamento do neurônio u_1 tenha resultado no vetor de pesos $W_1 = \langle -0,6\ 0,73\ -0,72 \rangle$ e *bias* $\theta_1 = -1,5$. Como este hiperplano separa somente a instância E^6 das outras instâncias, o peso da conexão será $\Delta_1 = 30,9$. A Tabela 5.6 mostra as saídas do neurônio u_1 e os valores de v^k, com $1 \leq k \leq 8$, atualizados.

Tabela 5.6 – Conjunto com as classes originais, saídas de u_1 e novos valores de v^k (k = 1,..., 8)

Exemplos E^k (k = 1,..., 8)	**O_1^k (k = 1,..., 8)**	**Classe C^k (k = 1,..., 8)**	**v^k (k = 1,..., 8)**
E^1	–1	1	12,3
E^2	–1	–1	–10,4
E^3	–1	–1	30,4
E^4	–1	1	7,7
E^5	–1	–1	–7,7
E^6	–1	1	0,5
E^7	1	1	10,4
E^8	–1	–1	–12,3

Como a rede ainda comete um erro de negativo, em E^3, o algoritmo insere um neurônio para corrigir este erro. O conjunto usado para treinar um neurônio, para que este corrija erros de negativo, deve ser formado pelas instâncias que na rede atual geram $v^k > 0$, para $1 \leq k \leq 8$, com as suas classes invertidas. Assim, como pode ser verificado na Tabela 5.6, o conjunto de treinamento do neurônio u_2 será formado pelas instâncias: E^1, E^3, E^4, E^6 e E^7 com as respectivas classes invertidas.

Suponha que o neurônio u_2, treinado com o conjunto de treinamento $\{E^1, E^3, E^4, E^6, E^7\}$, tenha obtido o vetor de pesos $W_2 = \langle 1{,}1\ -0{,}86\ 1 \rangle$ e *bias* $\theta_2 = -2{,}1$. Essa configuração permite que u_2 fique ativo somente para a instância de treinamento E^3 dentre todas do conjunto original, dessa forma $\Delta_2 = -30{,}9$. A rede, mostrada na Figura 5.3 não comete nenhum erro.

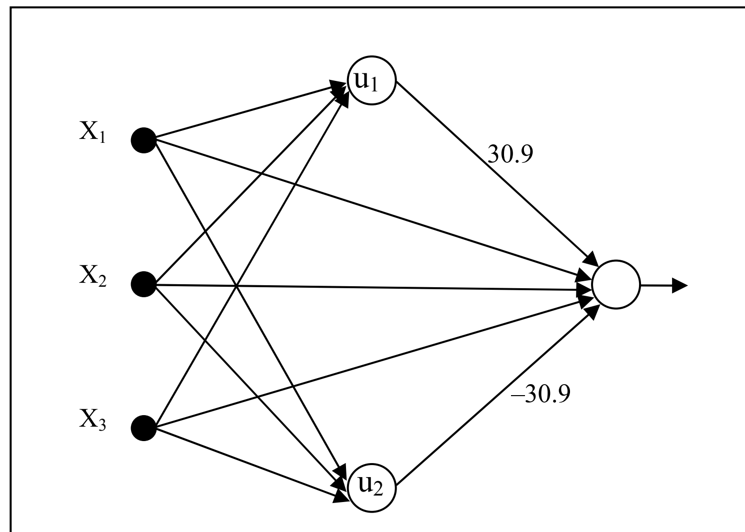


Figura 5.3 – Rede Shift que representa o conceito de Paridade-3, na qual as TLUs foram treinadas com o BCPMin

5.3 O Algoritmo Offset

Proposto por [Martinez & Estève, 1992] para problemas com atributos booleanos, o Offset é um algoritmo neural construtivo que cria uma rede com duas camadas intermediárias. O processo de construção da rede é dividido em duas fases. Na primeira fase o algoritmo constrói uma rede com duas camadas intermediárias. Na segunda fase, a rede construída é podada, com a eliminação de qualquer neurônio redundante da segunda camada.

A primeira camada intermediária é criada adicionando-se a ela neurônios de acordo com a necessidade da rede, até que o último neurônio não cometa nenhum erro de classificação em seu conjunto de treinamento. Ao fim desse processo essa camada é uma máquina de paridade[15] [Mitchison & Durbin 1989] que tem por objetivo transformar qualquer função booleana da entrada em um problema de paridade.

O próximo passo do algoritmo é construir a segunda camada intermediária que tem como objetivo resolver o problema de paridade criado pela primeira camada; a segunda

[15] Uma máquina de paridade é uma rede neural que transforma qualquer problema booleano em um problema de paridade.

camada terá, a principio, um número de neurônios igual à dimensão da paridade criada pela primeira camada, ou seja, o número de neurônios na primeira camada.

Após o termino da construção da segunda camada e, conseqüentemente, da primeira fase do algoritmo, a rede Offset classifica corretamente todas as instâncias de treinamento. Como pode acontecer que a segunda camada da rede possua neurônios redundantes, é iniciada a segunda fase do algoritmo, que tem como objetivo reduzir o tamanho da rede com o propósito de aumentar seu poder de generalização [Baum & Haussler 1989], por meio da eliminação de neurônios redundantes na segunda camada.

5.3.1 Fase I - Construção da Primeira Camada de uma Rede Offset

Considere a função $f(C^k, O^k)$ como uma determinada medida de distância entre C^k e O^k para a instância de treinamento E^k, tal que O^k é a saída do neurônio u_n para a instância E^k de classe C^k. A idéia é corrigir os erros medidos por f com a introdução de um novo neurônio u_{n+1}.

Para que o novo neurônio u_{n+1} possa corrigir os erros do neurônio anterior é necessário que este fique ativo para os exemplos que foram classificados incorretamente por u_n e inativo para os que já estão corretamente classificados. Escolhendo a função booleana ou-exclusivo (*xor*), como a função de medição de distância, os exemplos que geram erros serão mapeados para 1 e os que estão classificados corretamente em –1. Dessa forma, o procedimento de correção sucessiva na primeira camada pode ser feito como descrito a seguir.

O primeiro neurônio u_1 é treinado com o conjunto de exemplos original E. Supondo que este neurônio cometa alguns erros de classificação, é necessário a adição de um segundo neurônio, u_2, para corrigir os erros de u_1. O novo neurônio u_2 será treinado com E1, que é o conjunto de treinamento E com suas classes trocadas pelos resultados da função de distância entre a saída do neurônio u_1 e suas respectivas classes, para cada toda instância de E.

Se u_2 não achar um vetor de pesos que classifique corretamente todos os exemplos de E1, será criado um novo neurônio u_3, e este será treinado com E2 que será formado a partir de E1 da mesma maneira que no caso anterior, e assim sucessivamente até que a criação e adição de u_n classifique corretamente todos as instâncias do conjunto de treinamento E(n – 1).

É importante ressaltar que o processo de criação da primeira camada de uma rede Offset, previamente citado, somente irá convergir se as instâncias de treinamento possuírem atributos booleanos i.e. $\langle x_1, x_2,..., x_p\rangle$ é tal que $x_i \in \{0,1\}$, $1 \le i \le p$ ou $\langle x_1, x_2,..., x_p\rangle$ é tal que $x_i \in \{-1,1\}$, $1 \le i \le p$.

5.3.2 Fase I - Construção da Segunda Camada de uma Rede Offset

Quando o algoritmo termina a construção da primeira camada, esta camada torna-se uma máquina de paridade que representa o problema booleano original como um problema de paridade-j, sendo j é o número de neurônios que compõem a primeira camada. O problema inicial foi transformado em um problema de paridade de dimensão j.

Para resolver o problema de paridade criado pela primeira camada o algoritmo cria uma segunda camada. Esta camada é formada, a principio, por j neurônios ou seja, o mesmo número de neurônios da camada anterior. Os neurônios da segunda camada, no entanto, não são treinados, e dessa forma, esses neurônios não têm vetor de pesos associado. Ao invés disso, os neurônios da segunda camada fazem uso de *thresholds* pré-determinados para classificação das paridades. O *threshold* do i-ésimo neurônio, com $1 \leq i \leq j$, inserido na segunda camada é obtido segundo a equação (5.5).

$$\theta_i = 2i - j - 1 \qquad (5.5)$$

Cada *threshold* determinado pela equação (5.5) separa duas paridades, uma que contém as representações com números ímpares de 1s, e a outra, com números pares de 1s. Assim a segunda camada cria j + 1 hiperplanos intercalados entre par e ímpar.

O próximo passo é fazer com que a saída fique inativa para valores pares de i e ativa para valores ímpares. Para isso são adicionados os pesos +1 e –1 de maneira intercalada, começando com +1, e fazendo o *threshold* do neurônio de saída igual a 0,5. Essa configuração permite que a saída varie de acordo com a paridade de i.

5.3.3 Fase II – A Poda em uma Rede Offset

Como a segunda camada divide o espaço em todos os hiperplanos possíveis, é provável que algum hiperplano nunca seja usado. Portanto os neurônios que geram esses hiperplanos são redundantes na rede, de modo que podem ser podados. Essa etapa é útil para melhorar a generalização da rede, uma vez que a rede criada anteriormente pode não ser mínima devido ao fato da segunda camada ser construída com um número pré-determinado de neurônios.

Para saber se a rede gerada não é mínima considere q o número de representações internas distintas e j o número de neurônios na primeira camada da rede.

(1) Se $q = 2^j$, o problema de paridade–j está completo. A rede necessita de todos os neurônios da segunda camada para resolvê-lo; neste caso a rede é mínima.

(2) Se $q < 2^j$, só uma parte do problema de paridade está presente. É possível que algum neurônio da segunda camada seja redundante; neste caso a rede não é mínima.

A implementação da poda pode ser feita de duas maneiras: uma delas é verificar quais dos *thresholds* não são ativados por nenhuma representação interna. Para isso cada linha da matriz de representações internas é somada e comparada com os *thresholds*. A outra maneira é utilizar a rede no conjunto original de exemplos e marcar quais dos neurônios da segunda camada nunca são usados.

5.3.4 Considerações sobre o Pseudocódigo do Algoritmo Offset

O Algoritmo 5.2 mostra o pseudocódigo do algoritmo Offset. O primeiro passo do algoritmo é criar um neurônio na primeira camada (representada no pseudocódigo por meio do vetor *Camada*[]). Esse neurônio é treinado com o conjunto de treinamento original, utilizando algum algoritmo para o treinamento de TLUs para duas classes, representado no pseudocódigo pela classe *Neuronio*.

Se o primeiro neurônio não consegue classificar corretamente todos os exemplos de treinamento, o método *criaTreinamento*(), cria um novo conjunto de treinamento, baseado no conjunto de treinamento usado para criar o neurônio anterior. Este método aplica a função de distância *xor* entre as classes do conjunto usado para treinar o último neurônio e as saídas deste neurônio para criar as classes do novo conjunto de treinamento.

O processo de inserção de neurônios na primeira camada termina quando o último neurônio adicionado conseguir classificar corretamente todo o seu conjunto de treinamento. Quando isso acontece a primeira camada é uma máquina de paridade e o próximo passo é criar a segunda camada para resolver o problema da paridade. A segunda camada é representada no pseudocódigo pelo vetor *SegCamada*[]; esse vetor guarda os *thresholds*, que são obtidos com o uso de um contador.

Para verificar se o último neurônio adicionado classifica todas as instâncias de seu conjunto de treinamento corretamente, este neurônio juntamente com seu conjunto de treinamento são passados como parâmetro para o método *precisao*(). Quando a precisão obtida for igual a 1 o método *criaMRI*() é chamado para criar a matriz de representação interna da primeira camada. Esse método constrói a matriz de representações internas com as saídas já obtidas dos neurônios da primeira camada e retira as possíveis repetições.

O último passo do algoritmo é verificar se a poda pode ser realizada na segunda camada; caso seja, o método *poda*() é chamado. Este método faz a poda por meio da verificação das somas de cada linha da matriz de representações internas. Se existir, no vetor de *thresholds*, um valor que não seja ultrapassado por nenhum resultado das somas das linhas da matriz de representação interna, então esse valor não é copiado para o novo vetor que representará a segunda camada.

```
Class Offset
begin
{Entradas: E - conjunto de treinamento com n instâncias da forma:
            (‘x^k_0’, x^k_1, x^k_2,..., x^k_p, C^k), sendo que 1 ≤ k ≤ n
  MAX - número máximo de neurônios primeira camada da rede, pre-determinado}
{Saída: objeto offset que representa a rede neural criada}
camadaMax ← MAX
nroNeuroino ← 1 {conta número de neurônios na primeira camada }
Camada[camadaMax]  {vetor de neurônio que representa a primeira camada}
SegCamada[camadaMax] {vetor de thresholds, representando a segunda camada}

method Offset (Matrix E)
  begin
    Camada[nroNeuronio] ← new Neuronio(E)
    while (nroNeuronio < camadaMax and precisao() ≠ 1) do
      begin
        E ← criaTreinamento(E)
        nroNeuronio ← nroNeuronio + 1
        Camada[nroNeuronio] ← new Neuronio(E)
      end
            {fim da primeira camada}
    for i ← 1 to nroNeuronio do
      SegCamada[i] ← 2i – nroNeuronio – 1
            {fim da segunda camada e fim da primeira etapa}

    A ← criaMRI()
    q  ← |A|
    if ( q < 2^nroNeuronio) then
      SegCamada ← poda(A)
  end

Vector poda (Matriz A)
  begin
    vetAux[]← ⟨0,0,…,0⟩ {vetor de q posições iniciado com 0’s}
    for i ← 1 to q do
      begin
        soma ← 0
        for j ← 1 to nroNeuronio do
          soma ← soma + A[i][j]

        for k ← 1 to q do
        if vetAux[k] ≠ 1 and soma > SegCamada[k] then
          vetAux[k] ← 1
      end

    for i ← 1 to nroNeuronio do
      if vetAux[i] = 1 then
        novoSegCamada[i] ← SegCamada[i]
  end
end  {Class}
```

Algoritmo 5.2 – Pseudocódigo do algoritmo Offset

5.3.5 Exemplo de Execução do Algoritmo Offset usando o Algoritmo PMR

Para exemplificar a execução do algoritmo Offset será usado o conjunto de treinamento que representa o problema de Paridade-3 e, como o algoritmo PMR será usado para o treinamento de cada neurônio individualmente, o termo *bias* é inserido no conjunto de treinamento.

A primeira etapa do algoritmo é criar uma máquina de paridade na primeira camada. Para isso o algoritmo inicia treinando o primeiro neurônio da primeira camada; suponha que o treinamento deste neurônio tenha resultado no vetor de pesos $W_1 = \langle 1\ 1\ -1\ 1 \rangle$. Esse neurônio comete dois erros de classificação nos oito exemplos de treinamento, especificamente nos exemplos E^3 e E^6. A Tabela 5.7 mostra o conjunto de treinamento original (instâncias e classe) juntamente com a saída do primeiro neurônio bem como o resultado da função distância $f(C^k,O^k)$. O conjunto de treinamento usado para treinar o próximo neurônio é o conjunto original no qual cada instância tem sua classe C^k substituída por $f(C^k,O^k)$.

Tabela 5.7 – Representação do problema paridade-3 mais a saída do primeiro neurônio e a função de distância aplicada entre C e O_1

Exemplos E^k (k = 1,..., 8)	**X_1**	**X_2**	**X_3**	**Classe(C^k) (k = 1,..., 8)**	**O_1^k (k = 1,..., 8)**	**$f(C^k, O_1^k)$ (k = 1,..., 8)**
E^1	1	1	1	1	1	–1
E^2	1	1	–1	–1	–1	–1
E^3	1	–1	1	–1	1	1
E^4	1	–1	–1	1	1	–1
E^5	–1	1	1	–1	–1	–1
E^6	–1	1	–1	1	–1	1
E^7	–1	–1	1	1	1	–1
E^8	–1	–1	–1	–1	–1	–1

Como o primeiro neurônio não classificou corretamente todas as instâncias de seu conjunto de treinamento, a primeira camada ainda não esta completa. O algoritmo então adiciona outro neurônio para tentar terminar a camada. Suponha que o treinamento do segundo neurônio da primeira camada tenha resultado no vetor de pesos $W_2 = \langle -1\ -1\ 1\ -1 \rangle$. Esse vetor comete apenas um erro de classificação em seu conjunto de treinamento, na instância E^3. A Tabela 5.8 mostra o conjunto de treinamento do segundo neurônio, a saída deste e a função de distância. Note que as instâncias de treinamento são as instâncias originais, e as classes são os resultados da função de distância aplicada às classes do conjunto de treinamento do neurônio anterior com as respectivas saídas do mesmo.

Tabela 5.8 – Conjunto de treinamento do segundo neurônio da primeira camada juntamente com a saída do neurônio e a função de distância

Exemplos E^k (k = 1,..., 8)	**X_1**	**X_2**	**X_3**	**Classe(C^k) (k = 1,..., 8)**	**O_2^k (k = 1,..., 8)**	**f(C^k, O_2^k) (k = 1,..., 8)**
E^1	1	1	1	–1	–1	–1
E^2	1	1	–1	–1	1	1
E^3	1	–1	1	1	–1	–1
E^4	1	–1	–1	–1	–1	–1
E^5	–1	1	1	–1	–1	–1
E^6	–1	1	–1	1	1	–1
E^7	–1	–1	1	–1	–1	–1
E^8	–1	–1	–1	–1	–1	–1

O segundo neurônio adicionado à primeira camada ainda comete erros em seu conjunto de treinamento, dessa forma a camada ainda não é uma máquina de paridade. Assim o terceiro neurônio é adicionado. Suponha que o seu treinamento tenha criado o vetor de pesos $W_3 = \langle -1 \;\; 1 \;\; -1 \;\; 1 \rangle$. Note que este vetor de pesos classifica corretamente todos os exemplos do seu conjunto de treinamento, mostrado na Tabela 5.9, juntamente com a saída do neurônio u_3. Como o neurônio classifica todos os exemplos corretamente não há necessidade de calcular a função de distância, que neste caso teria todos seus valores iguais a –1.

Tabela 5.9 – Conjunto de treinamento do terceiro neurônio da primeira camada mais a saída deste

Exemplos E^k (k = 1,..., 8)	**X_1**	**X_2**	**X_3**	**Classe(C^k) (k = 1,..., 8)**	**O_3^k (k = 1,..., 8)**
E^1	1	1	1	–1	–1
E^2	1	1	–1	1	–1
E^3	1	–1	1	–1	1
E^4	1	–1	–1	–1	–1
E^5	–1	1	1	–1	–1
E^6	–1	1	–1	–1	–1
E^7	–1	–1	1	–1	–1
E^8	–1	–1	–1	–1	–1

Terminada a primeira camada, esta é uma máquina de paridade, como mostrado na Tabela 5.10. Para resolver o problema de paridade criado na primeira camada, a segunda camada é criada. Os neurônios desta camada não são treinados; como mencionado anteriormente, eles fazem uso de *thresholds* dados pela equação: $\theta_i = 2i - j - 1$ (neste exemplo $1 \leq i \leq 3$). Os *thresholds* obtidos são pois $\theta_1 = -2$, $\theta_2 = 0$ e $\theta_3 = 2$. A Figura 5.4 mostra a rede neural criada pelo algoritmo Offset, para o exemplo em questão.

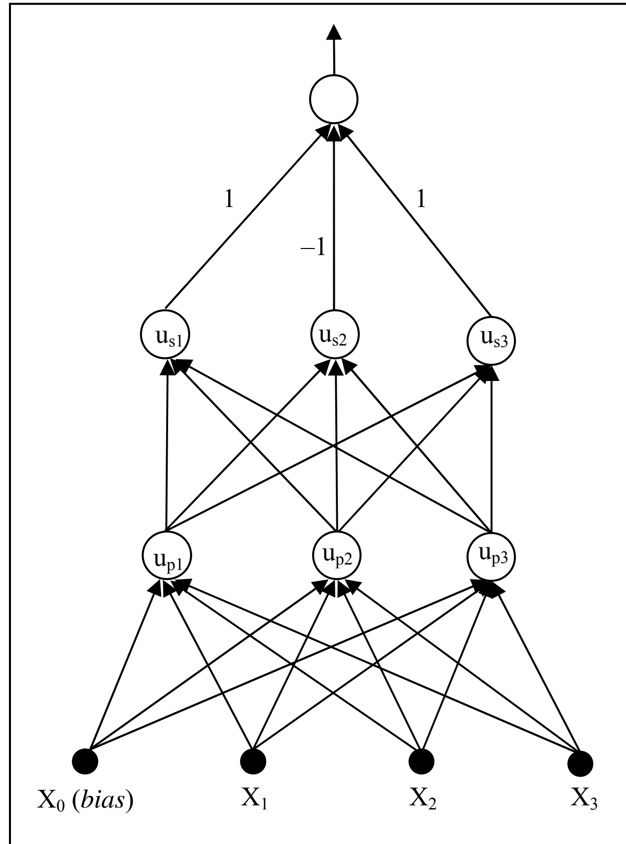


Figura 5.4 – Conceito Paridade-3 aprendido por uma rede Offset. Neurônios indexados com p estão na primeira camada e os indexados com s, estão na segunda camada

As saídas dos neurônios da segunda camada são apresentadas na Tabela 5.10; note que essas saídas formam um problema de paridade que deverá ser solucionado pela segunda camada da rede.

Tabela 5.10 – Saída dos neurônios da primeira camada para toda instância do conjunto de treinamento

Exemplos E^k (k = 1,...8)	**O_1^k (k = 1,..., 8)**	**O_2^k (k = 1,..., 8)**	**O_3^k (k = 1,..., 8)**	**Classe (C^k) (k = 1,..., 8)**
E^1	1	–1	–1	1
E^2	–1	–1	–1	-1
E^3	1	–1	1	-1
E^4	1	–1	–1	1
E^5	–1	–1	–1	-1
E^6	–1	1	–1	1
E^7	1	–1	–1	1
E^8	–1	–1	–1	-1

Uma vez terminada a fase de construção da rede, o algoritmo verifica a matriz de representações internas gerada pela primeira camada (Tabela 5.10) e determina o número q de representações internas distintas. A nova matriz de representação interna, agora sem repetições, é mostrada na Tabela 5.11.

Tabela 5.11 – Matriz de representações internas sem repetições

Exemplos E^k (k = 1,..., 8)	**O_1^k (k = 1,..., 8)**	**O_2^k (k = 1,..., 8)**	**O_3^k (k = 1,..., 8)**	**Classe (C^k) – paridade (k = 1,..., 8)**
E^1, E^4, E^7	1	–1	–1	1
E^2, E^5, E^8	–1	–1	–1	–1
E^3	1	–1	1	–1
E^6	–1	1	–1	1

Como a matriz de representações internas possui quatro entradas distintas (correspondentes às 4 linhas da matriz), tem-se q = 4. Como $q < 2^j$ (no caso j = 3) o problema de paridade não está completo, portanto deve haver neurônios redundantes na segunda camada. A Figura 5.5 mostra a rede Offset da Figura 5.4 em sua versão podada.

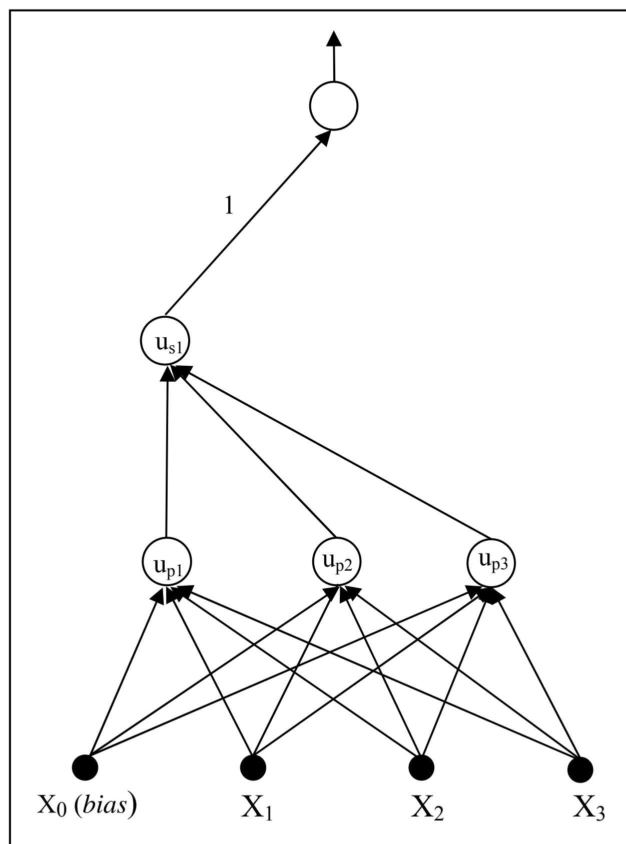


Figura 5.5 – Conceito Paridade-3 aprendido por uma rede Offset após fase da poda (comparar com a Figura 5.4)

5.3.6 Uma Alternativa para o Tratamento de Problemas com Atributos Reais para uma Rede Offset – A Proposta Híbrida OffTiling

Como mencionado anteriormente, o algoritmo Offset somente converge para problemas com atributos booleanos [Martinez & Estève 1992] e para alguns problemas com

atributos reais. O motivo pelo qual esse algoritmo não suporta treinamento com instâncias descritas com valores reais é que o processo de construção da primeira camada precisaria de muitos neurônios para tornar o problema real em um problema de paridade.

Com o propósito de eliminar esta limitação do algoritmo uma abordagem híbrida é proposta neste trabalho e descrita nesta seção. Essa abordagem consiste em usar um outro algoritmo, no caso o Tiling (Seção 4.4), para discretizar os dados, i.e. criar uma representação booleana do conjunto de treinamento original e, então, construir a rede Offset a partir da camada criada pelo algoritmo Tiling.

Uma pesquisa detalhada sobre algoritmos de discretização (ou quantização) pode ser encontrada em [Dougherty *et al.* 1995]. A maioria dos métodos de discretização, no entanto, não considera o inter-relacionamento entre atributos, pois realiza a discretização de maneira independente. Existe, porém, um método de discretização baseado em instâncias chamado quantização vetorial (*vector quantization*) que considera o relacionamento entre atributos (ver [Kohonen 1989], [Okabe *et al.* 1992] e [Yang & Honavar 1996] para detalhes deste método).

A razão pela qual o algoritmo Tiling foi escolhido[16] se deve ao fato que o processo que este algoritmo usa para criar uma representação confiável do conjunto de treinamento é, na verdade, um método dinâmico de quantização vetorial [Parekh 1998]. Dessa forma, nenhuma característica do conjunto original é perdida ao final da camada Tiling.

O processo de construção da rede pelo algoritmo híbrido OffTiling é iniciado com a criação de uma camada confiável por meio do processo de construção de camadas do algoritmo Tiling. Esta camada gera uma matriz de representação interna confiável, formada por atributos booleanos. Esta camada gera $1 \leq q \leq n$ representações internas distintas (n é o número de instâncias do conjunto de treinamento original).

O próximo passo é construir a rede Offset usando como conjunto de treinamento a matriz de representações internas (com as q representações internas distintas) gerada pela camada Tiling. O pseudocódigo do algoritmo híbrido chamado de OffTiling é apresentado em Algoritmo 5.3.

[16] O algoritmo PTI (detalhado na Seção 5.4) também pode ser usado.

```
Class OffTiling
begin
{Entradas: E - conjunto de treinamento com n instâncias da forma:
        ('x^k_0', x^k_1, x^k_2,..., x^k_p, C^k), sendo que 1 ≤ k ≤ n
  MAX - número máximo de neurônios na camada Tiling, pré-determinado}
{Saída: objeto offTiling que representa a rede neural criada}
auxiliaresMax ← MAX
CamadaTiling[camadaMax] {vetor de neurônios que representa camada do algoritmo Tiling}

method OffTiling (Matrix E)
  begin
    A ← criaCamada(E)
    offset ← new Offset(A)
  end

method Matrix criaCamada(Matrix E)
  begin
    neuronio ← new Neuronio(E)
    CamadaTiling[1] ← neuronio {mestre}
    criaAuxiliares(E)
    A ← criaMRI()
    return A
  end

 method criaAuxiliares (Matrix E)
  begin
    auxiliares ← 1
    while ( naoConfiavel() and auxiliares ≤ auxililaresMax) then
      begin
        novoNeurnonio  ← new Neuronio(criaTreinamentoAux(E))
        auxiliares ← auxiliares +1
        CamadaTiling[auxiliares] ← novoNeuronio  {auxiliares}
      end
  end
end  {Class}
```

Algoritmo 5.3 – Pseudocódigo do algoritmo híbrido OffTiling

5.3.7 Exemplo de Execução do Algoritmo OffTiling usando o Algoritmo BCPMin

Para exemplificar o processo de criação da rede neural pelo algoritmo hibrido OffTiling, será usado o conjunto com valores reais representado na Tabela 5.12. Por se tratar de um conjunto pequeno somente o Offset com o BCPMin seria capaz de criar uma rede para classificar esse conjunto de treinamento. No entanto o uso de um conjunto reduzido neste exemplo facilitará o entendimento do algoritmo OffTiling.

Tabela 5.12 – Conjunto de treinamento com vinte instâncias descritas por valores reais

Exemplos E^k (k = 1,... 10)	**X_1**	**X_2**	**Classe C^k (k = 1,... 10)**	**Exemplos E^k (k = 11,..., 20)**	**X_1**	**X_2**	**Classe C^k (k = 11,..., 20)**
E^1	–3,0	3,5	1	E^{11}	–1,0	–2,9	–1
E^2	1,4	2,9	1	E^{12}	–1,1	–3,0	–1
E^3	2,1	2,2	1	E^{13}	–2,6	–1,6	–1
E^4	3,8	1,8	1	E^{14}	–3,3	–2,7	–1
E^5	3,3	4,9	1	E^{15}	–2,4	2,8	–1
E^6	3,3	2,7	1	E^{16}	–2,8	4,0	–1

E^7	2,5	5,6	1	E^{17}	–4,7	1,2	–1
E^8	–1,0	–1,0	1	E^{18}	–4,6	5,3	–1
E^9	2,6	–1,0	1	E^{19}	–5,8	3,6	–1
E^{10}	3,0	–2,0	1	E^{20}	–6,0	5,0	–1

Como mencionado anteriormente o primeiro passo do algoritmo OffTiling é criar a primeira camada, ou camada Tiling, assim como o algoritmo Tiling. Para isso o primeiro neurônio é adicionado e treinado. Suponha que o treinamento tenha resultado no vetor de pesos $W_1 = \langle 6{,}62\ 1{,}63 \rangle$ e *bias* $\theta_1 = 9{,}79$. O hiperplano gerado por esse vetor de pesos e *bias* comete um erro na instância de treinamento E^1; dessa forma a saída do neurônio não é confiável.

A fim de tornar a camada Tiling confiável, o algoritmo adiciona outro neurônio, este neurônio assim como no Tiling é treinado com o menor conjunto que gerou um padrão não confiável, i.e. (instâncias de classe negativa mais a instância E^1). Suponha que o treinamento com essas instâncias tenha resultado no vetor de pesos $W_2 = \langle 1{,}4\ 0{,}53 \rangle$ e *bias* $\theta_2 = 2{,}64$.

A inserção do segundo neurônio ainda não torna a camada confiável. Desse modo o processo continua com a inserção de um terceiro neurônio. O terceiro neurônio é treinado com as instâncias E^1, E^{15} e E^{16}, uma vez que estas formam o menor conjunto que dá origem a uma representação interna não confiável. Suponha que o terceiro neurônio, treinado com o conjunto $\{E^1, E^{15}, E^{16}\}$, resulte no vetor de pesos $W_3 = \langle -0{,}4\ 0{,}08 \rangle$ e *bias* $\theta_3 = -1{,}43$.

Note que com a inserção do terceiro neurônio a camada Tiling se torna confiável; o próximo passo agora é criar a matriz de representação interna com as saídas dos neurônios da camada Tiling. Essa matriz, representada na Tabela 5.13, será então usada para continuar a criação da rede, agora pelo Offset.

Tabela 5.13 – Matriz de representação interna gerada pela camada Tiling

Exemplos E^k (k = 1,..., 20)	**O_1^k (k = 1,..., 20)**	**O_2^k (k = 1,..., 20)**	**O_3^k (k = 1,..., 20)**	**Classe C^k (k = 1,..., 20)**
E^1	–1	1	1	1
$E^2,E^3,E^4,E^5,E^6,E^7,E^8,E^9,E^{10}$	1	1	–1	1
$E^{11},E^{12},E^{13},E^{14}$	–1	–1	–1	–1
E^{15}, E^{16}	–1	1	–1	–1
$E^{17},E^{18}, E^{19},E^{20}$	–1	–1	1	–1

Note que a matriz de representação interna gerada pela camada Tiling é uma representação booleana do conjunto de treinamento original. Essa matriz será usada para criar a parte Offset da rede. Neste ponto pode-se escolher outros que não o BCPMin para o treinamento dos neurônios. Recomenda-se, no entanto, que a camada Tiling seja construída

usando o BCPMin uma vez que este algoritmo possui bom desempenho quando usado para treinar TLUs em algoritmos que criam uma nova representação dos dados de entrada como o Tiling e o PTI.

Como o exemplo de execução anterior foi feito usando o PMR, desta vez será usado o BCPMin também para criar a parte Offset da rede. O próximo passo do algoritmo é construir a camada (segunda camada neste algoritmo) que representará a máquina de paridade, i.e. tornará o problema booleano da Tabela 5.13 em um problema de paridade. Para isso o algoritmo adiciona um neurônio que é treinado com o conjunto da Tabela 5.13.

Suponha que o treinamento desse neurônio tenha resultado no vetor de pesos $W_1 = \langle 1{,}21\ 1{,}27\ 0{,}1 \rangle$ e *bias* $\theta_1 = -0{,}06$. Como o hiperplano gerado por esse neurônio separa corretamente todas as instâncias de seu conjunto de treinamento, essa camada já é uma máquina de paridade. O próximo passo é adicionar tantos neurônios na terceira camada (segunda camada Offset) quantos forem os da segunda (primeira camada Offset), neste caso apenas um e com *threshold* igual a zero. A rede obtida é apresentada na Figura 5.6.

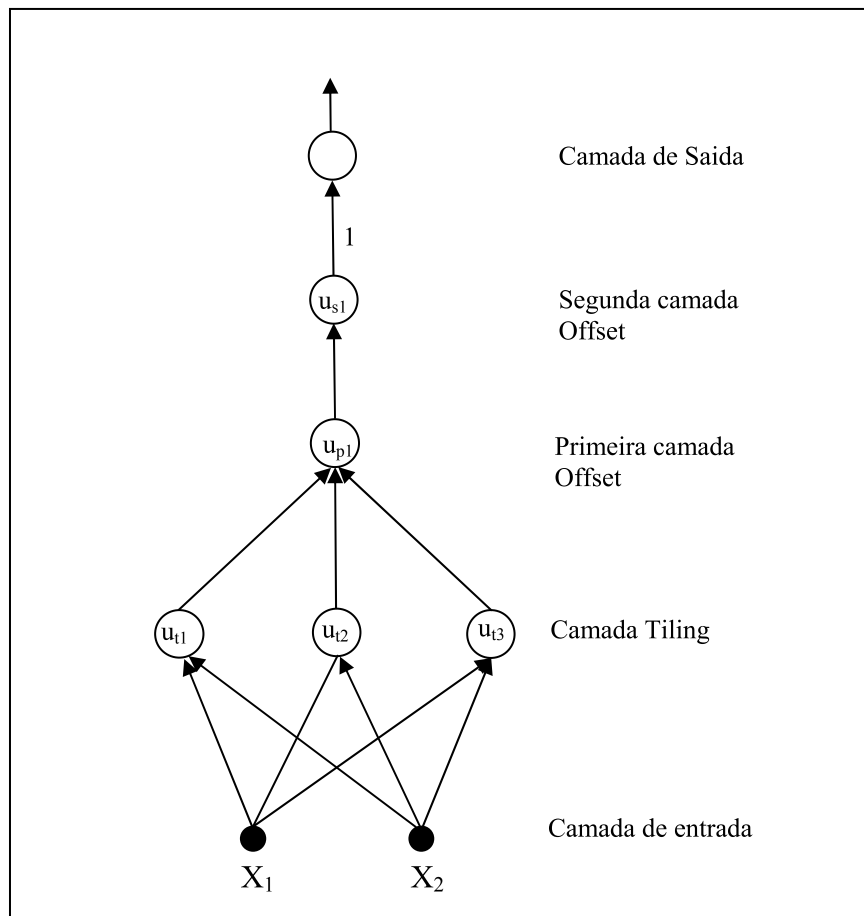


Figura 5.6 – Rede gerada pelo algoritmo OffTiling. Neurônios indexados com t estão na camada Tiling, os indexados com p e s estão na primeira e segunda camada, criadas pelo algoritmo Offset, respectivamente

5.4 O Algoritmo *Partial Target Inversion* (PTI) – Uma Versão do Algoritmo Tiling

O Partial Target Inversion (PTI) é um algoritmo neural construtivo, proposto em [Amaldi & Guenin 1998], que cria uma rede neural multi-camadas de modo semelhante ao algoritmo Tiling [Mézard & Nadal 1989] (ver Capítulo 4 Seção 4.4 para uma descrição do Tiling). O PTI foi proposto para o aprendizado de classificadores que envolvem duas classes apenas.

Assim como no Tiling, o PTI faz uso de neurônios mestres e auxiliares. Um neurônio mestre é o primeiro neurônio adicionado a cada camada; apenas um mestre é adicionado por camada. Já os neurônios auxiliares são adicionados a uma camada com o objetivo de torná-la confiável.

De maneira semelhante ao Tiling, os neurônios da camada c recebem conexões apenas dos neurônios da camada c – 1. Quando um neurônio mestre é adicionado a uma nova camada c, ele é treinado com o conjunto de treinamento E(c – 1), que é formado pelas saídas dos neurônios da camada anterior (com a adição do termo *bias* se usado o algoritmo PMR para o treinamento das TLUs). Se o treinamento do neurônio mestre resultar em um vetor de pesos que classifica corretamente todos os exemplos do conjunto de treinamento ou se o neurônio mestre da camada c não classificar mais instâncias de treinamento que o neurônio mestre da camada c – 1, o algoritmo termina.

Se algum exemplo, no entanto, foi classificado incorretamente pelo neurônio mestre e este classifica um maior número de instâncias que o neurônio mestre da camada anterior, o algoritmo adicionará neurônios auxiliares à camada em questão a fim de torná-la confiável. Quando a camada atual c for confiável, o algoritmo criará uma nova camada c + 1 com um novo neurônio mestre. Esse processo continua até que algum outro critério de parada seja atingido, tal como o número de neurônios mestres (ou auxiliares) atingiu o limite pré-estabelecido.

O processo de tornar uma camada confiável, na verdade, é a única diferença do algoritmo PTI em relação ao Tiling (consulte [Mayoraz & Aviolat 1996] para uma análise dos dois processos). Mais especificamente, a diferença está na escolha dos exemplos de treinamento que serão usados para treinar os neurônios auxiliares da camada atual. Diferente do Tiling que usa a abordagem "dividir e conquistar", a idéia no PTI para construir uma camada confiável é inverter as classes das instâncias que ativaram o último neurônio

adicionado na camada, seja ele mestre ou auxiliar, e manter inalteradas as classes das instâncias que não ativam o último neurônio.

Note que este processo pode ser visto como a aplicação da função *xor* para a determinação da classe dos exemplos que serão usados para treinar o próximo neurônio, como feito no algoritmo Offset[17] (Seção 5.3). Porém, diferente do Offset, o PTI seleciona apenas os exemplos que geram representações internas não confiáveis na camada que está sendo construída.

Como comentado anteriormente, uma representação interna em uma camada é o conjunto de saídas dos neurônios dessa camada para uma instância. Uma representação pode ser comum a uma ou mais instâncias de treinamento; desse modo, uma determinada camada intermediária pode gerar q representações internas distintas, $1 \leq q \leq n$, e n é o número de instâncias no conjunto de treinamento. Uma representação interna é confiável se todos os exemplos que a geram em uma determinada camada possuirem a mesma classe.

A justificativa para a inversão parcial das classes é que, uma vez que se deseja construir representações internas confiáveis, as entradas para as quais o i-ésimo neurônio tem saídas distintas não precisam ser distinguidas novamente e podem ter a mesma classe no novo conjunto de treinamento. Quando tenta-se tornar confiável várias representações internas simultaneamente, pode ser válido inverter a classe das entradas na c-ésima camada. De fato as classes não são importantes, a única exigência é que as entradas correspondentes a diferentes classes C^k tenham saídas diferentes para ao menos uma unidade na camada (c + 1) [Amaldi & Guenin 1998].

Uma vantagem de algoritmos neurais que constroem a rede em multicamadas, citada em [Hajnal *et al.* 1987], é que algumas funções terão uma representação bem mais compacta nesse tipo de rede do que em redes com apenas uma camada. Funções que podem ser calculadas com um número polinomial de neurônios em uma rede de três camadas, eventualmente, podem exigir uma quantidade exponencial de neurônios em uma rede de apenas uma camada.

O pseudocódigo do algoritmo PTI, mostrado em Algoritmo 5.4, é semelhante ao do algoritmo Tiling (apresentado no Capítulo 4, Figura 4.5); a principal diferença está no método *criaTreinamentoAux*(), que cria o treinamento para os neurônios auxiliares. Como mostrado no Algoritmo 5.4, esse método chama o método *naoConfiavel*() que por sua vez retorna as instâncias do conjunto de treinamento da camada anterior que geram representação interna

[17] Considerando que o Offset use a função xor como medida de distância.

não confiável na camada corrente. Essas instâncias formarão o conjunto de treinamento para o próximo neurônio auxiliar, porém, dentre essas instâncias, as que ativaram o último neurônio adicionado desta camada têm sua classe invertida.

O restante do pseudocódigo é idêntico ao do algoritmo Tiling. A rede é criada na matriz *Camadas*[][]. Essa matriz armazena os neurônios intermediários da rede. Cada linha da matriz representa uma camada da rede. Em cada linha, o elemento da primeira coluna representa o neurônio mestre desta camada e o restante os neurônios auxiliares.

Os neurônios são treinados por algum algoritmo de treinamento de TLU para duas classes, representado no pseudocódigo por meio da classe *Neuronio*. Um método dessa classe, acessado pelo método *criaTreinamentoAux*() é o método *saida*(), que cria a saída do neurônio corrente para a instância cujo índice foi passado como parâmetro.

O método *criaTreinamento*() cria o conjunto de treinamento para o próximo neurônio mestre; este método retorna a matriz de representação interna sem repetições, juntamente com as classes das instâncias, gerada pelas saídas dos neurônios da camada anterior. Note que se o algoritmo usado para treinar as TLUs for alguma variação do Perceptron, este método deve adicionar o termo *bias* ao conjunto de treinamento.

```
Class PTI
begin
{Entradas: E - conjunto de treinamento com n instâncias da forma:
            (‘xk0’, xk1, xk2,..., xkp, Ck), sendo que 1 ≤ k ≤ n
  MAX_CAM - número máximo de camadas, pré-determinado
  MAX_AUX - número máximo de neurônios auxiliares por camada, pré-determinado}
{Saída: objeto pti que representa a rede neural criada}
camada ← 0{ número de camadas intermediarias da rede}
camadaMax  ← MAX_CAM
auxiliarMax ← MAX_AUX
Camadas[camadaMax][auxiliarMax] {matriz de neurônios, onde cada linha representa uma camada
intermediária da rede }
precisaoAnterior ← 0
auxiliares ← 0

method PTI (Matrix E)
  begin
    criaCamada(E)
    if precisao() ≠ 1 then
      begin
        apaga(Camadas[camada][1])
        camada ← camada - 1
        for i ← 2 to auxiliarMax do
          apaga(Camadas[camada][i])
      end
  end

method criaCamada(Matrix E)
  begin
    camada ← camada + 1
    neuronio ← new Neuronio(E)
    Camadas[camada][1] ← neuronio {mestre}
    precisao ← precisao()
    if( precisao > precisaoAnterior and camada ≤ camadaMax and precisao ≠ 1) then
      begin
        precisaoAnterior ← precisao
        criaAuxiliares(E)
        criaCamada( criaTreinamento(E) )
      end
  end

method criaAuxiliare (Matrix E)
  begin
    auxiliares ← 1
    while (not confiavel() and auxiliares ≤ auxililaresMax) do
      begin
        novoNeurnonio  ← new Neuronio(criaTreinamentoAux(E))
        auxiliares ← auxiliares +1
        Camadas[camada][auxiliares] ← novoNeuronio   {auxiliares}
      end
  end

method Matrix criaTreinamentoAux(Matrix E)
  begin
    A ← naoConfiavel(E)
    for k ← 1 to nA do
      if Camadas[camada][auxiliares].saida(k) = 1 then
        CkA ← −1CkA
    return A
  end
end {Class}
```

Algoritmo 5.4 – Pseudocódigo do algoritmo PTI

5.4.1 Exemplo de Execução do PTI Utilizando o PMR como Algoritmo para o Treinamento de TLUs

Para exemplificar o funcionamento do algoritmo PTI será usado o conjunto de treinamento que representa a função Paridade-3, apresentado na Tabela 5.14. Como, para este exemplo, será usado o PMR como algoritmo de treinamento das TLUs, é necessário a adição do termo *bias* ao conjunto de treinamento.

Tabela 5.14 - Conjunto de treinamento que representa o problema Paridade-3 com a adição do termo *bias*

Exemplos E^k (k = 1,..., 8)	**X_0(*bias*)**	**X_1**	**X_2**	**X_3**	**Classe C^k (k = 1,..., 8)**
E^1	1	1	1	1	1
E^2	1	1	1	–1	–1
E^3	1	1	–1	1	–1
E^4	1	1	–1	–1	1
E^5	1	–1	1	1	–1
E^6	1	–1	1	–1	1
E^7	1	–1	–1	1	1
E^8	1	–1	–1	–1	–1

O primeiro passo do algoritmo é criar um neurônio mestre na primeira camada para tentar classificar os exemplos de treinamento. Como o conjunto de treinamento da Tabela 5.14 não é linearmente separável, o primeiro neurônio mestre (u_{11}) não consegue encontrar um vetor de pesos que classifica todos os exemplos corretamente. Suponha, entretanto, que o neurônio tenha aprendido um vetor de pesos ótimo para o conjunto Paridade-3, e que este vetor seja $W_{11} = \langle 1\ \ 1\ \ -1\ \ 1 \rangle$. Este vetor comete apenas dois erros de classificação, nos exemplos E^3 e E^6 do conjunto de treinamento.

Como o primeiro mestre não classificou todas as instâncias corretamente, o próximo passo é tornar a camada corrente confiável. A Tabela 5.15 mostra o conjunto de treinamento do primeiro neurônio mestre u_{11} (conjunto original), as saídas deste neurônio $O_{u_{11}}$ e as novas classes das instâncias do conjunto de treinamento, que serão usadas para treinamento do primeiro neurônio auxiliar.

Repare que as classes para o conjunto de treinamento do segundo neurônio da primeira camada ($C_{u_{12}}$), são iguais às classes do conjunto de treinamento do primeiro neurônio (conjunto original) quando a saída deste ($O_{u_{11}}$) é –1 e são inversas quando $O_{u_{11}}$ é igual a 1.

Tabela 5.15 – Conjunto de treinamento com a saída do neurônio mestre mais as classes atualizadas para o próximo conjunto

Exemplo E^k (k = 1,..., 8)	**X_0(*bias*)**	**X_1**	**X_2**	**X_3**	**Classe**	**$O^k_{u_{11}}$ (k = 1,..., 8)**	**$C^k_{u_{12}}$ (k = 1,..., 8)**
E^1	1	1	1	1	1	1	–1
E^2	1	1	1	–1	–1	–1	–1
E^3	1	1	–1	1	–1	1	1
E^4	1	1	–1	–1	1	1	–1
E^5	1	–1	1	1	–1	–1	–1
E^6	1	–1	1	–1	1	–1	1
E^7	1	–1	–1	1	1	1	–1
E^8	1	–1	–1	–1	–1	–1	–1

Com as classes do conjunto de treinamento atualizadas o próximo passo é criar o primeiro neurônio auxiliar da primeira camada. Note que todos os exemplos devem ser considerados uma vez que nenhum protótipo confiável foi obtido.

Suponha que o treinamento do primeiro neurônio auxiliar u_{12} tenha gerado o vetor de pesos $W_{12} = \langle -1\ -1\ 1\ -1 \rangle$. Esse vetor classifica corretamente todos os exemplos do conjunto de treinamento com exceção de E^3. A adição de u_{12}, no entanto, ainda não tornou a camada confiável, pois como pode ser visto na matriz de representações internas representada na Tabela 5.16, a representação $\langle 1\ -1 \rangle$ está associada a ambas as classes, quando deveria estar associada a apenas uma.

Tabela 5.16 – Atual matriz de representação interna da primeira camada

Exemplos E^k (k = 1,..., 8)	**$O^k_{u_{11}}$ (k = 1,..., 8)**	**$O^k_{u_{12}}$ (k = 1,..., 8)**	**Classe C^k (k = 1,..., 8)**
E^1	1	–1	1
E^2	–1	–1	–1
E^3	1	–1	–1
E^4	1	–1	1
E^5	–1	–1	–1
E^6	–1	1	1
E^7	1	–1	1
E^8	–1	–1	–1

Note que as representações $\langle -1\ -1 \rangle$, e $\langle -1\ 1 \rangle$ são confiáveis, pois cada uma delas está associada apenas a uma mesma classe e, portanto, os exemplos que geraram essas representações internas não farão parte do conjunto de treinamento para o novo neurônio.

Como a camada atual ainda não é confiável o algoritmo deve seguir adicionando neurônios auxiliares até que a camada se torne confiável. O conjunto de treinamento para o próximo neurônio auxiliar (u_{13}) é mostrado na Tabela 5.17.

Tabela 5.17 – Conjunto de treinamento do segundo neurônio auxiliar u_{13} da primeira camada

Exemplos E^k (k = 1, 3, 4, 7)	**X_0(*bias*)**	**X_1**	**X_2**	**X_3**	**Classe C^k (k = 1, 3, 4, 7)**
E^1	1	1	1	1	–1
E^3	1	1	–1	1	1
E^4	1	1	–1	–1	–1
E^7	1	–1	–1	1	–1

O conjunto da Tabela 5.17 é linearmente separável e o vetor $W_{13} = \langle -1\ 1\ -1\ 1 \rangle$ classifica todos os exemplos corretamente. A Tabela 5.18 mostra a matriz de representações internas referente à primeira camada; note que agora todas as representações são confiáveis e, portanto, a camada é confiável.

Tabela 5.18 – Matriz de representação interna da primeira camada

Exemplos E^k (k = 1,..., 8)	**$O^k_{u_{11}}$ (k = 1,..., 8)**	**$O^k_{u_{12}}$ (k = 1,..., 8)**	**$O^k_{u_{13}}$ (k = 1,..., 8)**	**Classe C^k (k = 1,..., 8)**
E^1	1	–1	–1	1
E^2	–1	–1	–1	–1
E^3	1	–1	1	–1
E^4	1	–1	–1	1
E^5	–1	–1	–1	–1
E^6	–1	1	–1	1
E^7	1	–1	–1	1
E^8	–1	–1	–1	–1

A matriz de representação interna da primeira camada, sem instâncias repetidas e com a adição do termo *bias*, pois o algoritmo está usando o PMR, será o conjunto de treinamento do neurônio mestre da segunda camada. Como este conjunto é linearmente separável o neurônio mestre da segunda camada encontra um vetor de pesos $W_{21} = \langle 1\ 1\ 1\ -1 \rangle$ que classifica todos os exemplos corretamente. A rede PTI que representa o conceito Paridade-3 é ilustrada na Figura 5.7.

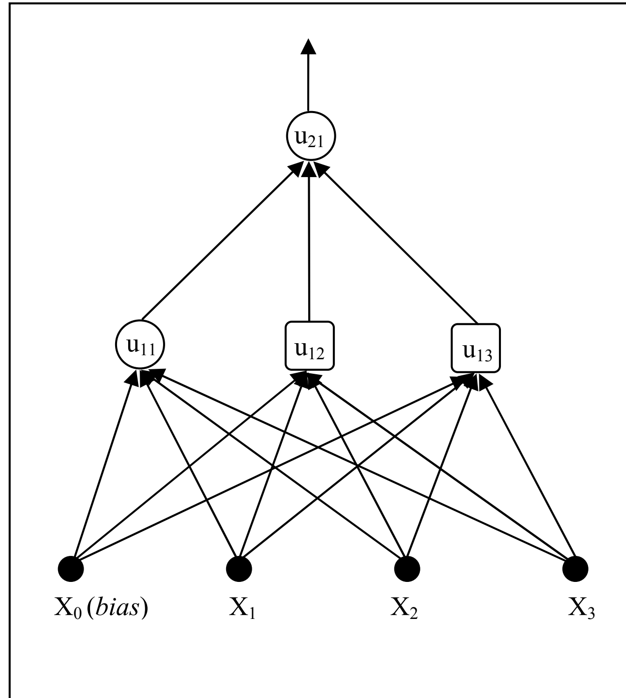


Figura 5.7 – Rede PTI que representa o conceito de Paridade-3

5.4.2 Exemplo de Execução do Algoritmo PTI Utilizando o BCPMin como Algoritmo para o Treinamento de TLUs

Para exemplificar a execução do algoritmo PTI com o BCPMin será usado o conjunto da Tabela 5.14 que representa o conceito de paridade-3, sem a adição do termo *bias*.

O algoritmo começa treinando o primeiro neurônio u_{11} da primeira camada; este neurônio é o mestre desta camada e é treinado com o conjunto de treinamento original. Suponha que o treinamento deste neurônio tenha obtido o vetor de pesos $W_{11} = \langle -0,4\ 1,15\ 0,8 \rangle$ e termo *bias* $\theta_{11} = 0$. Essa solução comete erro somente nas instâncias E^4 e E^5. Note na Tabela 5.19 que somente onde ocorreram os erros a classe para treinar o próximo neurônio será igual a 1.

Tabela 5.19 – Conjunto com as instâncias de treinamento com as classes originais e as classes para treinar o próximo neurônio

Exemplos E^k (k = 1,..., 8)	**X_1**	**X_2**	**X_3**	**Classe C^k (k = 1,..., 8)**	**Nova Classe**
E^1	1	1	1	1	–1
E^2	1	1	–1	–1	–1
E^3	1	–1	1	–1	–1
E^4	1	–1	–1	1	1
E^5	–1	1	1	–1	1
E^6	–1	1	–1	1	–1
E^7	–1	–1	1	1	–1
E^8	–1	–1	–1	–1	–1

Suponha que o treinamento do primeiro neurônio auxiliar (u_{12}) com o conjunto de treinamento da Tabela 5.19, com as novas classes, tenha resultado no vetor de pesos $W_{12} = \langle -0,87\ 0,82\ 0,57 \rangle$ e *bias* $\theta_{12} = -1,69$. Este hiperplano comete apenas um erro, na instância E^4. Como as representações geradas pelas instâncias E^2, E^3 E^4 e E^8 não são confiáveis, estas instâncias serão usadas para formar o próximo conjunto de treinamento, mostrado na Tabela 5.20.

Tabela 5.20 – Conjunto de treinamento para o segundo neurônio auxiliar da primeira camada

Exemplos E^k (k = 2, 3, 4, 8)	**X_1**	**X_2**	**X_3**	**Classe C^k (k = 2, 3, 4, 8)**
E^2	1	1	–1	–1
E^3	1	–1	1	–1
E^4	1	–1	–1	1
E^8	–1	–1	–1	–1

O segundo neurônio auxiliar (u_{13}) com $W_{13} = \langle 0,72\ -0,63\ -0,63 \rangle$ e *bias* $\theta_{13} = -1,36$ classifica todos os exemplo da Tabela 5.20 corretamente, e como as representações tornaram-se confiáveis, a camada atual é confiável. O próximo passo é criar o mestre da segunda camada, que será treinado com as representações internas geradas pela primeira camada, como mostra a Tabela 5.21.

Tabela 5.21 – Matriz de representação interna da primeira camada

$O_{u_{11}}$	**$O_{u_{12}}$**	**$O_{u_{13}}$**	**Classe**
1	–1	–1	1
–1	–1	–1	–1
–1	–1	1	1
1	1	–1	–1

Como o conjunto é linearmente separável o mestre da segunda camada (u_{21}) classifica todas as instâncias corretamente com vetor de pesos $W_{21} = \langle 1,44\ -1,34\ 1,21 \rangle$ e *bias* $\theta_{21} = 0$. A rede gerada é ilustrada na Figura 5.8, Note que a única diferença em relação à arquitetura da rede gerada no exemplo de execução anterior (Figura 5.7) é que desta vez o termo *bias* não é adicionado ao conjunto de treinamento.

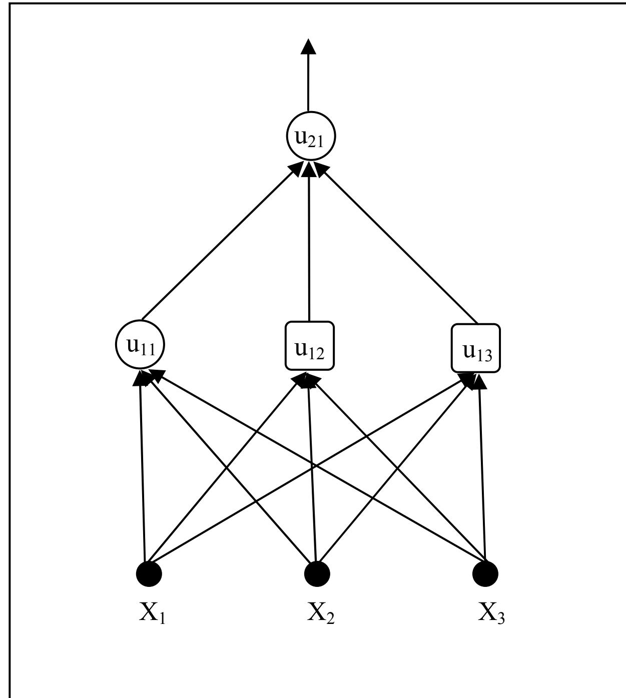


Figura 5.8 – Conceito Paridade-3 aprendido por uma rede PTI com o algoritmo BCPMin

5.5 O Algoritmo Perceptron Cascade

O Perceptron Cascade [Burgess 1994] é um algoritmo neural construtivo que combina a arquitetura de uma rede criada pelo algoritmo Cascade Correlation [Fahlman & Lebiere 1990] com as regras de correção de erro usadas no algoritmo Upstart [Frean 1990a]. O treinamento dos neurônios, diferente do Cascade Correlation, é feito usando o Perceptron ou um de seus derivados.

O fato do algoritmo usar TLUs lineares[18] aliado às regras de correção de erro do Upstart, além de diminuir o tempo de treinamento da rede em relação ao Cascade Correlation com o Quickprop [Fahlman 1988][19], também garante convergência da rede para qualquer conjunto de treinamento consistente com entradas reais. Essa convergência não é garantida no Cascade Correlation.

A rede no Perceptron Cascade começa a ser construída pelo neurônio de saída u_0, como no Upstart, e os neurônios intermediários são adicionados como no Cascade Correlation, no qual cada novo neurônio é conectado ao neurônio de saída e recebe conexões dos neurônios da camada de entrada e de todos os neurônios intermediários previamente adicionados.

---

[18] TLUs treinada com algum algoritmo de tempo polinomial; e.g. variante do perceptron ou o BCPMin.
[19] Proposta original do Cascade Correlation.

Um neurônio intermediário é adicionado com o objetivo de diminuir os erros cometidos pela rede. Esses erros podem ser erros de positivo (onde a saída da rede é negativa e a classe do exemplo em questão é positiva) ou erros de negativo (caso em que a saída da rede é positiva e a classe do exemplo tratado é negativa ou nula).

Cada vez que um novo neurônio intermediário é adicionado à rede o neurônio de saída é retreinado com o novo conjunto de treinamento, i.e. com a saída do novo neurônio incluída. O próximo passo é verificar se a rede ainda comete erros, e qual tipo de erro é mais freqüente, erro de positivo ou erro de negativo.

O conjunto de treinamento usado para treinar o novo neurônio u_n será determinado pelo tipo de erro mais freqüente da rede após a inserção do último neurônio u_{n-1}. Como cada novo neurônio pode ser alocado para corrigir tipos de erros diferentes (de positivo ou de negativo), o conjunto usado para treiná-lo deve ser adaptado de acordo com o tipo de erro a ser corrigido.

Se o neurônio u_n for alocado para corrigir erros de positivo, o conjunto de treinamento que será usado para treiná-lo deverá ser formado pelos exemplos de classe negativa ou nula mais os exemplos em que a rede atual comete erros de positivo (i.e. exemplos de classe positiva que a rede classifica como negativo ou nulo).

Se, no entanto, o novo neurônio u_n for alocado para corrigir erros de negativo, o conjunto de treinamento que deverá ser usado será formado pelos exemplos de classe positiva mais os exemplos de classe negativa que a rede classifica como positivo (i.e. erro de negativo). Nesse caso, porém, todas as instâncias selecionadas têm suas classes invertidas.

Após a inserção do neurônio u_n e o treinamento do neurônio de saída u_0, se o número de erros da rede não diminuir o peso referente ao neurônio u_n no vetor de pesos do neurônio de saída (i.e. último peso de W_0), é trocado por zero a fim de garantir convergência. Apesar de não ter corrigido nenhum erro este neurônio ainda será útil, pois a sua saída será usada na entrada do próximo neurônio.

O Algoritmo 5.5 apresenta o pseudocódigo orientado a objeto do algoritmo Perceptron Cascade. A classe *PerceptronCascade* utiliza objetos da classe *Neuronio* que, quando criados com o conjunto de treinamento corrente, geram um vetor de pesos por meio do uso de algum método tal como o PMR ou o BCPMin. Os neurônios intermediários são armazenados no vetor *Camadas*[], que representa as camadas intermediárias da rede, e o neurônio de saída corresponde ao objeto de nome saída.

O método *precisao*() retorna a precisão da rede atual e os métodos *erroNegativo*() e *erroPositivo*() retornam o número de erros de negativo e de positivo cometidos pela rede

atual, respectivamente. O algoritmo alocará um novo neurônio intermediário para corrigir o tipo de erro de maior freqüência.

Se o erro mais freqüente for o erro de positivo o método *criaTreinamentoPositivo*() é chamado; este método retorna o conjunto de treinamento que será usado para treinar o novo neurônio de modo que este corrija erros de positivo. Este método seleciona, dentre o conjunto de treinamento atual, instâncias com classe negativa mais as instâncias que provocam erro de positivo na rede atual.

O método *criaTreinamentoNegativo*(), por sua vez, retorna o conjunto que será usado para o treinamento de um novo neurônio, agora alocado para corrigir erros de negativo. Tendo como base o conjunto de treinamento atual, este método seleciona as instâncias positivas mais as instâncias que provocam erro de negativo na rede atual e em seguida inverte as classes de todas essas instâncias selecionadas.

O método *saidaU*() da classe *Neuronio*, retorna a saída do neurônio corrente para a instância de treinamento cujo índice é passado como parâmetro. Note que o método *saidaU*(), retorna 0 como valor inativo ao invés de –1 como retornava o método *saida*() usado em outros algoritmos. A saída para todas as instâncias são adicionadas, como um novo atributo, ao conjunto de treinamento pelo método *criaTreinamentoSaida*().

Utilizando o atual conjunto E, o neurônio de saída é retreinado por meio do método Neuronio e novamente os métodos *erroPositivo*() e *erroNegativo*() são chamados. Se com a adição deste último neurônio a rede piorar a precisão da classificação, o algoritmo pára e o último neurônio adicionado é retirado da rede por meio do método *apaga*().

```
Class PerceptronCascade
begin
{Entradas: E - conjunto de treinamento com n instâncias da forma:
            (‘x^k_0’, x^k_1, x^k_2,..., x^k_p, C^k), sendo que 1 ≤ k ≤ n
 MAX - número máximo de neurônios intermediários, pré-determinado}
{Saída: objeto perceptronCascade que representa a rede neural criada}
 nroCamadas ← 1
 CamadaMax ← MAX
 Camadas[camadaMax] {vetor que representa as camadas intermediárias da rede}

method PerceptronCascade(Matrix E)
  begin
    saida ← new Neuronio(E)
    precisao ← precisao()
    precisaoAnterior ← 0
    erroPositivo ← erroPositivo()
    erroNegativo ←  erroNegativo()
    while precisao > precisaoAnterior and precisão ≠ 1 and nroCamadas ≤ CamadaMax) do
      begin
        precisaoAnterior ←  precisao
        if erroPositivo  ≥ erroNegativo then
          begin
            Epos ← criaTreinamentoPositivo()
            Camadas[nroCamadas] ← new Neuronio(Epos)
          end
        else if erroNegativo > erroPosivivo then
            begin
              Eneg ← criaTreinamentoNegativo()
              Camadas[nroCamadas] ← new Neuronio(Eneg)
            end
        E ← criaTreinamentoSaida(E)
        saidaAnt ← saida
        saida ←  Neuronio(E)
        erroPositivo ← erroPositivo()
        erroNegativo ←  erroNegativo()
        precisao ← precisao()
        nroCamadas ← nroCamadas  + 1
      end

      if precisao ≤ precisaoAnterior then
        begin
          apaga(Camadas[nroCamadas])
          camada ← camada – 1
          saida ← saidaAnt
        end
  end

method Matrix criaTreinamentoSaida(Matrix E)
  begin
    A ← E
      for k ← 1 to n do
         A[k][p+1] ← Camadas[nroCamada].saidaU(k)

    return A
  end
end  {Class}
```

Algoritmo 5.5 – Pseudocódigo do algoritmo Perceptron Cascade

5.5.1 Exemplo de Execução do Algoritmo Perceptron Cascade Usando o PMR

Será usado, novamente, o conjunto de dados que representa o problema de Paridade-3 (Tabela 5.14) para exemplificar o funcionamento do algoritmo Perceptron Cascade.

O primeiro passo do algoritmo é treinar o primeiro neurônio, o neurônio de saída u_0, com o conjunto de treinamento da Tabela 5.14. Suponha que o treinamento deste neurônio tenha resultado no vetor de pesos $W_0 = \langle 1\ 1\ 1\ -1 \rangle$; este vetor define um hiperplano que comete um erro de positivo (em E^7) e um erro de negativo (em E^2) no conjunto de treinamento original.

Sempre que a rede cometer o mesmo número de erros de positivo e de negativo, será alocado um neurônio para corrigir os erros de positivo. Desse modo, o conjunto de treinamento para o neurônio u_1 será formado pelas instâncias de classe negativa (E^2, E^3, E^5 e E^8) mais a instância relativa ao erro de positivo cometido pela rede (E^7) (ver Tabela 5.22).

Tabela 5.22 – Conjunto de treinamento para que o primeiro neurônio corrija erros de positivo

Exemplos E^k (k = 2, 3, 5, 7, 8)	**X_0 (*bias*)**	**X_1**	**X_2**	**X_3**	**Classe C^k (k = 2, 3, 5, 7, 8)**
E^2	1	1	1	–1	–1
E^3	1	1	–1	1	–1
E^5	1	–1	1	1	–1
E^7	1	–1	–1	1	1
E^8	1	–1	–1	–1	–1

Um vetor de pesos que classifica corretamente essas instâncias é o vetor $W_1 = \langle -1\ -1\ -1\ 1 \rangle$. Agora as saídas desse neurônio quando calculadas com o conjunto da Tabela 5.22, são adicionadas ao conjunto de treinamento como um novo atributo[20] X_4 (ver Tabela 5.23), e o neurônio de saída u_0 é retreinado.

Tabela 5.23 – Conjunto de treinamento original com adição de uma entrada referente ao neurônio adicionado

Exemplos E^k (k = 1,..., 8)	**X_0 (*bias*)**	**X_1**	**X_2**	**X_3**	**X_4**	**Classe C^k (k = 1,..., 8)**
E^1	1	1	1	1	0	1
E^2	1	1	1	–1	0	–1
E^3	1	1	–1	1	0	–1
E^4	1	1	–1	–1	0	1
E^5	1	–1	1	1	0	–1
E^6	1	–1	1	–1	0	1
E^7	1	–1	–1	1	1	1
E^8	1	–1	–1	–1	0	–1

[20] Foi decidido manter a terminologia original do autor que nomeia a saída de cada novo neurônio introduzido na rede, por atributo.

Suponha que com a Tabela 5.23 como conjunto de treinamento do neurônio u_0, este tenha encontrado o vetor de pesos $W_0 = \langle 1\ 1\ 1\ -1\ 5 \rangle$. A rede atual ($u_0$ e u_1), comete um erro de negativo em E^2. Então um novo neurônio u_2, deve ser adicionado para corrigi-lo, a Tabela 5.24 ilustra o conjunto de treinamento usado para treiná-lo.

Tabela 5.24 – Conjunto para o treinamento do neurônio u_2

Exemplos E^k (k = 1, 3, 4, 6, 7)	**X_0 (*bias*)**	**X_1**	**X_2**	**X_3**	**X_4**	**Classe C^k (k = 1, 3, 4, 6, 7)**
E^1	1	1	1	1	0	–1
E^3	1	1	1	–1	0	1
E^4	1	1	–1	–1	0	–1
E^6	1	–1	1	–1	0	–1
E^7	1	–1	–1	1	1	–1

Um vetor de pesos que classifica corretamente todas as instâncias de treinamento do conjunto na Tabela 5.24 pode ser $W_2 = \langle -1\ 1\ 1\ -1\ 0 \rangle$. Como mais um neurônio foi adicionado, o próximo passo é retreinar o neurônio de saída, com o conjunto de treinamento da Tabela 5.25; repare que o atributo X_5 é a saída do neurônio u_2 para os exemplos da Tabela 5.23.

Tabela 5.25 – Conjunto original com a adição das entradas referentes aos dois neurônios adicionados

Exemplos E^k (k = 1,..., 8)	**X_0 (*bias*)**	**X_1**	**X_2**	**X_3**	**X_4**	**X_5**	**Classe C^k (k = 1,..., 8)**
E^1	1	1	1	1	0	0	1
E^2	1	1	1	–1	0	1	–1
E^3	1	1	–1	1	0	0	–1
E^4	1	1	–1	–1	0	0	1
E^5	1	–1	1	1	0	0	–1
E^6	1	–1	1	–1	0	0	1
E^7	1	–1	–1	1	1	0	1
E^8	1	–1	–1	–1	0	0	–1

O neurônio de saída após ser retreinado com o conjunto de treinamento da Tabela 5.25, define o vetor de pesos $W_0 = \langle 1\ 1\ 1\ -1\ 5\ -4 \rangle$. Esse vetor classifica corretamente todos os exemplos de treinamento da Tabela 5.25, repare que o conjunto de treinamento da Tabela 5.25 possui todos os estados possíveis da rede. Desse modo se o neurônio de saída não cometer erros, a rede converge.

A rede obtida está ilustrada na Figura 5.9 e os respectivos vetores de pesos são: $W_0 = \langle 1\ 1\ 1\ -1\ 5\ -4 \rangle$ (neurônio saída), $W_1 = \langle -1\ -1\ -1\ 1 \rangle$ e $W_2 = \langle -1\ 1\ 1\ -1\ 0 \rangle$.

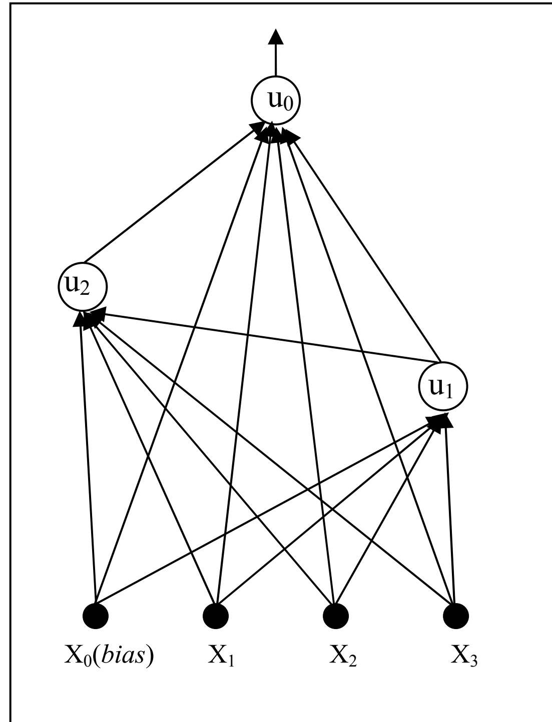


Figura 5.9 – Conceito Paridade-3 representado por uma rede Perceptron Cascade

5.5.2 Exemplo de Execução do Algoritmo Perceptron Cascade Usando o BCPMin

Como mencionado anteriormente o primeiro passo do algoritmo é treinar o neurônio de saída. Suponha que este treinamento tenha resultado no vetor de pesos $W_1 = \langle -1{,}36\ -1{,}21\ 1{,}34 \rangle$ e *bias* $\theta_1 = 0$. Este vetor de pesos com o *bias* gera um hiperplano que comete dois erros, um erro de positivo na instância E^6 e um erro de negativo na instância E^2. Dessa forma, o primeiro erro que o algoritmo tenta corrigir é o erro de positivo.

Para corrigir este erro de positivo o algoritmo treina um neurônio com as instâncias negativas mais a instância que provocou o erro. O conjunto de treinamento deste novo neurônio está mostrado na Tabela 5.26.

Tabela 5.26 – Conjunto de treinamento para o neurônio alocado para corrigir erro de positivo

Exemplos E^k (k = 2, 3, 5, 7, 8)	**X_1**	**X_2**	**X_3**	**Classe C^k (k = 2, 3, 5, 7, 8)**
E^2	1	1	–1	–1
E^3	1	–1	1	–1
E^5	–1	1	1	–1
E^7	–1	–1	1	1
E^8	–1	–1	–1	–1

O treinamento do neurônio com o conjunto da Tabela 5.26 resulta no vetor de pesos $W_2 = \langle -0{,}98\ -0{,}93\ 1{,}07 \rangle$ e *bias* $\theta_2 = -2{,}05$. Com a adição deste neurônio, o neurônio saida deve ser retreinado, o conjunto usado para retreinar o neurônio de saída será o conjunto original mais a saída do neurônio recém adicionado, como mostra a Tabela 5.27.

Tabela 5.27 – Conjunto para retreinar o neurônio de saída após a inserção do primeiro neurônio intermediário

Exemplo E^k (k = 1,..., 8)	**X_1**	**X_2**	**X_3**	**X_4**	**Classe C^k (k = 1,..., 8)**
E^1	1	1	1	0	1
E^2	1	1	–1	0	–1
E^3	1	–1	1	0	–1
E^4	1	–1	–1	0	1
E^5	–1	1	1	0	–1
E^6	–1	1	–1	0	1
E^7	–1	–1	1	1	1
E^8	–1	–1	–1	0	–1

Após retreinado, o neurônio de saída tem o peso $W_0 = \langle 7{,}64\ -8{,}14\ -9{,}21\ 1{,}66 \rangle$ e *bias* $\theta_0 = 0{,}9$; esta configuração porém, ainda comete dois erros no conjunto de treinamento, nas instâncias E^4 e E^5. Como a precisão permanece a mesma (0,75), o algoritmo pára retira o último neurônio adicionado e retomando a configuração da última rede bem sucedida, no caso um único neurônio com $W_1 = \langle -1{,}36\ -1{,}21\ 1{,}34 \rangle$ e *bias* $\theta_1 = 0$.

5.6 O Algoritmo Sequential

O Sequential, proposto por [Marchant *et al.* 1990], é um algoritmo neural construtivo que cria a rede neural de maneira diferenciada das anteriores. Os neurônios adicionados durante a construção da rede não são treinados para classificarem um número máximo de instâncias de treinamento. Ao invés disso, a idéia é treinar neurônios para excluírem o maior número possível de instâncias de treinamento pertencentes a uma mesma classe.

Uma rede neural criada pelo algoritmo Sequential possui apenas uma camada intermediária com conexões com a camada de entrada e com a camada de saída, que possui um único neurônio. As conexões entre a camada intermediária e o neurônio da camada de saída possuem pesos e o neurônio de saída possui um *bias*. A camada intermediária é criada adicionando-se neurônios seqüencialmente, sendo que cada neurônio adicionado tem o

objetivo de procurar um hiperplano que separe o maior número[21] de instâncias pertencentes a uma mesma classe do resto do conjunto de treinamento.

Achado esse hiperplano, as instâncias que este neurônio separou são excluídas do conjunto de treinamento. Então um novo neurônio é adicionado à camada intermediária com o mesmo objetivo do anterior: definir um hiperplano que separe o maior número de instâncias de treinamento de uma mesma classe. O processo chega ao fim quando restarem somente instâncias pertencentes a uma única classe no conjunto de treinamento.

Ao final desse processo os neurônios intermediários geram uma matriz de representação interna confiável que representa o conjunto de treinamento. Como a camada intermediária é confiável a rede converge por meio do uso dos pesos nas conexões entre os neurônios da camada intermediária e o neurônio da camada de saída e com o uso de um *bias* no neurônio de saída. O peso da conexão entre o neurônio intermediário i e o neurônio de saída é obtido por meio da equação (5.6).

$$\lambda_i = 2^{n-i+1} \tag{5.6}$$

na qual λ_i representa o peso da conexão entre o i-ésimo neurônio da camada intermediária e o neurônio de saída, e n é o número total de neurônios na camada intermediária.

O *bias* do neurônio de saída depende dos pesos das conexões, e é definido por meio da equação:

$$\theta = \sum_{i=1}^{n} (-1)^{i+1} \lambda_i + (-1)^n \tag{5.7}$$

Como o Sequential cria a camada intermediária por meio da exclusão seqüencial de conjuntos de instâncias de treinamento de mesma classe do conjunto original, cada neurônio adicionado deve ser treinado com o objetivo de excluir o máximo número de instâncias de treinamento com uma mesma classe. Sendo assim métodos que procuram por um hiperplano que classifique corretamente o maior número de instâncias de treinamento, como o perceptron e suas variantes ou o BCPMin, não poderão ser usados.

No que segue são apresentados e discutidos dois métodos para a construção de uma rede Sequential, um utilizando o algoritmo BCPMax para o treinamento dos neurônios e o outro utilizando o algoritmo IncLp, respectivamente.

[21] Nem sempre o hiperplano encontrado separa o maior número possível de instâncias de treinamento.

5.6.1 O Algoritmo Sequential com o BCPMax

Um algoritmo eficiente para o treinamento de neurônios que funciona com base na exclusão do maior número de exemplos de treinamento é o BCPMax proposto em [Poulard 1995] e discutido no Capítulo 3, Seção 3.5. O algoritmo BCPMax acha o maior conjunto de instâncias (positivas ou negativas) linearmente separáveis do resto do conjunto de treinamento cada vez que é executado.

Como o BCPMax encontra um termo *bias* que colabora na separação das instâncias de treinamento, todo neurônio da camada intermediária treinado com o BCPMax terá um termo *bias*, bem como um termo ε que também é encontrado pelo BCPMax para separar as instâncias de treinamento.

O termo ε é usado pelo BCPMax com o objetivo de verificar se para um determinado hiperplano, o hiperplano seu ortogonal separa mais instâncias de treinamento. Se, de fato o hiperplano ortogonal ao hiperplano encontrado separar mais instâncias de treinamento, o valor de ε será –1 e W será invertido (–W).

Em uma rede Sequential, se o primeiro neurônio adicionado à camada intermediária possuir ε igual a –1 significa que o vetor de pesos W associado a esse neurônio deve ser invertido. Esse fato modifica o cálculo do *bias* do neurônio da camada de saída que passa a ser obtido como especificado na equação (5.8).

$$\theta = \sum_{i=1}^{n} (-1)^{i} \lambda_{i} + (-1)^{n+1} \tag{5.8}$$

O Algoritmo 5.6 descreve o algoritmo Sequential usando o BCPMax para treinar os neurônios. O método construtor da classe BCPMax quando chamado, gera o vetor de pesos W e os termos ε e *bias* referente a esse neurônio.

No pseudocódigo do Algoritmo 5.6, a camada intermediaria da rede Sequential é construída no laço while, que termina quando o conjunto de treinamento E1 for linearmente separável, indicado pelo método *isLS*() da classe BCPMax.

O método *retiraExcluidos*() recebe como parâmetro o conjunto de treinamento E1 atual e o índice para o neurônio atual. Esse método acessa o método *getIndexes*() da classe BCPMax, esse método retorna um vetor com os índices dos elementos do conjunto de treinamento excluídos por esse neurônio. Então o método *retiraExcluidos*() de posse desse vetor, não copia os índices nele presentes para o conjunto que será retornado newE.

```
Class Sequential
begin
{Entradas: E - conjunto de treinamento com n instâncias da forma:
             (‘xk0’, xk1, xk2,..., xkp, Ck), sendo que 1 ≤ k ≤ n
  MAX - número máximo de neurônios intermediários, pré-determinado}
{Saída: objeto sequential que representa a rede neural criada}
CamadaMax ← MAX
Camada[CamadaMax]

method Sequential(Matrix E)
  begin
    cont  ← 1 {contador de neurônios}
    Camada[cont] ← new BCPMax(E1)
    while (not Camada[cont].isLS()) do
      begin
        E1 ← retiraExcluidos(E1, cont)
        cont ← cont +1
        Camada[cont] ← new BCPMax(E1)
      end

     indNeuronio ← 1
     nroNeuronios ← cont                    {cálculo dos pesos das conexões}
     while ( nroNeuronios > 0 ) do
       begin
         pesos[indNeuronio] ← 2^nroNeuronios
         indNeuronio ← indNeuronio + 1
         nroNeuronios ← nroNeuronios - 1
       end

     aux ← Camada[1].getEps()
     for i ← 1 to cont do            {cálculo do bias do neurônio de saída}
       begin
         bias ← bias + aux * pesos[i]
         aux ← aux * -1
       end
     bias ← bias + aux
  end  {method}

method Matrix retiraExcluidos(Matrix A, Integer n)
  begin
    j ← 1
    IndE ← Camada[n].getIndexes()
    for i ← 1 to |A| do
      begin
        if ( j < |IndE| and i = IndE^j ) then
          j ← j +1
        else
          newE ← A^i
      end
    return newE
  end
 end   {Class}
```

Algoritmo 5.6 – Pseudocódigo do algoritmo Sequential usando o BCPMax

5.6.1.1 Exemplo de Execução do Sequential em um Conjunto Não Linearmente Separável, usando o BCPMax

A fim de exemplificar o funcionamento do algoritmo Sequential com o BCPMax, será usado o conjunto de treinamento Paridade-3, representado na Tabela 5.28.

Tabela 5.28 – Conjunto de exemplos que representa o problema Paridade-3

Exemplos E^k (k = 1,...,8)	**X_1**	**X_2**	**X_3**	**Classe C^k (k = 1,...,8)**
^{1}	1	1	1	1
E^2	1	1	–1	–1
E^3	1	–1	1	–1
E^4	1	–1	–1	1
E^5	–1	1	1	–1
E^6	–1	1	–1	1
E^7	–1	–1	1	1
E^8	–1	–1	–1	–1

O primeiro passo do algoritmo é treinar o primeiro neurônio u_1 usando o BCPMax. Suponha que o treinamento desse neurônio resulte no vetor $W_1 = \langle 0{,}02\ 0{,}1\ 0{,}16 \rangle$, e termo *bias* $\theta_1 = -0{,}26$ e $\varepsilon_1 = 1$. O hiperplano separa somente a instância E^1 das demais. Essa instância é retirada do conjunto de treinamento e um segundo neurônio é treinado.

Suponha que o treinamento do segundo neurônio u_2 usando o BCPMax, resulte no vetor de pesos $W_2 = \langle -0{,}53\ -0{,}23\ -0{,}45 \rangle$, $\theta_2 = 0$ e $\varepsilon_2 = 1$. O hiperplano ortogonal a esse vetor de pesos separa as instâncias E^2, E^3 e E^5 da Tabela 5.28. Essas instâncias são removidas do conjunto de treinamento, e o restante do conjunto é usado para treinar o terceiro neurônio.

O treinamento do terceiro neurônio u_3 usando o BCPMax determina o vetor de pesos $W_3 = \langle 0{,}74\ 0{,}58\ 0{,}67 \rangle$, $\theta_3 = 1{,}41$ e $\varepsilon_3 = 1$. Como nesse ponto a camada intermediária tem uma matriz de representações interna confiável (ver Tabela 5.29), o passo seguinte é definir as conexões entre a camada intermediária e o neurônio de saída e o *bias* do neurônio de saída. A rede Sequential com o BCPMax que resolve o problema Paridade-3 está mostrada na Figura 5.10.

Tabela 5.29 – Matriz de representação interna do exemplo de execução do Sequential com o BCPMax

Exemplos E^k (k = 1,...,8)	**O_{u1}**	**O_{u2}**	**O_{u3}**	**Classe C^k (k = 1,...,8)**
E^1	1	–1	1	1
E^2	–1	–1	1	–1
E^3	–1	–1	1	–1
E^4	–1	1	1	1
E^5	–1	–1	1	–1
E^6	–1	1	1	1
E^7	–1	1	1	1
E^8	–1	1	–1	–1

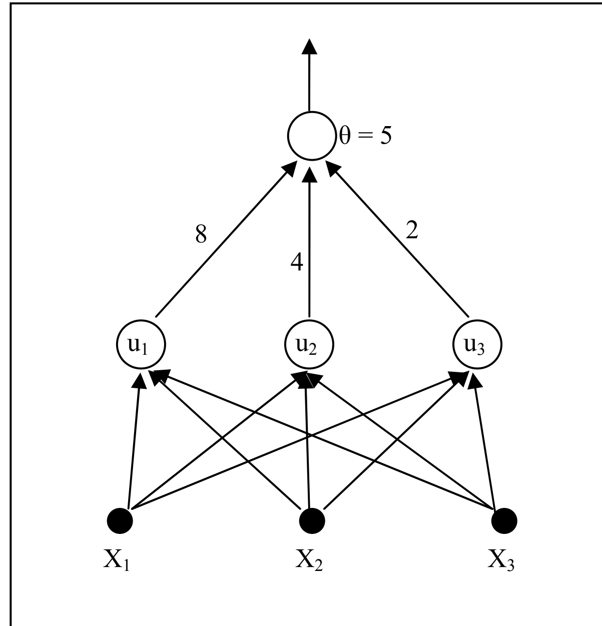


Figura 5.10 – Rede Sequential com o BCPMax representando o problema Paridade-3

5.6.2 O Algoritmo *Incremental Linear Programing* (IncLp)

Um outro algoritmo que pode ser usado na construção de uma rede Sequential é o algoritmo IncLp, proposto em [Marchand & Golea 1993]. Este algoritmo procura pelo maior conjunto de instâncias pertencentes a uma única classe que seja linearmente separável do resto das instâncias de treinamento. Para achar esse conjunto o IncLp divide o conjunto de treinamento inicial em dois: um deles contendo as instâncias de classe positiva e o outro as instâncias de classe negativa.

Em uma execução o algoritmo procura por um vetor de pesos, cujo hiperplano seu ortogonal, separe o maior conjunto de instâncias de classe positiva das instâncias de classe negativa, ou o maior conjunto de instâncias de classe negativa das instâncias de classe positiva. Para selecionar qual conjunto será obtido em uma determinada execução, o IncLp é controlado por um parâmetro (isPos, ver pseudocódigo em Algoritmo 5.7) que é passado pelo Sequential. O parâmetro isPos, quando verdadeiro, direciona o algoritmo a separar instâncias de classe positiva e, quando falso, direciona o algoritmo a separar instâncias de classe negativa.

Considere o caso em que o algoritmo deve procurar o maior conjunto de instâncias de classe positiva linearmente separável do resto do conjunto de treinamento. Para isso o algoritmo procura um vetor de pesos W (usando o Perceptron) utilizando um conjunto de treinamento composto por todos os exemplos de classe negativa e apenas um exemplo de

classe positiva. Se não existir W que separe o exemplo de classe positiva desse conjunto de instâncias com classes negativas, um outro exemplo de classe positiva é tomado e a busca começa novamente.

Quando um vetor de pesos W que satisfaz a condição anterior é achado, o algoritmo tentará classificar, usando este W, os exemplos restantes (todos os outros exemplos de classe positiva). Desses exemplos, todos os que forem classificados corretamente serão linearmente separáveis do resto do conjunto de treinamento. Esses exemplos são armazenados em um conjunto L e são retirados do conjunto de treinamento de instâncias com classe positiva.

```
Class IncLp
begin
{Entradas: E - conjunto de treinamento com n instâncias da forma:
            (‘x^k_0’, x^k_1, x^k_2,..., x^k_p, C^k), sendo que 1 ≤ k ≤ n
  isPos – parâmetro booleano passado como parâmetro}
{Saída: objeto incLp}

method IncLp (Matrix E, Bool isPos)
  begin
    if (isPos) then
      begin
        E1 ← separaInstanciasNegativas(E)
        E2 ← separaInstanciasPositivas(E)
      end
    else
      begin
        E1 ← separaInstanciasPositivas(E)
        E2 ← separaInstanciasNegativas(E)
      end

    while (|E2| > 0) do
      begin
        k ← seleciona (E2)
        if ( perceptron(L, k) ) then
          L ← E2^k

        for j ← 1 to |E2| do
          if ( j ≠ i and perceptron(L, j))
            L ← E2^j

        E2 ← retiraElementos(E2,L)

        for i ← 1 to |U| do
          if ( perceptron(L, i))
            L ← U^i

        if (|L| > |Lpoc|) then
          begin
            Lpoc ← L
            Wpoc ← W
          end

        U ← uniao(L, U)

      end {while}
  end {method}
end {Class}
```

Algoritmo 5.7 – Pseudocódigo do algoritmo IncLp

O algoritmo mantem um conjunto U que armazena todas as instâncias de treinamento excluídas em fases anteriores. Cada vez que um novo W é encontrado, o algoritmo tenta classificar as instâncias pertencentes a U na tentativa de incluí-las no conjunto L atual, aumentando assim as chances de encontrar um conjunto linearmente separável maior.

Depois de verificar se algum elemento de U pode também ser incluído no conjunto L atual, o algoritmo verifica se L possui mais elementos que Lpoc[22]. Se possuir, o conjunto L e o vetor de pesos W atuais são copiados para Lpoc e Wpoc respectivamente. O próximo passo é adicionar o conjunto L encontrado nessa iteração ao conjunto U.

O processo termina quando o conjunto de treinamento que continha as instâncias de classe positiva estiver vazio. Então o maior conjunto linearmente separável encontrado estará em Lpoc, e o vetor de pesos que o separou estará em Wpoc. O processo de procura de um hiperplano que separe o maior número de exemplos de classe negativa dos exemplos de classe positiva é análogo.

O método *perceptron*() apresentado no pseudocódigo é uma variação do Perceptron que recebe como parâmetros, uma instância e um vetor de instâncias de treinamento. Este método retorna verdadeiro se conseguir achar um vetor de pesos W que classifique corretamente as instâncias pertencentes ao conjunto E1 mais as instâncias passadas por parâmetro, i.e. uma instância de treinamento de E2 e o conjunto de instâncias L. Repare que nesta implementação o conjunto que será esvaziado sempre será E2.

5.6.3 O Algoritmo Sequential com o IncLP

O Algoritmo 5.8 descreve o pseudocódigo do Sequential, que faz uso do IncLp para encontrar um conjunto de instâncias de uma mesma classe que é linearmente separável do restante do conjunto de treinamento.

Com o objetivo de encontrar o maior conjunto linearmente separável a cada iteração, o Sequential cria dois objetos IncLp, um para encontrar o maior conjunto de instâncias positivas (exPos, ver pseudocódigo em Algoritmo 5.8) e o outro para encontra o maior conjunto de instâncias negativas (exNeg) linearmente separáveis do resto do conjunto de treinamento.

[22] Lpoc é uma referência a pocket.

```
Class Sequential
begin
Entradas: E - conjunto de treinamento com n instâncias da forma:
            ('xk0', xk1, xk2,..., xkp, Ck), sendo que 1 ≤ k ≤ n
  MAX - número máximo de neurônios intermediários, pré-determinado}
{Saída: objeto sequential que representa a rede neural criada}
CamadaMax ← MAX
Camada[CamadaMax]

method Sequential(Matrix E)
  begin
    while not oneClass(E1) do
      begin
        exPos ← new IncLp(E1, true)
        exNeg ← new IncLp(E1, false)
        cont ← 1
        if (exPos.getNroL() ≥ exNeg.getNroL()) then
            begin
              Camada[cont] ← exPos
              E1 ← exPos.getNewE()
              Camada[cont].setClassL(true)
            end
          else
            begin
              Camada[cont] ← exNeg
              E1 ← exNeg.getNewE()
              Camada[cont].setClassL(false)
            end
         cont ← cont + 1
       end

      indNeuronio ← 0
      nroNeuronios ← cont
      while nroNeuronios > 0  do
        begin
          pesos[indNeuronio] ← 2^nroNeuronios
          indNeuronio ← indNeuronio + 1
          nroNeuronios ← nroNeuronios - 1
        end

      for i ← 1 to cont do
        begin
          bias ← bias + aux * pesos[i]
          aux  ← aux * –1
        end
      bias ← bias + aux
  end  {method}
end  {Class}
```

Algoritmo 5.8 – Pseudocódigo do Sequential usando o IncLp

O Sequential adicionará na camada intermediária apenas o neurônio que mais excluiu exemplos de treinamento (exPos ou exNeg). Após adicionado, os exemplos de treinamento que este excluiu são retirados do conjunto original de treinamento (*getNewE*()) e o neurônio é marcado como positivo ou negativo, de acordo com os exemplos que este excluiu.

Esta marcação é feita, pois os neurônios treinados com o IncLp podem produzir algumas saídas iguais a 0, fato que pode tornar a matriz de representação interna não confiável. Esse problema, todavia, pode ser resolvido da seguinte forma: Se o neurônio que

gerou a saída 0 foi marcado como positivo, i.e. exclui exemplos de classe positiva, substituí-se o 0 por 1. Se, no entanto, o neurônio foi marcado como negativo, i.e. foi alocado para excluir instâncias negativas, substitui-se 0 por –1.

O processo chega ao fim quando houver somente exemplos de treinamento pertencentes a uma única classe no conjunto de treinamento original, quando *oneClass*(), no pseudocódigo, retornar verdadeiro.

5.6.3.1 Exemplo de Execução do Sequential em um Conjunto Não Linearmente Separável usando o IncLp

Para exemplificar o funcionamento do algoritmo Sequential com o IncLp, como descrito no pseudocódigo do Algoritmo 5.8, será usado o conjunto de treinamento Paridade-3, representado na Tabela 5.30.

Tabela 5.30 – Conjunto de exemplos que representa o problema Paridade-3 com a adição do termo *bias*

Exemplos E^k (k = 1,..., 8)	**X_0 (*bias*)**	**X_1**	**X_2**	**X_3**	**Classe C^k (k = 1,..., 8)**
E^1	1	1	1	1	1
E^2	1	1	1	–1	–1
E^3	1	1	–1	1	–1
E^4	1	1	–1	–1	1
E^5	1	–1	1	1	–1
E^6	1	–1	1	–1	1
E^7	1	–1	–1	1	1
E^8	1	–1	–1	–1	–1

O primeiro passo do algoritmo é criar dois objetos IncLp: um para separar exemplos positivos e o outro para separar exemplos negativos. Após criado ambos, o que separou mais instâncias é escolhido; no caso de separarem o mesmo número de instâncias o que separa exemplos positivos é escolhido.

Como no conjunto apresentado na Tabela 5.30 o maior conjunto inicial que pode ser separado possui apenas um elemento (positivo ou negativo), o neurônio que exclui exemplos positivos u_1 é adicionado na camada intermediária. Suponha que esse neurônio obtenha o vetor de pesos $W_1 = \langle -3\ -3\ 1\ -1 \rangle$ através do IncLp. Esse vetor de pesos define um hiperplano que separa a instância de classe positiva E^6, que é retirada do conjunto de treinamento.

Como não restam apenas instâncias de uma mesma classe no conjunto de treinamento, o algoritmo segue adicionando neurônios na camada intermediária. Suponha que o segundo neurônio u_2 adicionado, quando treinado defina o vetor de pesos $W_2 = \langle -1\ 3\ -3\ 3 \rangle$. Esse vetor

define um hiperplano que separa as instâncias negativas E^2, E^5 e E^8 que são então retiradas do conjunto de treinamento.

O conjunto restante formado pelas instâncias E^1, E^3, E^4 e E^7 ainda não possui apenas uma classe e desse modo, um terceiro neurônio u_3 deve ser adicionado. Suponha que quando treinado, o vetor de pesos associado a u_3 seja $W_3 = \langle 4\ -2\ 2\ -2 \rangle$, que define um hiperplano que separa as instâncias E^1, E^4 e E^7. Após a retirada dessas instâncias do conjunto de treinamento, sobra apenas a instância E^3 e o critério de parada é satisfeito.

A Tabela 5.31 representa a matriz de representações internas da rede Sequential. Como dito anteriormente, o Sequential com o IncLp pode gerar alguns valores iguais a 0 nesta matriz, o que pode torná-la não confiável. Neste exemplo somente o neurônio u_1 gerou 0, e como este neurônio foi marcado como positivo troca-se os zeros por –1 como mostra a Tabela 5.32.

Tabela 5.31 – Matriz de representação interna do exemplo de execução do Sequential com o IncLp

Exemplos E^k (k = 1,..., 8)	**O_{u1}**	**O_{u2}**	**O_{u3}**	**Classe C^k (k = 1,..., 8)**
E^1	–1	1	1	1
E^2	–1	–1	1	–1
E^3	–1	1	–1	–1
E^4	–1	1	1	1
E^5	0	–1	1	–1
E^6	1	–1	1	1
E^7	–1	1	1	1
E^8	0	–1	1	–1

Tabela 5.32 – Matriz de representação interna do exemplo de execução do Sequential com o IncLp corrigida

Exemplos E^k (k = 1,...,8)	**O_{u1}**	**O_{u2}**	**O_{u3}**	**Classe C^k (k = 1,..., 8)**
E^1	–1	1	1	1
E^2	–1	–1	1	–1
E^3	–1	1	–1	–1
E^4	–1	1	1	1
E^5	–1	–1	1	–1
E^6	1	–1	1	1
E^7	–1	1	1	1
E^8	–1	–1	1	–1

A Figura 5.11 ilustra a arquitetura da rede Sequential com o IncLp criada para esse exemplo. Note que a rede possui quatro neurônios na camada de entrada, uma vez que esta variação do Sequential faz uso do termo *bias* adicionado ao conjunto de treinamento.

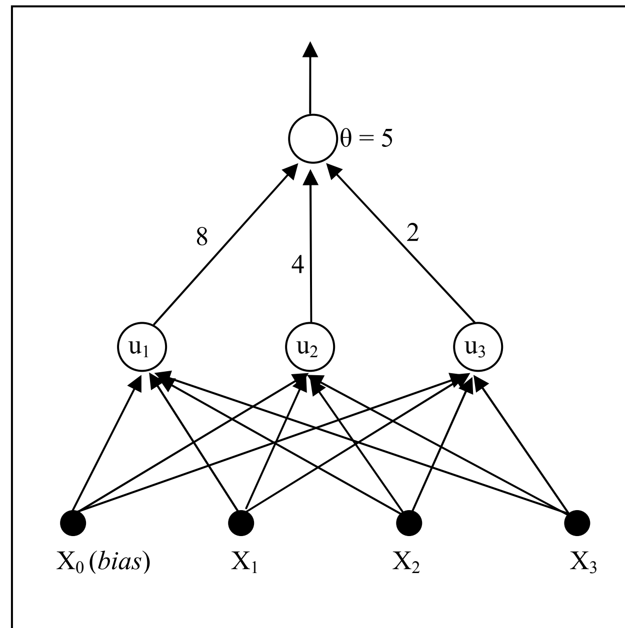


Figura 5.11 – Rede Sequential com o IncLp representando o conceito Paridade-3

6 capítulo

Aprendizado em Domínios Multiclasses – Versões Estendidas dos Algoritmos PMR e BCP

6.1 Introdução

Muitos problemas relacionados à classificação de padrões envolvem classificação em mais do que duas classes. Como visto anteriormente (Capítulo 2), uma única TLU pode ser treinada para discriminar instâncias que pertencem a uma de duas classes apenas. O aprendizado da aproximação de uma função que discrimina instâncias pertencentes a m classes distintas (com m > 2)[23] envolve o treinamento de m TLUs [Nilsson 1965].

Cada uma das m TLUs é associada a uma classe. Durante a fase de classificação, a classe de uma nova instância é dada pela classe associada à TLU que a instância ativar. Para que uma dentre as m TLUs seja ativada, é preciso que o seu valor de ativação seja maior do que o valor de ativação de todas as outras. Assim, a TLU com o maior valor de ativação terá saída 1 e as demais terão saídas –1.

Durante a fase de treinamento, uma instância é classificada incorretamente quando ativar uma TLU associada a uma classe diferente da sua. Um outro problema ocorre quando não existe um único valor de ativação maior que os demais. Desse modo duas ou mais TLUs possuirão o mesmo valor de ativação para uma determinada instância; nesse caso todas as m TLUs tornam-se inativas (saída –1).

O treinamento de TLUs para conjuntos multiclasses pode ser feito de duas maneiras: individualmente ou utilizando a abordagem *winner-takes-all* (WTA) [Gallant 1994]. No treinamento individual cada um dos m neurônios é treinado independentemente dos demais; para essa abordagem, é necessário que o conjunto utilizado para o treinamento de cada TLU possua duas classes.

A abordagem individual não leva em conta o relacionamento entre as classes. Este método de treinamento também não garante que a pertinência de uma instância a uma determinada classe inviabilize sua pertinência a qualquer outra classe. Já na abordagem WTA

[23] Para o restante do texto m será considerado maior que 2.

os m neurônios são treinados em conjunto garantindo a interação entre classes e explorando o fato de que uma instância pertence, idealmente, a uma única classe.

A abordagem de treinamento WTA oferece uma vantagem significativa sobre o treinamento individual para problemas multiclasses. Instâncias de classes que são separáveis duas a duas podem ser corretamente classificadas usando a abordagem WTA, enquanto a abordagem individual somente separa instâncias da classe que é separável de todas as outras. Na Seção 6.2 são apresentadas as abordagens individuais dos algoritmos PMR e BCP, com o pseudocódigo e exemplos de execução. Em seguida, na Seção 6.3 e Seção 6.4 serão apresentadas as versões WTA dos algoritmos PMR e BCP, respectivamente, também mostrando o pseudocódigo e exemplo de execução para cada algoritmo.

6.2 As Versões Multiclasses dos Algoritmos PMR e BCP – Abordagem Individual

Para viabilizar o treinamento individual de cada uma das TLUs, é preciso que o conjunto de treinamento original, constituído de n instâncias de treinamento representando m classes, seja reescrito como m conjuntos de treinamento com duas classes. Cada um desses m conjuntos vai ser usado no treinamento de uma das m TLUs, como descrito em Algoritmo 6.1. Ao fim do processo descrito cada TLU foi treinada para distinguir uma determinada classe de todas as outras.

No pseudocódigo do Algoritmo 6.1, o método *treinaTLU*() representa qualquer algoritmo de treinamento de TLU para duas classes, no caso o PMR ou o BCPMin. Este método retorna um objeto, obtido como resultado do treinamento realizado com o conjunto de treinamento que lhe foi passado como parâmetro. Se o algoritmo usado for o PMR o objeto conterá somente o vetor de pesos W. Se, no entanto, o algoritmo for o BCPMin, o objeto deve conter além do vetor de pesos, o termo *bias* calculado para o respectivo conjunto de treinamento.

O método *criaTreinamento*() recebe como parâmetros o valor de uma determinada classe e o conjunto de treinamento original, e retorna um conjunto de treinamento que possui duas classes (1 e –1). Esse conjunto é formado pelas instâncias de treinamento originais com suas classes trocadas da seguinte forma: troca-se por 1 as classes das instâncias que tem classe igual à que foi passada como parâmetro e todas as outras são trocadas por –1.

```
Class TreinaMTLUs
begin
{Entrada: E - conjunto de treinamento com n instâncias, distribuídas em m classes,
da forma: (‘xk0’, xk1, xk2,..., xkp, Ck), sendo que 1 ≤ k ≤ n}
{Saída: objeto referente ao PMR/Individual ou ao BCP/Individual}
W {Vetor de com m vetores de pesos, indexados como Wi, onde 1 ≤ i ≤ m}
classes ← encontraClasses() {vetor classes armazena as classes encontradas
no conjunto de treinamento}

method TreinaMTLUs(Matrix E)
  begin
    for i ← 1 to m do
      begin
        Ei ← criaTreinamento(classes[i], E)
        Wi ← treinaTLU(Ei)
      end
  end

method Matrix criaTreinamento(Integer k, Matrix E)
  begin
    for i ← 0 to n do
      begin
        Ai ← Ei
        if Ci = k then
          CkA ← 1
        else
          CkA ← –1
      end
    return A
  end
end  {Class}
```

Algoritmo 6.1 – Pseudocódigo para o treinamento individual de m TLUs

6.2.1 Exemplo do PMR Multiclasse – Abordagem Individual

Considere o conjunto de treinamento com três classes representado na Tabela 6.1. Como o algoritmo de treinamento de TLU usado é o PMR, considere que este conjunto tenha sofrido a inclusão do termo *bias* (não representado na Tabela 6.1). A saída da rede será um vetor com tantas posições quantas forem as classes e, portanto, as classes ‘+’, ‘–’ e ‘*’; estão associadas aos vetores: $\langle 1\ -1\ -1 \rangle$, $\langle -1\ 1\ -1 \rangle$ e $\langle -1\ -1\ 1 \rangle$ respectivamente. Esta associação ocorre de acordo com a ordem em que as classes são encontradas no conjunto de treinamento. De acordo com a associação das classes com os respectivos vetores de saída tem-se que o vetor de pesos W_1 deverá classificar a classe ‘+’; W_2 classifica a classe ‘–’; e por fim W_3 classifica a classe representada por ‘*’.

Tabela 6.1 – Conjunto de treinamento com três classes e vinte e quatro instâncias

Exemplos E^k (k = 1,..., 12)	**X**	**Y**	**Classe C^k (k = 1,..., 12)**	**Exemplos E^k (k = 13,..., 24)**	**X**	**Y**	**Classe C^k (k = 13,..., 24)**
E^1	1	5	+	E^{13}	12	2	–
E^2	1	2	+	E^{14}	12	5	–
E^3	2	1	+	E^{15}	13	4	–
E^4	2	3	+	E^{16}	14	3	–
E^5	3	2	+	E^{17}	5	6	*
E^6	3	4	+	E^{18}	6	5	*
E^7	4	1	+	E^{19}	6	6	*
E^8	5	3	+	E^{20}	6	7	*
E^9	10	1	–	E^{21}	7	5	*
E^{10}	10	3	–	E^{22}	7	7	*
E^{11}	11	2	–	E^{23}	8	4	*
E^{12}	11	3	–	E^{24}	8	6	*

O PMR, versão duas classes, foi executado 24000 vezes[24] e os vetores de pesos obtidos foram: $W_1 = \langle 42\ –6\ –3 \rangle$, $W_2 = \langle –41\ 20\ 39 \rangle$ e $W_3 = \langle –65\ –2\ 17 \rangle$. Esses três vetores classificam corretamente todas as instâncias de treinamento. Os hiperplanos gerados por esses vetores de pesos: H_1: $–6X – 3Y + 42 = 0$, H_2: $20X + 39Y – 41 = 0$ e H_3: $–2X + 17Y – 65 = 0$, bem como o conjunto de treinamento da Tabela 6.1 estão mostrados na Figura 6.1.

Note que a classe representada por '*' não é linearmente separável do resto do conjunto de treinamento, porém, este conjunto de hiperplanos separa corretamente todos os exemplos de treinamento. A instância (8,4), apesar de ter sido classificada incorretamente pelo hiperplano H_3, é classificada no contexto multiclasse, pois a instância induz o maior valor de ativação na TLU associada a W_3.

[24] Número correspondente a 1000 vezes o número de instâncias de treinamento.

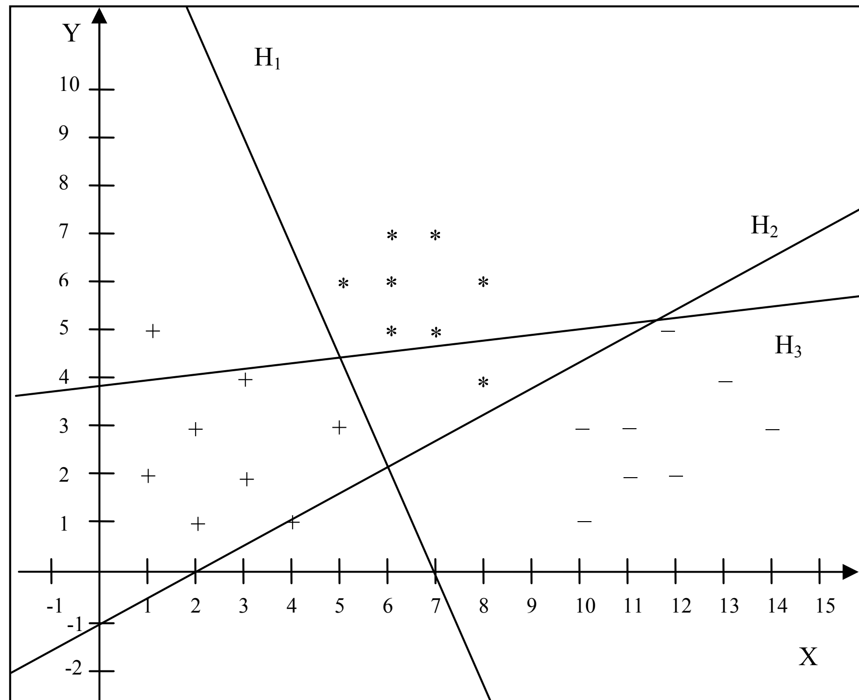


Figura 6.1 – Classificação multiclasse utilizando o PMR em abordagem individual

6.2.2 Exemplo do BCP Multiclasse – Abordagem Individual

Considere o mesmo conjunto de treinamento dado na Tabela 6.1, dessa vez o conjunto não sofre adição do termo *bias*, pois o BCP não necessita de um termo *bias* no conjunto de treinamento. Considere também que as classes foram associadas aos mesmos vetores de saída, uma vez que as suas posições no conjunto de treinamento não mudaram. A execução do algoritmo BCPMin com um número de 10 iterações, produziu os seguintes vetores de pesos e respectivos *biases*: $W_1 = \langle -6{,}42 \;\; -1{,}6 \rangle$, $\theta_1 = 39{,}28$; $W_2 = \langle 7{,}07 \;\; -1{,}36 \rangle$, $\theta_2 = -58{,}9$ e $W_3 = \langle 0{,}9 \;\; 8{,}11 \rangle$ $\theta_3 = -45{,}08$. Essa solução classifica corretamente todas instâncias de treinamento e define os seguintes hiperplanos: H_1: $-6{,}42X - 1{,}6 + 39{,}28 = 0$, H_2: $7{,}07X - 1{,}36Y - 58{,}9 = 0$ e H_3: $0{,}9X + 8{,}11Y - 45{,}08 = 0$.

Os hiperplanos H_1, H_2 e H_3 representam as classes '+', '–' e '*' respectivamente. Note, na Figura 6.2, que a instância (8,4), não foi classificada corretamente pelo hiperplano H_3 (situação análoga aconteceu no exemplo da Subseção 6.2.1). A instância, no entanto é classificada corretamente quando considerado o caso multiclasse, também como no exemplo anterior.

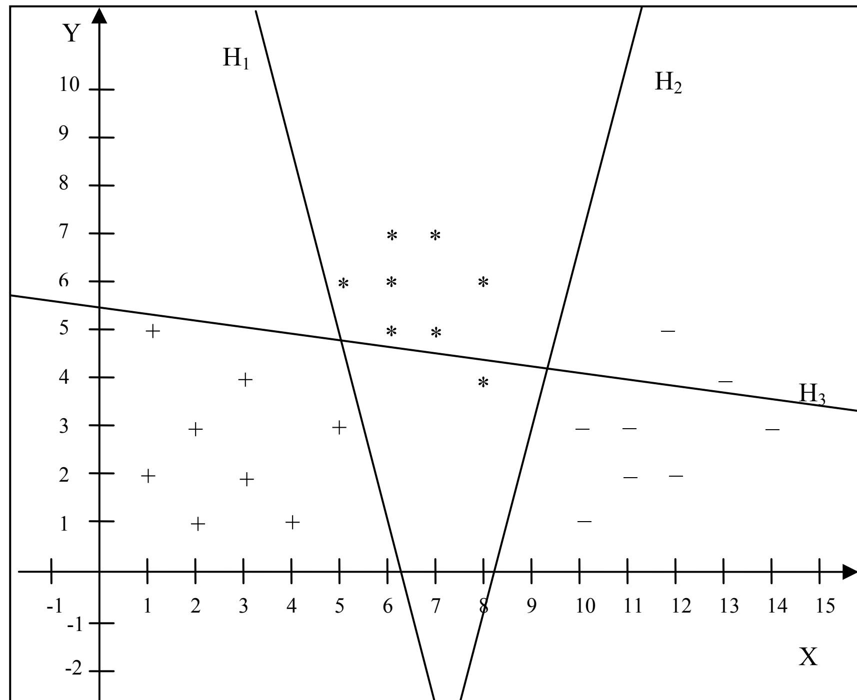


Figura 6.2 – Classificação multiclasse utilizando o BCP em abordagem individual

6.3 A Versão Multiclasse do Algoritmo PMR – Abordagem WTA

A versão multiclasse WTA do algoritmo PMR, como descrita em [Parekh *et al.* 2000] é uma extensão direta do algoritmo original para duas classes. A única alteração está no fato da versão treinar tantos neurônios quantos forem as classes ao invés de apenas um, como o PMR original.

A cada um dos m neurônios é associada uma determinada classe antes do início do treinamento; esta associação determina qual dos m neurônios deverá estar ativo para determinada classe. A associação é feita conforme a ordem em que as classes são encontradas no conjunto de treinamento, i.e. a enésima classe encontrada é associada ao enésimo neurônio.

Como visto anteriormente, o treinamento de uma TLU usando instâncias de treinamento descritas por p atributos e uma classe associada resulta na obtenção de um vetor de pesos W de dimensão p + 1 (considerando o *bias*). O treinamento de m TLUs resulta na obtenção de um vetor de vetores de pesos, W, no qual cada posição W_j ($1 \leq j \leq m$) representa o vetor de pesos obtido no treinamento da TLU identificada por j.

Como o algoritmo treina m neurônios em paralelo a saída será um vetor com m posições, no qual cada posição O_j^k representa a saída do neurônio j para a instancia E^k.

Seja $O^k = \langle O_1^k, O_2^k, ..., O_m^k \rangle$ o vetor obtido com as saídas dos m neurônios para a instância E^k. O vetor saída $O^k = \langle O_1^k, O_2^k, ..., O_m^k \rangle$ é calculado como segue:

(1) Se $\exists$ j $\in$ {1,...,m} tal que $W_jE^k > W_iE^k$ $\forall i \neq j$, i $\in$ {1,...,m} então $O_j^k = 1$ e $O_i^k = -1$ para todo $i \neq j$.

(2) Se, no entanto, $\exists$ $j_1, j_2, ..., j_r \in \{1,...,m\}$ tal que $W_{j_1}E^k = W_{j_2}E^k = ... = W_{j_r}E^k$ e $W_{j_1}E^k > W_iE^k$ $\forall i \notin \{j_1, j_2,...,j_r\}$ então $O_j^k = -1$ $\forall j \in \{1,...,m\}$.

Durante o treinamento, considere que $D^k = \langle D_1^k, D_2^k, ..., D_m^k \rangle$ seja o vetor que representa a saída esperada dos m neurônios para a instância E^k. Se ao final de uma iteração o vetor O^k for igual ao vetor D^k então o algoritmo classifica a instância E^k corretamente. Se, no entanto, os vetores O^k e D^k forem diferentes, os vetores de pesos que geraram saída diferente são atualizados de acordo com a regra do Perceptron, dada pela equação (6.1).

$$W_j \leftarrow W_j + \eta\left(D_j^k - O_j^k\right)E^k \quad \forall j \in \{1,...,m\} \tag{6.1}$$

na qual η é a taxa de aprendizado. Não existe uma regra geral para determinar um valor para a taxa de aprendizado. Na verdade esse valor depende do conjunto de treinamento sendo tratado. Uma abordagem satisfatória, no entanto, é inicializar η com o valor 1 e cada vez que o número de iterações do algoritmo for divisível pelo número de instâncias de treinamento, η tem seu valor multiplicado por 0,95. Isso faz com que a variação de W diminua de intensidade à medida que o treinamento progride.

No pseudocódigo apresentado em Algoritmo 6.2, o algoritmo armazena o conjunto de vetores de pesos W que classifica mais instâncias de treinamento corretamente em Wpoc. Assim como na versão original, para que Wpoc seja atualizado com os pesos de W é necessário que, além de classificar corretamente mais instâncias de treinamento consecutivas que Wpoc, W também classifique maior número de instâncias de treinamento distintas que Wpoc. O número de instâncias consecutivas classificadas corretamente por W e por Wpoc são armazenados, no pseudocódigo, em consecW e consecWpoc respectivamente. Os totais de instâncias classificadas corretamente por W e Wpoc (calculado pelo método *nroCorretos*()) são armazenados em corretosW e corretosWpoc respectivamente.

```
Class PMRWTA
begin
{Entradas: E - conjunto de treinamento com n instâncias, distribuídas em m classes,
            da forma: (x^k_0, x^k_1, x^k_2,..., x^k_p, C^k), sendo que 1 ≤ k ≤ n
  MAX - número de iterações, pré-determinado}
{Saída: objeto pmrwta que contém os m vetores de pesos}
method PMRWTA(Matrix E)
  begin
    for i ← 1 to m do
      W_i ← ⟨0, 0, 0,..., 0⟩  {Wi é iniciado como um vetor nulo com p+1 posições }
    Wpoc ← W;   consecW ← 0;   consecWpoc ← 0;  corretosW ← 0;  corretosWpoc  ← 0
    it ← 0;   itMax ← MAX {número máximo de iterações, pré-determinado}
    η ← 1 {taxa de aprendizado}
    D ← criaSaidaEsperada()

    while it < itMax do
      begin
        k ← seleciona(1,n)
        O^k  ← saidaM(k)
        if ( O^k = D^k ) then
          begin
            consecW ← consecW + 1
            if consecW > consecWpoc then
              begin
                corretosW ← nroCorretos()
                if  corretosW > corretosWpoc then
                  begin
                    consecWpoc ← consecW
                    corretosWpoc ← corretosW
                    for i ← 1 to m do
                      Wpoc_i ← W_i
                  end
              end
          end
        else
          begin
            consecW ← 0
            for i ← 1 to m do
              W_i ← W_i + η(D^k_i − O^k_i)E^k
          end
        if (it mod n) = 0 then
          η ← 0,95η
        it ← it + 1
      end
  end   {method}

method Vector saidaM(Integer k)
  begin
    O ← ⟨0, 0, 0,..., 0⟩ ; P ← ⟨0, 0, 0,..., 0⟩
    for i ← 1 to m do
      P_i ← W_i E^k            {vetor P armazena os m potenciais de ativação para a instância k}
    maior ← P_1;   ind ← 1
    for j ← 2 to m do
      if maior < P_j then
        begin
          O_ind  ← -1;  maior ← P_j;   O_j ← 1;  ind ← j
        end
      else if  maior = P_j then
          for i ← 1 to m do
            O_i ←  −1
    return O
  end   {method}
end  {Class}
```

Algoritmo 6.2 – Pseudocódigo do algoritmo PMRWTA

Assim como no PMR original, na sua versão multiclasse o conjunto de treinamento é acessado de maneira aleatória. No pseudocódigo, o método *seleciona*() gera um número inteiro e aleatório k dentro do intervalo passado como parâmetro (no caso $k \in \{1,..., n\}$). Este número é usado como índice e é passado para o método *saidaM*(), que gera o vetor saída, com base no W atual, para a instância k em questão.

O vetor saída O^k gerado por *saidaM*() é então comparado com a linha k da matriz D, que é a matriz com as saídas esperadas para todas as instância de treinamento. A saída esperada D é gerada antes do inicio do treinamento pelo método *criaSaidaEsperada*(). Se todas as posições dos vetores forem iguais, a instância é classificada corretamente, caso contrário os vetores de pesos que apresentam saída errada são atualizados de acordo com a regra apresentada na equação (6.1). Note que dentre os m vetores de pesos, só serão atualizados os que geraram saída diferente da saída esperada.

6.3.1 Exemplo do PMR Multiclasse – Abordagem WTA

Para este exemplo de execução, considere o conjunto de treinamento com três classes representado na Tabela 6.1 com a inclusão do termo *bias* (não representado na Tabela 6.1). As classes '+', '–' e '*' apresentadas no conjunto de treinamento continuam sendo associadas aos vetores de saída ⟨1 –1 –1⟩, ⟨–1 1 –1⟩ e ⟨–1 –1 1⟩ respectivamente.

A Tabela 6.2 mostra os vetores de pesos W_1, W_2 e W_3 e o número de instâncias corretamente classificadas a cada 100 iterações do algoritmo, obtidas da execução do algoritmo PMRWTA, tendo o conjunto da Tabela 6.1 como entrada. Note que apesar do valor dos exemplos classificados corretamente variar bastante, o conjunto de vetores de pesos ótimo é obtido ao final da execução.

Tabela 6.2 – Vetores de pesos obtidos da execução do algoritmo PMRWTA para o conjunto de treinamento da Tabela 6.1

Iteração	W_1	W_2	W_3	Número de acertos
0	⟨0 0 0⟩	⟨0 0 0,	⟨0 0 0,	8
100	⟨2 6 8⟩	⟨0 0 0⟩	⟨0 0 0⟩	8
200	⟨22 –2 20⟩	⟨–8 28 –38⟩	⟨–10 –14 30⟩	14
300	⟨40 –20 20⟩	⟨–14 34 –44⟩	⟨–22 –2 36⟩	18
400	⟨60 –16 24⟩	⟨–26 38 –58⟩	⟨–30 –10 46⟩	20
500	⟨74 –24 36⟩	⟨–32 36 –60⟩	⟨–38 0 36⟩	21
600	⟨80 –26 6⟩	⟨–36 50 –54⟩	⟨–40 –12 60⟩	16
700	⟨92 –44 34⟩	⟨–42 56 –62⟩	⟨–46 0 40⟩	19
800	⟨106 –28 42⟩	⟨–50 54 –74⟩	⟨–50 –6 46⟩	19
900	⟨122 –12 38⟩	⟨–64 42 –80⟩	⟨–52 –10 56⟩	14
1000	⟨126 –24 32⟩	⟨–66 50 –74⟩	⟨–54 –6 52⟩	24

Os vetores de pesos obtidos W_1, W_2 e W_3 geram os hiperplanos H_1, H_2 e H_3, graficamente mostrados na Figura 6.3, cujas equações são: H_1: $-24X + 32Y + 126 = 0$; H_2: $50X - 74Y - 66 = 0$ e H_3: $-6X + 52Y - 54 = 0$ respectivamente. Neste exemplo de execução o hiperplano H_1 está associado às instâncias de treinamento representadas por '+'; o hiperplano H_2, por sua vez, está associado às instâncias representadas por '-' e o hiperplano H_3 às instâncias representadas por '*'. Note a disposição dos hiperplanos obtidos na Figura 6.3, que um hiperplano não separa a classe que representa, das outras duas classes, como acontece com a abordagem individual.

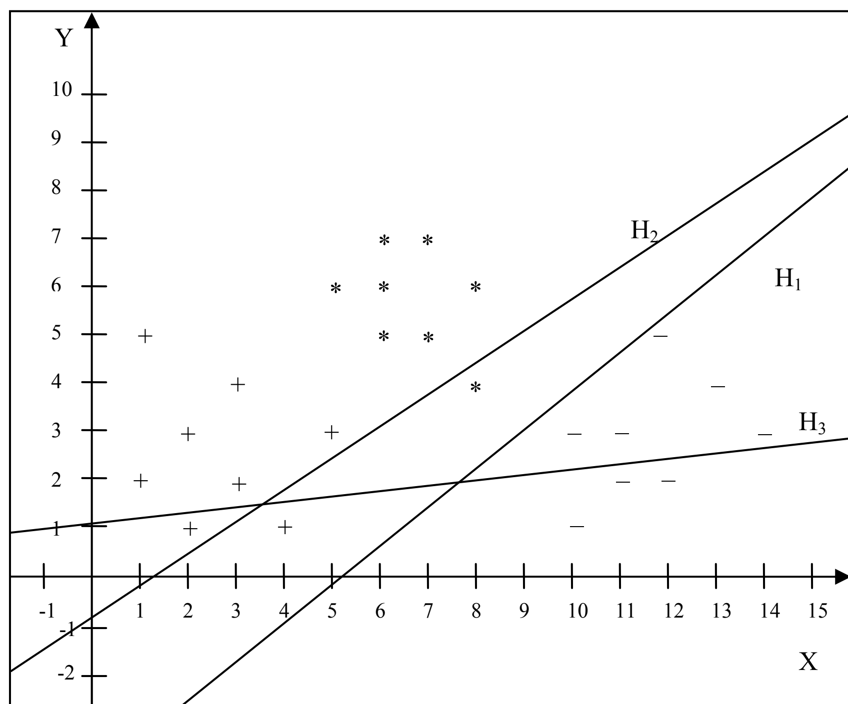


Figura 6.3 – Classificação multiclasse feita pelo PMRWTA

6.4 A Versão Multiclasse do Algoritmo BCP – Abordagem WTA

A extensão do algoritmo BCP [Poulard 1995] para a classificação de instâncias em domínios multiclasses usando a abordagem WTA não é natural como no caso do algoritmo PMR discutida na Seção 6.3. A versão multiclasse WTA do BCP [Parekh *et al.* 2000] é apenas uma versão aproximada da versão duas classes do algoritmo original.

Na versão BCP usando a abordagem WTA o vetor de pesos, W, de um determinado neurônio é o baricentro da classe representada por este neurônio. Os valores do *bias* dos neurônios são calculados por meio da minimização dos erros acumulados, usando uma adaptação do algoritmo *Loss Minimization* [Hrycej 1992].

O baricentro é determinado com a ajuda de um vetor de coeficientes de pesos α. Assim como na versão original, este vetor é inicializado com valores aleatórios pertencentes ao intervalo [1,*a*]. Um bom valor para *a*, para o caso de problemas multiclasses, é *a* = 4, como sugerido em [Parekh *et al.* 2000].

O vetor α de coeficientes de pesos é atualizado ao fim de cada época; na atualização são adicionados valores aleatórios do intervalo [0,1] aos pesos cuja respectiva instância foi classificada incorretamente. Isto faz com que o baricentro de uma determinada classe aproxime-se das instâncias pertencentes a esta classe que foram classificadas incorretamente e melhore a classificação destas instâncias nas próximas iterações.

Determinados os vetores de pesos, o erro (*loss*) pode ser definido como a soma dos erros ao quadrado para todas as instâncias. Suponha que o i-ésimo neurônio obtenha o maior potencial de ativação para uma instância E^k, e que a saída esperada para esta instância fosse o j-ésimo neurônio. Desse modo o erro acumulado (*cumulative loss*) para as instâncias é dado pela equação (6.2).

$$Q = \sum_k \left(v_i^k - v_j^k\right)^2 \tag{6.2}$$

na qual $v_i^k = E^k.W_i$ é o potencial de ativação do neurônio i para a instância E^k. Pode ser demonstrado que a função do erro acumulado é convexa e diferenciável e, conseqüentemente, tem um mínimo único ([Shynk 1990] e [Hrycej 1992]). Os *biases* que correspondem ao menor valor (ponto de mínimo) da função Q são encontrados por meio do gradiente descendente:

$$\theta_j = \theta_j + \eta\left(v_i^k - v_j^k\right) \tag{6.3}$$

na qual θ_j é o *bias* do neurônio j e η é a taxa de aprendizado. A taxa de aprendizado pode ter um valor fixado no início do treinamento; esse valor, porém, muitas vezes não é o melhor valor para determinado conjunto de treinamento. Para evitar a escolha de um valor para cada conjunto de treinamento e para diminuir a oscilação do erro acumulado devido a altos valores de η, a taxa de aprendizado é dinamicamente atualizada para valores menores. Desse modo, a taxa de aprendizado é inicializada com 1 e ao fim de cada época o valor de η é multiplicado por 0,95 (como sugerido em [Parekh *et al.* 2000]).

O Algoritmo 6.3 descreve o pseudocódigo do algoritmo BCP com abordagem WTA. Como mencionado anteriormente os m vetores de pesos são obtidos do baricentro das m

classes. No pseudocódigo o método *baricentro*() retorna o vetor de pesos W da classe passada como parâmetro por meio do vetor classes[]. O vetor classes[] armazena as diferentes classes na ordem em que são encontradas no conjunto de treinamento; essas classes são recuperadas pelo método *encontraClasses*().

O vetor alfa[] é o vetor de coeficientes de pesos gerado pelo método *aleatorio*() com valores aleatórios no intervalo [1,4]. Ao final de uma época, os pesos cujas respectivas instâncias foram classificadas incorretamente, são acrescidos de um valor aleatório no intervalo [0,1]. Os índices das instâncias de treinamento que foram classificadas incorretamente durante uma época são armazenados no vetor erros[].

Os potenciais de ativação dos m neurônios calculados para toda instância em cada época são armazenados no vetor P[]. Para cada instância, se o índice do neurônio que classifica essas instâncias corresponder ao índice no vetor P[] cuja posição guarda o maior elemento, então a instância está corretamente classificada. Note que o *bias* só é atualizado quando a instância é classificada incorretamente.

O método *nroCorretos*() retorna o número de instâncias que foram classificadas corretamente; esse método é chamado ao final de cada época e então é verificado se a solução corrente é melhor do que a armazenada. Se for, os m vetores de pesos e os *biases* são armazenados em Wpoc e em θpoc, respectivamente.

```
Class BCPWTA
begin
{ Entradas: E - conjunto de treinamento com n instâncias, distribuídas em m classes,
            da forma: ( x^k_1, x^k_2,..., x^k_p, C^k), sendo que 1 ≤ k ≤ n
  MAX - número de iterações, pré-determinado}
{Saída: objeto bcpwta que contém os m vetores pesos juntamente com os m biases}
 classe ← encontraClasses() {vetor classes armazena as classes encontradas no conjunto
de treinamento}

method BPCWTA(Matrix E)
  begin
    exCorretos ← 0; η ← 1; it ← 0; itMax ← MAX; erro ← ⟨0,...,0⟩
    for i ← 1 to n do
      alfa[i] ← aleatorio(1,4)
    while it < itMax do
      begin
        for j ← 1 to m do
          W_j ← baricentro(E,classe[j])

        for i ← 1 to n do
          begin
            for j ← 1 to m do
              begin
                P[j] ← W_j E^i + θ_j
                if classes[j] = C^i then
                  indCorreto ← j
              end
            θ_indCorreto ← θ_indCorreto + η(maior(P) – P[indCorreto])
            if(maior(P) – P[indCorreto] ≠ 0) then {erra}
              erros[i] ← i
          end

        nroCorretos ← nroCorretos()
        if nroCorretos  > exCorretos  then
          begin
            exCorretos  ←  nroCorretos
            for i ← 1 to m do
              begin
                Wpoc_i ← W_i
                θpoc_i ← θ_i
              end
          end
        for k ←1 to n do        {atualização do vetor alfa}
          if erros[k] ≠ 0 then
            begin
              alfa[k] ← alfa[k] + aleatorio(1,2)
              erro[k] ← 0
            end
        η ←  0,95η
        it ← it + 1
      end
  end  {method}
end {Class}
```

Algoritmo 6.3 – Pseudocódigo do algoritmo BCPWTA

6.4.1 Exemplo do BCP Multiclasse – Abordagem WTA

Para este exemplo de execução, considere novamente o conjunto de treinamento da Tabela 6.1, sem a adição do termo *bias*. A Tabela 6.3 apresenta os vetores de pesos e os

termos *biases* encontrados durante a execução do algoritmo BCPWTA; note que o algoritmo converge na segunda época.

Tabela 6.3 – Conjunto de vetores de pesos e *bias* obtidos na execução do algoritmo BCPWTA

Época	W_1	θ_1	W_2	θ_2	W_3	θ_3	Número de Acertos
1	⟨2,65 2,68⟩	45,60	+11,67 2,68⟩	0,0	+6,67 5,73⟩	28,79	17
2	⟨1,95 2,32⟩	64,40	⟨11,67 2,86⟩	0,0	⟨6,74 6,30⟩	28,49	24

A Figura 6.4 mostra a representação geométrica dos hiperplanos H_1: 1,95X + 2,32Y + 64,4 = 0, H_2: 11,67X + 2,86Y = 0 e H_3: 6.74X + 6,3Y + 28,49 = 0 obtidos pelo algoritmo BCPWTA.

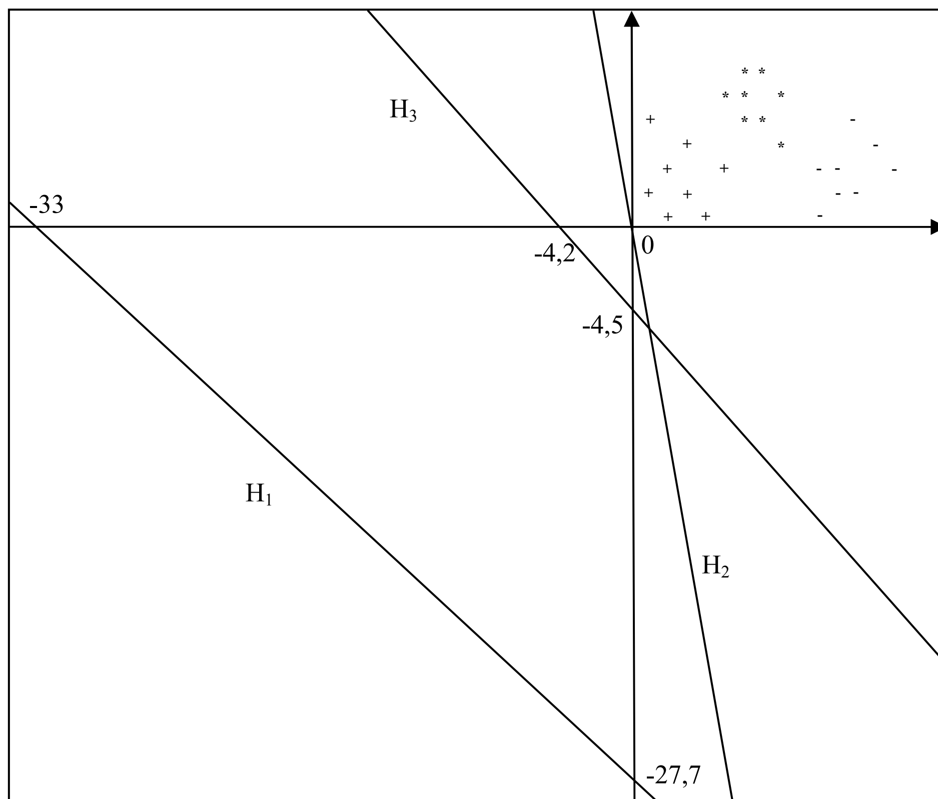


Figura 6.4 – Classificação multiclasse feita pelo algoritmo BCPWTA

7 capítulo

Aprendizado Neural Construtivo Multiclasse – Versões Estendidas dos Algoritmos Tower, Pyramid, Tiling, Upstart e Perceptron Cascade

7.1 Introdução

No Capítulo 4 foi feita uma revisão dos algoritmos Tower, Pyramid, Tiling e Upstart como propostos originalmente, ou seja, para problemas envolvendo instâncias de duas classes apenas. Este capítulo discute as versões multiclasses desses algoritmos e apresenta o pseudocódigo orientado a objeto de cada um deles. A extensão para problemas multiclasses do Perceptron Cascade (ver Seção 5.5) é também discutida.

Para a melhor visualização e compreensão das figuras que são apresentadas neste capítulo, considere a Figura 7.1. Para problemas com m classes (com m > 2) são treinadas m TLUs e essas TLUs são adicionadas de uma só vez em um determinado passo do algoritmo; cada grupo adicionado em um passo será denotado como mostra a Figura 7.1 (a) (para m = 3). A situação com duas camadas de neurônios, totalmente conectados, como mostrada na Figura 7.1 (b) será representada como na Figura 7.1 (c).

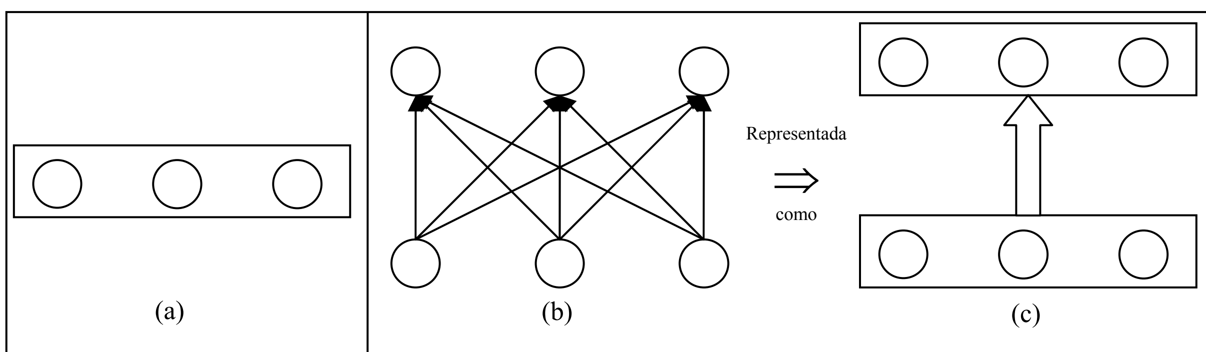


Figura 7.1 – Representações gráficas para algoritmos multiclasses. (a) inclusão de um grupo de neurônios; (b) duas camadas com neurônios totalmente conectados; (c) nova representação da situação mostrada em (b)

7.2 O Algoritmo Multiclasse MTower

Proposto em [Parekh *et al.* 1995] o algoritmo MTower é uma extensão direta do algoritmo Tower [Gallant 1986], para tratar problemas de aprendizado multiclasses. Assim como na versão original do Tower, o algoritmo MTower adiciona neurônios à rede de maneira a criar uma arquitetura semelhante a uma torre.

No caso multiclasses, entretanto, ao invés da adição de um neurônio por camada a cada iteração do algoritmo, adicionam-se tantos neurônios quantas forem as classes existentes no problema em questão. Em um problema com m classes, portando, cada camada será composta por m neurônios. Cada um dos m neurônios de uma determinada camada recebe conexões de todos os neurônios da camada de entrada mais as conexões de todos os m neurônios da camada imediatamente anterior como mostra o diagrama da Figura 7.2.

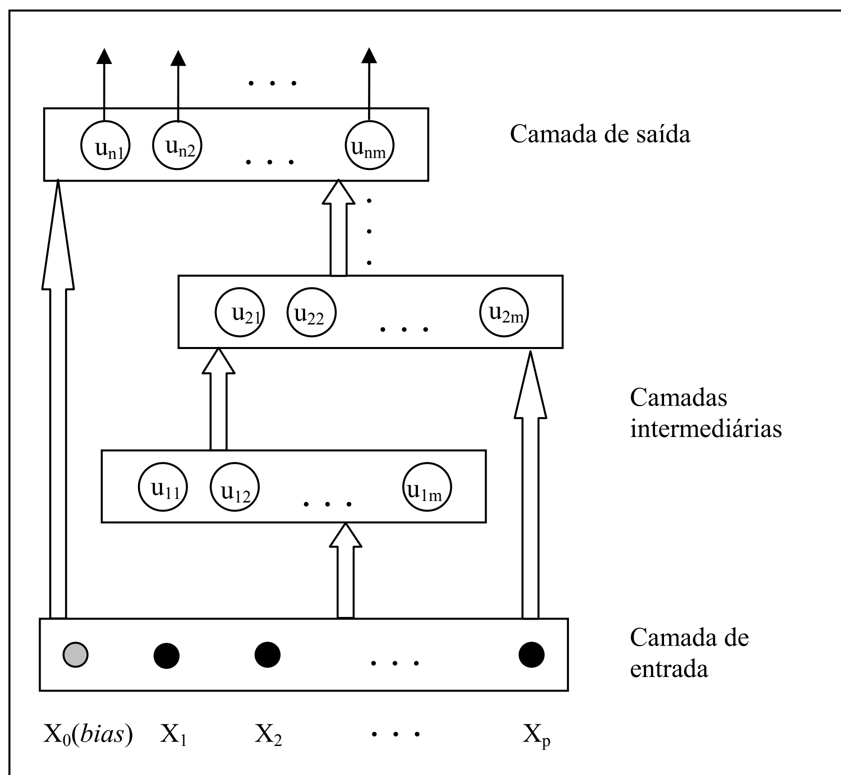


Figura 7.2 – Exemplo de arquitetura de uma rede MTower. O neurônio notado por u_{ij} é o j-ésimo neurônio da i-ésima camada intermediária

Note na Figura 7.2 que o neurônio referente ao termo *bias* é diferente dos demais, isso porque o conjunto de treinamento usado para treinar uma rede MTower possuirá o termo *bias* se o algoritmo usado para o treinamento das TLUs for algum derivado do Perceptron. Se, no

entanto, a rede usar o algoritmo BCP para o treinamento, o conjunto de treinamento não deverá sofrer a inclusão do termo *bias* (ver versões multiclasses do BCP no Capítulo 6).

Conside um conjunto de treinamento descrito por p atributos. Cada um dos m neurônios da primeira camada adicionada possuirá vetores de pesos com p ou p + 1 (referente à adição do *bias*) pesos. Já os conjuntos de treinamento para os neurônios pertencentes às camadas intermediárias seguintes sofrerão o acréscimo de m dimensões referentes às saídas da camada imediatamente anterior. Desse modo, os demais neurônios possuirão vetor de pesos com dimensão p + m (ou p + 1 + m, quando o *bias* for introduzido).

O processo de inserção de camadas termina quando algum dos critérios de parada for satisfeito. Os critérios de parada no MTower são os mesmo do Tower duas classes. O MTower termina quando uma das três situações ocorre: (1) A rede atual classifica todas as instâncias corretamente; (2) o número máximo de camadas atinge o limite previamente determinado e, o mais provável, (3) quando a precisão da rede corrente é pior do que a precisão da rede sem a última camada. Neste último caso, o algoritmo retira a camada que piorou a rede e finaliza.

O pseudocódigo do algoritmo MTower que está descrito em Algoritmo 7.1 é muito semelhante ao pseudocódigo do algoritmo Tower original. Nesta versão, como já mencionado, m neurônios são treinados a cada passo ao invés de apenas um. Para elucidar o fato de que o algoritmo é independente tanto da abordagem usada (individual ou WTA) quanto do algoritmo usado no treinamento de TLUs individuais, a classe de nome *NeuronioM* foi usada. Esta classe representa qualquer algoritmo de treinamento para m neurônios.

As camadas intermediárias da rede são armazenadas no vetor *Camadas*[]. Esse vetor armazena objetos do tipo *NeuronioM*. Se a construção da rede terminar porque a precisão da rede piorou com a adição de uma camada, esta camada é eliminada para que a rede volte a ter a precisão anterior. O método *apaga*() implementa a eliminação de uma camada da rede.

O método *criaTreinamento*() cria o conjunto de treinamento para ser usado no treinamento do próximo neurônio que será adicionado à rede. Este método cria um conjunto de treinamento juntando o conjunto de treinamento original com as m saídas do último neurônio intermediário adicionado à rede. O método *precisao*() retorna a precisão da rede corrente. O método *saidaM*() da classe *NeuronioM* (ver algoritmos de treinamento de m TLUs no Capítulo 6) cria a saída para a instância cujo índice foi passado como parâmetro.

```
Class MTower
begin
{Entradas: E - conjunto de treinamento com n instâncias, distribuídas em m (m > 2) classes,
           da forma: (‘xk0’, xk1, xk2,..., xkp, Ck), sendo que 1 ≤ k ≤ n
  MAX - número máximo de camadas, pre-determinado}
{Saída: objeto mTower que representa a rede neural criada}
camadaMax ← MAX
camada ← 0 {contador para as camadas da rede}
Camadas[camadaMax] { vetor de objetos do tipo NeuronioM que representa as camadas da
rede}

method MTower(Matrix E)
  begin
    E1 ← E
    precisao ← 0
    precisaoAnterior ← 0
    repeat
      begin
        camada ← camada + 1
        Camadas[camada] ← new NeuronioM(E1)
        precisaoAnterior ← precisao
        precisao ← precisao(E1)
        E1 ← criaTreinamento(E)
      end
    until (precisaoAnterior ≥ precisao or precisao = 1 or camada = camadaMax)

    if precisaoAnterior ≥ precisao then
      begin
        apaga (Camadas[camada])
        camada ← camada  – 1
      end
  end {method}

method Matrix criaTreinamento(Matrix E)
  begin
    A ← E
    S ← ⟨0,…,0⟩ {S é um vetor de m posições}
    for i ← 1 to n do
      begin
        S ← Camadas[camada].saidaM(i)
        for k ←  1 to m do
          A[i][p+k] ← S[k]
      end
    return A
  end
end {Class}
```

Algoritmo 7.1 – Pseudocódigo do algoritmo MTower

7.3 O Algoritmo Multiclasse MPyramid

O algoritmo MPyramid, também proposto em [Parekh *et al.* 1995], é uma extensão do algoritmo Pyramid [Gallant 1986] para tratar problemas multiclasses. Assim como o MTower, o algoritmo MPyramid é uma extensão direta do algoritmo original para duas classes. A extensão é feita simplesmente adicionando-se m neurônios ao invés de apenas um a cada passo mantendo, assim, as características do algoritmo original.

O processo de construção da rede pelo algoritmo MPyramid é semelhante ao processo de construção usado no MTower. Considerando que o problema tratado possui m classes, o algoritmo MPyramid adiciona m neurônios em cada camada, assim como faz o MTower. Cada neurônio adicionado, no entanto, recebe conexões de todos os neurônios das camadas anteriores, incluindo a camada de entrada.

A Figura 7.3 ilustra a arquitetura de uma rede criada pelo algoritmo MPyramid. Note que a única diferença em relação à Figura 7.2 (que representa a arquitetura de uma rede MTower) é que todo neurônio de uma determinada camada recebe conexões de todos os neurônios das camadas anteriores.

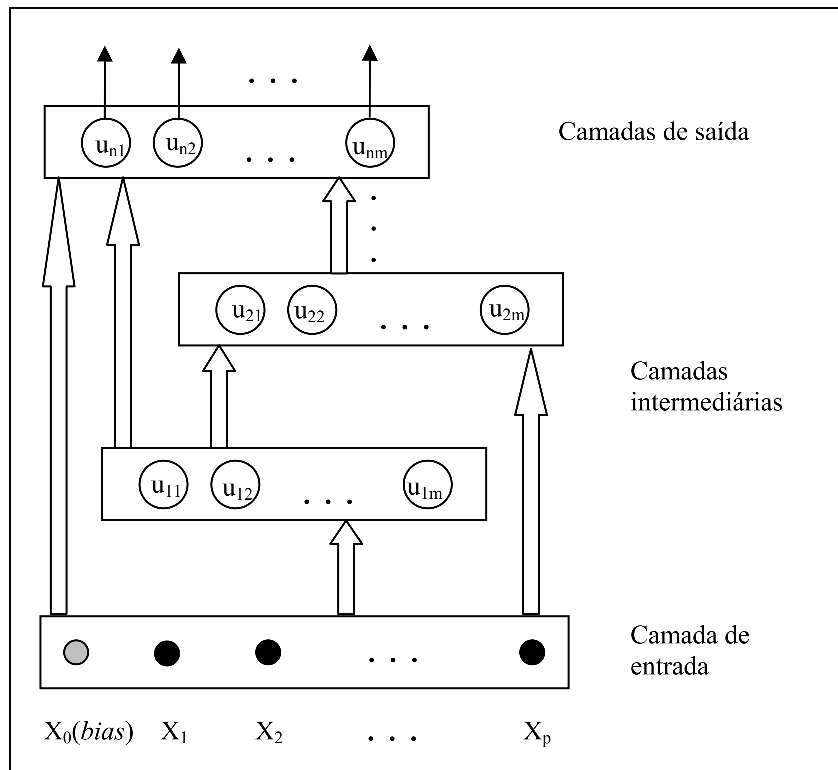


Figura 7.3 – Arquitetura geral de uma rede MPyramid

Considerando que o conjunto de treinamento inicial possui p atributos e m classes, o conjunto de treinamento para os neurônios da i-ésima[25] camada intermediária possuirá $p + (i - 1)m$ atributos, em uma situação que o termo *bias* não é incluído. Desse modo, um neurônio da i-ésima camada possuirá um vetor de pesos com m dimensões a mais do que um neurônio da camada $i - 1$.

A fase de treinamento da rede chega ao fim quando algum dos critérios de parada for satisfeito. Os critérios de parada são os mesmos três usados no algoritmo MTower, ou seja,

[25] Considerando que a primeira camada intermediária possui índice 1.

(1) A rede converge; (2) o número de camadas adicionadas atinge o limite pré-estabelecido ou (3) a adição de uma nova camada piora o desempenho da rede.

O Algoritmo 7.2 ilustra o pseudocódigo do algoritmo MPyramid, que é praticamente idêntico ao do algoritmo MTower. Note, entretanto, que o método *criaTreinamento*() recebe o conjunto de treinamento atual como parâmetro. Dessa forma cada vez que esse método for chamado serão adicionados m atributos ao conjunto de treinamento, correspondente às m saídas dos neurônios da última camada adicionada.

```
Class MPyramid
begin
{Entradas: E - conjunto de treinamento com n instâncias, distribuídas em m (m > 2) classes,
            da forma: (‘x^k_0’, x^k_1, x^k_2,..., x^k_p, C^k), sendo que 1 ≤ k ≤ n
  MAX - número máximo de camadas, pre-determinado}
{Saída: objeto mPyramid que representa a rede neural criada}
camadaMax ← MAX
camada ← 0 {contador para as camadas da rede}
Camadas[camadaMax] {vetor de objetos do tipo NeuronioM que representa as camadas da
rede }

method MPyramid(Matrix E)
  begin
    precisao ← 0
    precisaoAnterior ← 0
    repeat
      begin
        camada ← camada + 1
        Camadas[camada] ← new NeuronioM(E)
        precisaoAnterior ← precisao
        precisao ← precisao(E)
        E ← criaTreinamento(E)
      end
    until (precisaoAnterior ≥ precisao or precisao = 1 or camada = camadaMax)

    if precisaoAnterior ≥ precisao then
      begin
        apaga (Camadas[camada])
        camada ← camada  – 1
      end
  end  {method}

method Matrix criaTreinamento(Matrix E)
  begin
    A ← E
    S ← ⟨0,…,0⟩ {S é um vetor de m posições}
    for i ← 1 to n do
      begin
        S ← Camadas[camada].saidaM(i)
        for k ←  1 to m do
          A[i][p+k] ← S[k]
      end
    return A
  end
end {Class}
```

Algoritmo 7.2 – Pseudocódigo do algoritmo MPyramid

Assim como no MTower o vetor *Camadas*[] armazena os objetos do tipo *NeuronioM* que representa uma camada da rede; note que todo objeto da classe *NeuronioM* terá m TLUs. Assim como no MTower, se o critério de parada da construção da rede for a queda no desempenho causada pela adição da última camada, esta é retirada da rede.

7.4 O Algoritmo Multiclasse MTiling

A extensão do algoritmo Tiling [Mézard & Nadal 1989] para tratar problemas multiclasses, MTiling, foi proposta em [Yang *et al.* 1996]. Assim como na proposta original o algoritmo MTiling constrói uma rede neural multicamadas na qual a primeira camada recebe conexões da camada de entrada e toda camada intermediária subseqüente recebe conexões apenas da camada intermediária imediatamente anterior a ela.

Cada camada é formada por neurônios mestres e auxiliares. As funções dos neurônios mestres e auxiliares no MTiling continuam sendo as mesmas que as correspondentes do algoritmo Tiling, i.e. os neurônios mestres são responsáveis pela classificação das instâncias de treinamento e os neurônios auxiliares são responsáveis por tornar uma camada confiável.

O processo de criação de uma camada é semelhante ao Tiling; diferente do Tiling, entretanto, que adiciona apenas um neurônio mestre a cada camada, o MTiling adiciona m neurônios mestres a cada camada. Os m neurônios mestres são treinados com o conjunto de treinamento formado pelas saídas da camada anterior (ou pelo conjunto original, caso a camada intermediária sendo construída seja a primeira).

Se os m neurônios mestres recém adicionados classificarem corretamente mais instâncias de treinamento que os neurônios mestres da camada anterior, então o algoritmo adiciona neurônios auxiliares até tornar a camada corrente confiável. Se, no entanto, o desempenho da rede piorar com a adição dos últimos m neurônios mestres, então o algoritmo pára e retira da rede o grupo dos m neurônios mestres da camada atual e todos os auxiliares da camada anterior, retornando desse modo, à configuração da última rede obtida.

Como mencionado anteriormente o objetivo dos neurônios auxiliares é tornar uma camada confiável. A confiabilidade de uma camada é condição essencial para a convergência da rede [Mézard & Nadal 1989], pois garante que qualquer duas instâncias de treinamento pertencentes a classes diferentes não produzam a mesma saída em uma determinada camada.

Considere que o algoritmo esteja construindo a camada c, e seja E(c) o conjunto de treinamento formado pelas saídas dos neurônios da camada c – 1, usado para treinar o

neurônio mestre da camada c. Uma representação interna é definida como o conjunto de saídas obtidas por uma camada para uma instância de treinamento. Como mais que uma instância de treinamento pode gerar a mesma representação interna em uma mesma camada, pode-se dizer que a camada c terá q representações internas, sendo que $1 \leq q \leq n_c$, e n_c é o número de instâncias do conjunto de treinamento E(c).

Uma representação interna é confiável se todas as instâncias que geram saída idêntica a essa representação, em uma determinada camada, pertencem à mesma classe. Uma representação interna é não confiável quando pelo menos uma instância de treinamento, pertencente ao conjunto de instâncias que gerou uma determinada representação interna, possuir classe distinta das demais.

Dentre as representações internas não confiáveis geradas na camada corrente c, identifica-se aquela que corresponde ao maior número de instâncias de treinamento. Essas instâncias definem o conjunto de treinamento para treinar o próximo neurônio auxiliar. O conjunto que gerou esta representação interna é o maior conjunto não. Esse processo continua até que a camada c torne-se confiável. Caso o algoritmo exceda o número máximo de neurônios por camada sem que esta se torne confiável, os neurônios auxiliares adicionados à última camada são removidos e o algoritmo termina.

Como o conjunto usado para treinar cada neurônio auxiliar da camada c é um subconjunto do conjunto de treinamento E(c), os conjuntos de treinamento dos neurônios auxiliares poderão não conter todas as m classes. Desse modo, o número de neurônios auxiliares j adicionados a cada vez pode variar no intervalo $2 \leq j \leq m$.

A Figura 7.4 mostra uma possível arquitetura de uma rede MTiling. Para facilitar a visualização, as setas ligando os retângulos que representam uma determinada camada servem para ilustrar conectividade total entre camadas. Os retângulos envolvendo os neurônios representam que estes foram inseridos (e treinados se usada abordagem WTA) juntos.

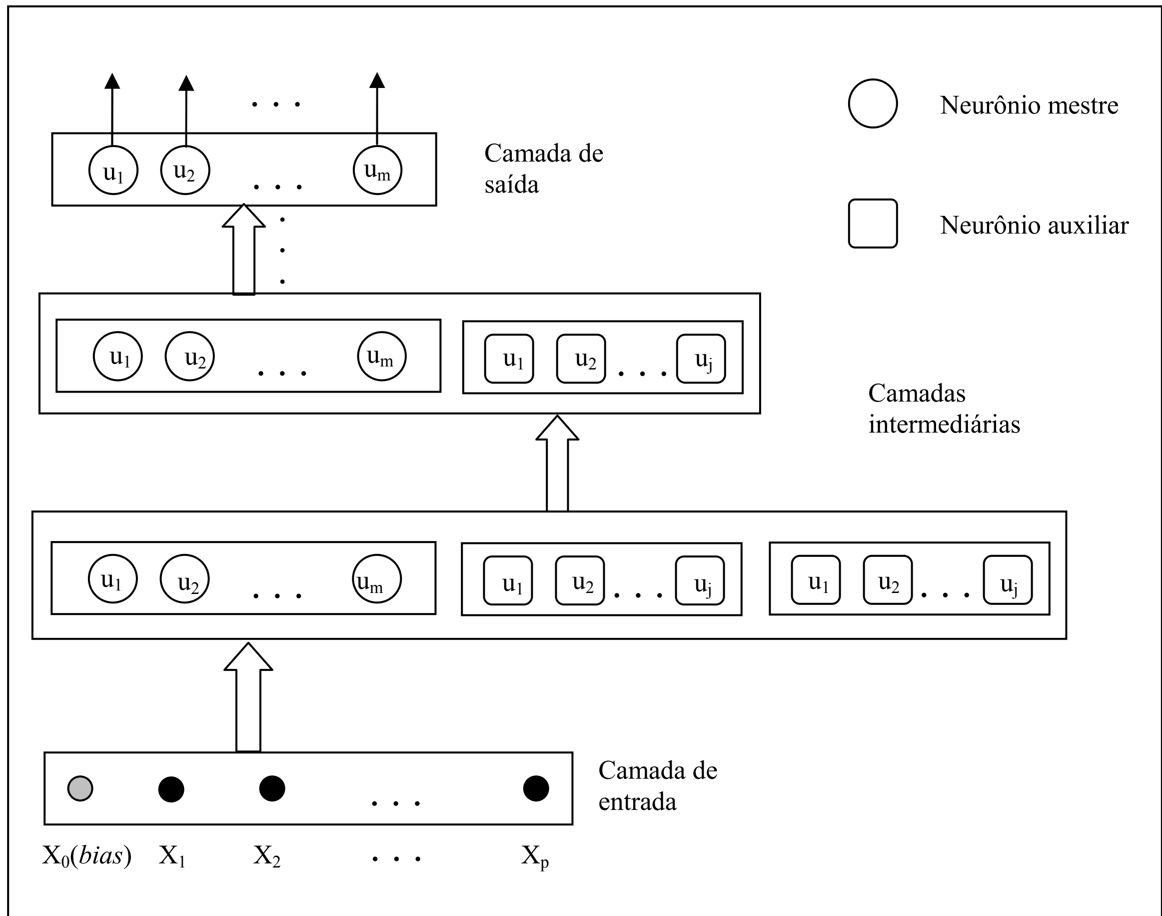


Figura 7.4 – Arquitetura geral de uma rede MTiling

O Algoritmo 7.3 apresenta o pseudocódigo do algoritmo MTiling. Assim como nos pseudocódigos de algoritmo anteriores, a classe *NeuronioM* representa um algoritmo de treinamento de TLUs, seja o PMRWTA ou o BCPWTA bem como algum método para treinar os neurônios de maneira individual e adicioná-los de uma única vez.

O método *criaCamada*() cria uma camada da rede e é chamado recursivamente para criar a rede toda, camada por camada. Para criar a rede o método *criaCamada*() cria o neurônio mestre e chama o método *criaAuxiliares*() para adicionar os neurônios auxiliares à rede. No método *criaAuxiliares*() são adicionados neurônios auxiliares até que o método *confiavel*() retorne verdadeiro ou o número máximo de neurônios auxiliares permitidos por camada tenha sido atingido.

```
Class MTiling
begin
{Entradas: E - conjunto de treinamento com n instâncias, distribuídas em m (m > 2) classes,
            da forma: (‘xk0’, xk1, xk2,..., xkp, Ck), sendo que 1 ≤ k ≤ n
  MAX_CAM - número máximo de camadas, pre-determinado
  MAX_AUX número máximo de neurônios auxiliares por camada}
{Saída: objeto mTiling que representa a rede neural criada}
camada ← 0{ número de camadas intermediarias da rede}
camadaMax  ← MAX_CAM
auxiliarMax ← MAX_AUX
Camadas [camadaMax][auxiliarMax] {matriz de neurônios, onde cada linha representa uma camada
intermediária da rede }

method MTiling (Matrix E)
  begin
    criaCamada(E)
    if precisao() ≠ 1 then
      begin
        apaga(Camadas[camada][1])   {mestre}
        camada ← camada  – 1
        for i ← 2 to auxiliarMax do
          apaga(Camadas[camada][i])  {auxiliares}
      end
  end

method criaCamada(Matrix E)
  begin
    camada ← camada + 1
    neuronio ← new NeuronioM(E)
    Camadas[camada][1] ← neuronio {mestre}
    precisao ← precisao()
      if( precisao > precisaoAnterior and camada < camadaMax and precisão ≠ 1) then
        begin
          criaAuxiliares()
          criaCamada(criaTreinamento())
        end
   end

method criaAuxiliares()
  begin
    auxiliares ← 1
    while (not confiavel() and auxiliares ≤ auxililaresMax) do
      begin
        novoNeurnonio  ← new NeuronioM(criaTreinamentoAux())
        auxiliares ← auxiliares +1
        Camadas[camada][auxiliares] ← novoNeuronio  {auxiliares}
      end
  end
end {Class}
```

Algoritmo 7.3 – Pseudocódigo do algoritmo MTiling

7.4.1 Estratégias de Poda para o Algoritmo MTiling

O processo de construção usado pelo MTiling para criar a rede neural não necessariamente cria uma rede mínima. Como a rede é criada em varias camadas e cada camada é tornada confiável por meio da adição de neurônios auxiliares, pode acontecer que neste processo algum neurônio auxiliar seja redundante.

Como mencionado em [Baum & Hassler 1989], redes neurais pequenas tendem a generalizar melhor que redes maiores. Este fato justifica a prática da poda de redes neurais, que eventualmente fiquem grandes demais para o problema em questão, como pode ser o caso de redes criadas pelo MTiling ou mesmo pelo Tiling. Resultados de algumas pesquisas sobre poda em redes neurais podem ser encontrados em [Reed 1993], [Sietsma & Dow 1988] e [Cun *et al.* 1990].

Para o algoritmo MTiling foram propostos em [Parekh *et al.* 1997b] três estratégias distintas de poda. Todas as três são aplicadas durante a construção da rede, sempre que uma camada se tornar confiável. A poda é realizada somente nos neurônios auxiliares, uma vez que os neurônios mestres são responsáveis pela classificação na rede. De maneira resumida, as estratégias de poda para o MTiling consistem em encontrar neurônios auxiliares na camada recém construída que, na sua ausência, a camada ainda seja confiável. A seguir são apresentadas as três estratégias de poda:

7.4.1.1 Poda de Neurônios Mortos

Neurônios mortos são neurônios auxiliares com a mesma saída (1 ou –1) para todas as instâncias do conjunto de treinamento, como mostra a coluna pontilhada na Tabela 7.1. Note que a retirada desse neurônio não interfere na confiabilidade da camada.

Tabela 7.1 – Exemplo de neurônio morto

Mestre 1	Mestre 2	Mestre 3	Auxiliar 1	Auxiliar 2	Classe
–1	1	–1	–1	1	1
1	–1	–1	–1	1	2
–1	1	–1	–1	–1	2
1	–1	1	–1	–1	1
–1	–1	–1	–1	–1	3

7.4.1.2 Poda de Neurônios Correlacionados

Neurônios correlacionados são pares de neurônios auxiliares que possuem saídas complementares para toda instância no conjunto de treinamento como o Auxiliar 2 e Auxiliar 3 da Tabela 7.2. Quando isso ocorre retira-se um deles e a camada permanece confiável.

Tabela 7.2 – Exemplo de neurônios correlacionados

Mestre 1	Mestre 2	Mestre 3	Auxiliar 1	Auxiliar 2	Auxiliar 3	Classe
–1	1	–1	1	1	–1	1
1	–1	–1	1	–1	1	2
–1	1	–1	1	–1	1	2
1	–1	1	–1	–1	1	1
–1	–1	–1	1	1	–1	3

7.4.1.3 Poda de Neurônios Redundantes

Um neurônio é redundante quando sua eliminação não altera a confiabilidade da rede. Os neurônios identificados nos itens anteriores também são redundantes e por essa razão, podem ser retirados da rede (Auxiliar 1 na Tabela 7.1 e Auxiliar 2 ou Auxiliar 3 da Tabela 7.2). Existem, entretanto, neurônios redundantes que não se caracterizam nem como neurônios mortos nem como neurônios correlacionados (ver Tabela 7.3). Esses neurônios só podem ser encontrados da seguinte maneira: para todo neurônio auxiliar, ignora-se a saída de cada um deles por vez e verifica-se se a camada permanece confiável. Se a camada se manter confiável na ausência do neurônio em questão, ele pode ser retirado da rede. Então o processo recomeça até que todo neurônio auxiliar da rede tenha sido avaliado.

Tabela 7.3 – Exemplo de neurônio redundante

Mestre 1	Mestre 2	Mestre 3	Auxiliar 1	Auxiliar 2	Classe
1	–1	–1	–1	–1	1
1	1	1	–1	–1	2
–1	–1	–1	–1	–1	2
1	–1	–1	1	–1	1
–1	–1	–1	–1	1	3

Note que a estratégia do item 7.4.1.3 engloba as estratégias 7.4.1.1 e 7.4.1.2 e, além disso, encontra neurônios redundantes que não pertencem ao espaço de busca das estratégias 7.4.1.1 e 7.4.1.2. Na verdade, o processo descrito no item 7.4.1.3 encontra todo neurônio redundante existente em uma camada; este processo, no entanto, é mais custoso computacionalmente que os processos descritos nos dois itens anteriores. Uma alternativa para a implementação da poda em redes neurais muito grandes é usar as estratégias 7.4.1.1 e 7.4.1.2 simultaneamente.

7.5 O Algoritmo Multiclasse MUpstart

O algoritmo Upstart [Frean 1990a] para problemas com duas classes visto na Seção 4.5 constrói a rede neural como uma árvore binária de TLUs. Uma extensão natural desse algoritmo para o tratamento de problemas multiclasses seria um algoritmo que construísse m árvores binárias, cada uma delas responsável por uma das m classes encontradas no conjunto de treinamento. Esse método, no entanto, não considera uma possível relação existente entre as m diferentes classes.

Uma abordagem aproximada, porém mais eficiente é a do algoritmo MUpstart, proposto em [Parekh *et al.* 1997a]. O MUpstart, diferente do Upstart, cria os neurônios intermediários em uma única camada intermediária. Todo neurônio intermediário é conectado diretamente a todos os m neurônios da camada de saída. A camada de entrada, por sua vez, é conectada aos neurônios da camada intermediária, bem como a todos os neurônios da camada de saída (ver arquitetura geral de uma rede MUpstart na Figura 7.5).

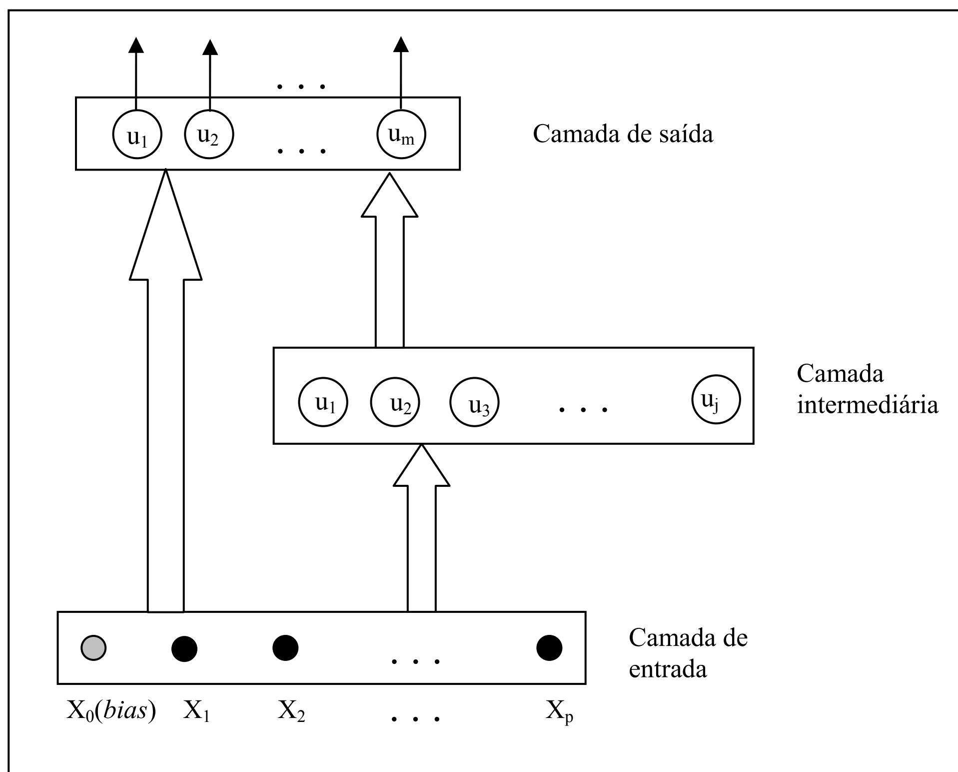


Figura 7.5 – Arquitetura geral de uma rede MUpstart

O MUpstart, assim como o Upstart, distingue entre dois tipos de erros: erro de negativo e erro de positivo. Como descrito no Capítulo 4 (ver Figura 4.5), um erro de positivo ocorre quando uma instância de classe positiva é classificada como sendo de classe negativa,

e um erro de negativo ocorre quando uma instância pertencente à classe negativa é classificada como positiva. Os neurônios no MUpstart, assim como no Upstart, geram 0 ou 1 como saída, diferente de outros algoritmos nos quais os neurônios geram –1 ou 1.

O MUpstart começa a construção da rede treinando os m neurônios da camada de saída. Se esses neurônios classificarem todos os exemplos de treinamento corretamente o algoritmo termina, caso contrário, o algoritmo começa a adicionar neurônios intermediários para a correção dos erros existentes. Os neurônios intermediários são adicionados um a um até que um dos três critérios de parada seja satisfeito, a saber: (1) a rede converge; (2) a adição do último neurônio piora o desempenho da rede; ou (3) o número máximo de neurônios intermediários foi atingido.

O processo de inserção dos neurônios intermediários é baseado nos erros de cada neurônio da camada de saída. Como a camada de saída é composta por m neurônios, seja $D^k = \langle D_1^k, D_2^k, ..., D_m^k \rangle$ a saída esperada para a instância E^k e $O^k = \langle O_1^k, O_2^k, ..., O_m^k \rangle$ a saída obtida pela rede para essa instância. Seja u_j, com $1 \leq j \leq m$, um neurônio da camada de saída. Um erro de positivo cometido pelo neurônio u_j para a instância k, ocorre quando $D_j^k = 1$ e $O_j^k = 0$. De maneira semelhante um erro de negativo ocorre quando $D_j^k = 0$ e $O_j^k = 1$.

Para criar um novo neurônio intermediário o algoritmo encontra o neurônio de saída que comete o maior número de erros (de positivo ou de negativo). Note que um neurônio pode cometer os dois tipos de erro, porém o que se busca é o maior número de um determinado tipo de erro em um neurônio. Suponha que o neurônio u_j tenha sido identificado como o neurônio que cometeu o maior número de erros, e que o tipo do erro cometido seja erro de positivo. Dessa forma o algoritmo deve criar um neurônio para corrigir esses erros.

O conjunto de treinamento que será usado para treinar este novo neurônio (conjunto A) será composto pelas instâncias de treinamento originais, porém com suas classes trocadas de acordo com as saídas do neurônio u_j e as saídas esperadas para cada instância. Para toda instância E^k, com $k \in \{1,..., n\}$, as classes C_A^k do novo conjunto de treinamento são determinadas de acordo com a regra dada em (7.1).

- Se $D_j^k = O_j^k$, então $C^k = -1$
- Se $D_j^k \neq O_j^k$ e $D_j^k = 1$ então $C_A^k = 1$ (7.1)

  senão $C_A^k = -1$

Suponha agora que o neurônio u_j tenha sido identificado como o neurônio que mais cometeu erros, e que os erros cometidos foram erros de negativo. O procedimento seria o mesmo: criar um neurônio para corrigir esse tipo e erro e inserí-lo na camada intermediária. O conjunto de treinamento usado para criar um neurônio que corrija erros de negativo também é o conjunto original de treinamento com suas classes trocadas. As classes, dessa vez, são trocadas de acordo com a regra dada em (7.2).

- Se $D_j^k = O_j^k$, então $C_A^k = -1$
- Se $D_j^k \neq O_j^k$ e $D_j^k = 0$ então $C_A^k = 1$ (7.2)

  senão $C_A^k = -1$

Note que ambos os conjuntos de treinamento para a criação de um novo neurônio intermediário possuem apenas duas classes. Dessa forma, o algoritmo usado para treiná-los deve ser um algoritmo para o treinamento de TLUs com duas classes, como o PMR (Capítulo 2, Seção 2.4) ou o BCPMin (Capítulo 3, Seção 3.4). Cada vez que um neurônio intermediário é adicionado, os neurônios de saída são retreinados, pois agora cada neurônio tem uma entrada adicional. Como conseqüência desta entrada os m vetores de peso de cada neurônio têm suas dimensões acrescidas de um.

O Algoritmo 7.4 apresenta o pseudocódigo orientado a objeto do algoritmo MUpstart. Nele, as classes *NeuronioM* e *Neuronio* são usadas para o treinamento dos neurônios. A classe *NeuronioM*, implementa algum método de treinamento para problemas multiclasses (WTA ou individual), usado neste algoritmo apenas para treinar os neurônios de saída. A classe *Neuronio* implementa algum algoritmo de treinamento de TLUs para duas classes.

Os métodos *neuronioErro*() e *tipoErro*() retornam, respectivamente, o neurônio que mais cometeu erros e o tipo do erro cometido. O índice do neurônio que mais cometeu erro é retornado na variável neuronioErro. Essa variável é passada como parâmetro para um dos métodos (*criaTreinamentoPos*() ou *criaTreinamentoNeg*()) que criará o conjunto de treinamento do próximo neurônio intermediário, dependendo do tipo do erro retornado na variável tipoErro. Note que no método *criaTreinamentoPos*(), mostrado no pseudocódigo, o método *saidaMU*() é usado, e não o método *saidaM*() como nos algoritmos anteriores. Considere o método *saidaMU*() sendo idêntico ao metodo *saidaM*(), no entando, retornando as saídas 0 e 1 e não –1 e 1 como no metodo *saidaM*().

O objeto saida, da classe *NeuronioM*, representa a camada de saída da rede. Antes da camada de saída ser retreinada, sempre que um neurônio intermediário é inserido na rede, o objeto saida é salvo em saidaAnt para o caso da rede piorar com adição do último neurônio. Nesse caso, o último neurônio é retiradado da rede e o objeto saidaAnt é compiado para saida.

O vetor *Camadas*[] representa a camada intermediária da rede. Como mencionado anteriormente, cada vez que um neurônio intermediário é adicionado, os neurônios de saída devem ser retreinados. No pseudocódigo, o método *criaTreinamentoSaida*() cria o treinamento para retreinar os neurônios de saída. Este método adiciona as saídas dos neurônios intermediários (geradas pelo conjunto de treinamento original) ao conjunto de treinamento original mantendo as classes originais inalteradas.

```
Class MUpstart
begin
{Entradas: E - conjunto de treinamento com n instâncias, distribuídas em m (m > 2) classes,
          da forma: ('xk0', xk1, xk2,..., xkp, Ck), sendo que 1 ≤ k ≤ n
  MAX - número máximo de neurônios na camada intermediária, pre-determinado}
{Saída: objeto mUpstart que representa a rede neural criada}
nroNeuronios ← 0{ número de camadas intermediarias da rede}
neuronioMax ← MAX
Camada[neuronioMax] {vetor de neurônios, representa a camada intermediária da rede}
D ← criaSaidaEsperada() {D matriz com as saídas esperadas}

method MUpstart(Matrix E)
  begin
    precisao ← 0
    precisãoAnterior ← 0
    saida ← new NeuronioM(E) {objeto saida é a camada de saída da rede}
    precisao ← precisao()

    while (precisao > precisaoAnterior and precisao ≠ 1 and nroNeuronios ≤ neuronioMax) do
      begin
        precisaoAnterior ← precisao
        neuronioErro ← neuronioErro()
        tipoErro ← tipoErro()
        if tipoErro = erro de positivo then
          begin
            Epos ← criaTreinamentoPos(neuronioErro)
            Camada[nroNeuronios] ← new Neuronio(Epos)
          end
        else if tipoErro = erro de negativo then
            begin
              Eneg ← criaTreinamentoNeg(neuronioErro)
              Camada[nroNeuronios] ← new Neuronio(Eneg)
            end
        nroNeuronios ← nroNeuronios + 1
        Esaida ← criaTreinamentoSaida()
        saidaAnt ← saida
        saida ← NeuronioM(Esaida)   {retreina a saída}
        precisao ← precisao()
      end   {do while}
    if precisao ≤ precisaoAnterior then
      begin
        apaga(Camada[nroNeuronio])
        nroNeuronio ← nroNeuronio – 1
        saida ← saidaAnt   {retoma última saída}
      end
  end

method Matrix criaTreinamentoPos(Integer k)
  begin
    A ← E {A é inicializado com o conjunto de treinamento original sem a classe}
    for i ← 1 to n do
      begin
        S ← saida.saidaMU(i)
        if D[i][k] ≠ S[k] and D[i][k] = 1 then
          C^k_A ← 1
        else
          C^k_A ← –1
      end
    return A
  end
end  {Class}
```

Algoritmo 7.4 – Pseudocódigo do algoritmo MUpstart

7.6 O Algoritmo Multiclasse MPerceptron Cascade

A extensão do algoritmo Perceptron Cascade [Burgess 1994] para problemas multiclasses proposta em [Parekh *et al.* 1995] e chamada de MPerceptron Cascade é bastante semelhante ao MUpstart discutido na Seção 7.5. A diferença entre os dois está na arquitetura da rede neural criada, uma vez que no MPerceptron Cascade, os neurônios intermediários são inseridos em novas camadas, ao invés de serem inseridos em uma única camada, como acontece no MUpstart.

Cada novo neurônio inserido na rede é conectado à camada de saída e recebe conexões da camada de entrada e de todos os neurônios intermediários previamente inseridos, como mostra a Figura 7.6. Dessa forma, cada novo neurônio intermediário inserido possuirá uma dimensão a mais no vetor de pesos do que o do neurônio intermediário adicionado previamente.

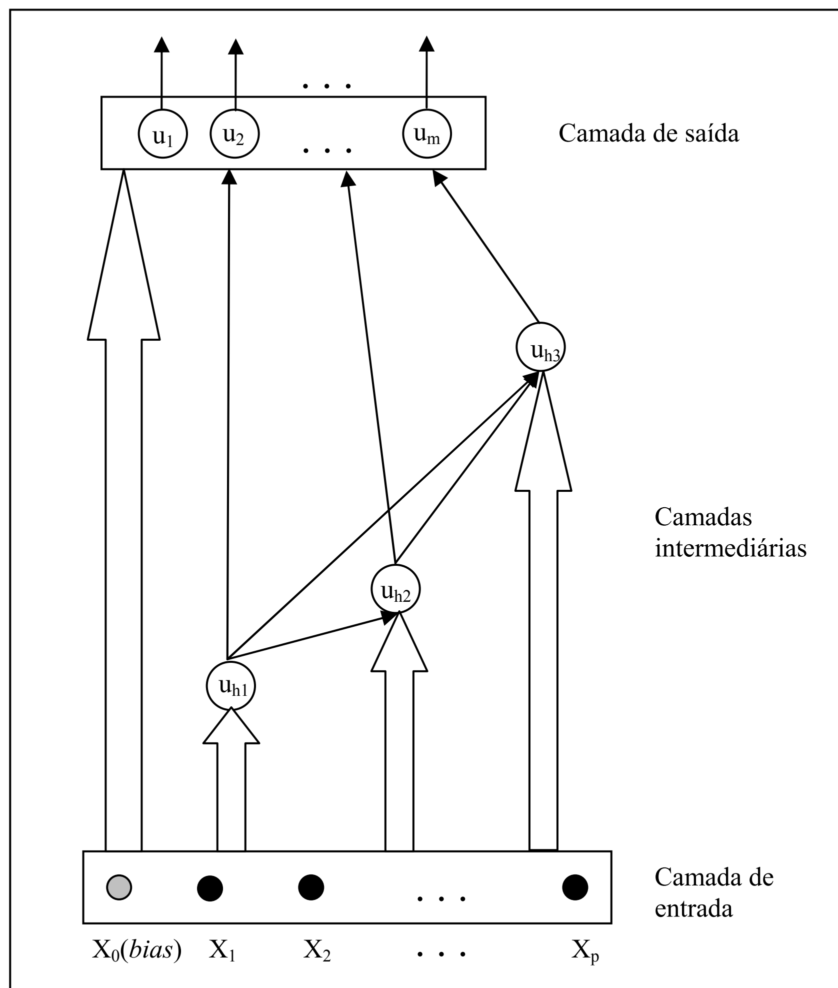


Figura 7.6 – Possível arquitetura de uma rede MPerceptron Cascade, os neurônios intermediários são notados por u_h.

Assim como na versão original para duas classes, a versão multiclasses começa a construção da rede pela camada de saída, treinando os m neurônios de saída com algum algoritmo de treinamento de TLUs para multiclasse. Se a saída atual não classificar todos os exemplos de treinamento corretamente, o algoritmo adiciona neurônios intermediários para corrigir os erros da saída. Os critérios de parada, são os mesmos do MUpstart, i.e. (1) a rede converge; (2) a inserção de um neurônio piora o desempenho da rede ou (3) o número de neurônios intermediários chegou ao seu limite pré-determinado.

O processo de inserção de neurônios intermediários é parecido com aquele do MUpstart. O algoritmo procura por erros de positivo e de negativo dos m neurônios da camada de saída. Um neurônio então é treinado para corrigir o erro mais freqüente existente em um dos m neurônios de saída. Esse neurônio é adicionado em uma camada imediatamente abaixo da camada de saída e é conectado aos m neurônios de saída. Cada vez que um neurônio intermediário é adicionado, o seu vetor de pesos permanece inalterado e os neurônios de saída são retreinados com a adição das saídas do novo neurônio no conjunto de treinamento.

Como os neurônios intermediários são alocados para corrigir um de dois tipos de erro, eles deverão se ativados somente para as instâncias em que o erro ocorreu e deverão permanecer inativos para todas as outras instâncias. Os neurônios intermediários, portanto, assim como no MUpstart, são treinados por algoritmos de treinamento de TLU para duas classes. O conjunto de treinamento usado para treinar um neurônio intermediário dependerá do tipo de erro que este neurônio deverá corrigir.

Desse modo, o conjunto de treinamento (conjunto A) que será usado para treinar este novo neurônio será composto pelas instâncias do conjunto de treinamento original mais as saídas dos neurônios intermediários já adicionados, com as classes determinadas de acordo com as saídas do neurônio, da camada de saída, que cometeu os erros. Caso esse neurônio tenha cometido erros de positivo, as classes devem ser trocadas de acordo com (7.1). Se, no entanto, o maior número de erros cometidos foram erros de negativo, as classes são trocadas de acordo com (7.2).

Para que o procedimento de correção de um tipo de erro funcione, é necessário que os neurônios, adicionados para corrigir esse tipo de erro, não atrapalhem na classificação dos demais exemplos de treinamento. Dessa forma, a saída inativa dos neurônios intermediários, bem como a saída dos neurônios de entrada deve ser 0, como nas versões do Upstart.

O Algoritmo 7.5 apresenta o pseudocódigo do algoritmo MPerceptron Cascade; note que este é idêntico ao algoritmo do MUpstart. A única diferença na criação da rede são os

métodos *criaTreinamentoPos*() e *criaTreinamentoNeg*(), que, neste algoritmo, adicionam as saídas de todos os neurônios intermediários já presentes na rede. O método *criaTreinamentoSaida*(), no entanto, é idêntico ao do MUpstart, pois ambos criam o conjunto de treinamento juntando o conjunto de treinamento original com as saídas de todos os neurônio intermediários da rede, toda vez que um neurônio intermediário é inserido.

O vetor *Camadas*[], que no MUpstart representa uma única camada, aqui representa as camadas intermediária da rede i.e. cada posição do vetor representa uma camada. Os demais métodos, como *neuronioErro*() e *tipoErro*(), são idênticos aos do MUpstart. Também como no MUpstart, somente a camada de saída é objeto da classe *NeuronioM*, os neurônios intermediários são objetos da classe *Neuronio*. Como mencionado anteriormente a primeira classe implementa um algoritmo de treinamento para m TLUs e a segunda, um algoritmo de treinamento de TLUs para duas classes.

Como todo neurônio em uma rede MPerceptron Cascade deve gerar saídas binárias (0 ou 1), as saídas dos neurônios da camada de saída são dadas pelo método *saidaMU*(), como no MUpstart. Já as saídas dos neurônios da camada intermediária são dadas pelo método *saidaU*(); este método, idêntico ao método *saida*() e também membro da classe *Neuronio*, retorna saídas 0 e 1 ao invés de –1 e 1 como faz o método *saida*().

```
Class MPerceptronCascade
begin
{Entradas: E - conjunto de treinamento com n instâncias, distribuídas em m (m > 2) classes,
           da forma: (‘x^k_0’, x^k_1, x^k_2,..., x^k_p, C^k), sendo que 1 ≤ k ≤ n
  MAX - número máximo de neurônios na camada intermediária, pré-determinado}
{Saída: objeto mPerceptronCascade que representa a rede neural criada}
nroNeuronios ← 0 {número de camadas intermediárias da rede}
neuronioMax ← MAX
Camadas[neuronioMax] {vetor de neurônios, representa a camada intermediária da rede}

method MPerceptronCascade(Matrix E)
  begin
    precisao ← 0
    precisãoAnterior ← 0
    saida ← new NeuronioM(E) {objeto saida é a camada de saída da rede, formada por m neurônios}
    precisao ← precisao()

    while (precisao > precisaoAnterior and precisao ≠ 1 and nroNeuronios ≤ neuronioMax) do
      begin
        precisaoAnterior ← precisao
        neuronioErro ← neuronioErro()
        tipoErro ← tipoErro()
        if tipoErro = erro de positivo then
          begin
            Epos ← criaTreinamentoPos(neuronioErro)
            Camadas[nroNeuronios] ← new Neuronio(Epos)
          end
        else if tipoErro = erro de negativo then
            begin
              Eneg ← criaTreinamentoNeg(neuronioErro)
              Camadas[nroNeuronios] ← new Neuronio(Eneg)
            end
        nroNeuronios← nroNeuronios+ 1
        Esaida ← criaTreinamentoSaida()
        saidaAnt ← saida
        saida ← NeuronioM(Esaida)   {retreina a saída}
        precisao ← precisao()
      end
    if precisao ≤ precisaoAnterior then
      begin
        apaga(Camadas[nroNeuronios])
        nroNeuronios ← nroNeuronios – 1
        saida ← saidaAnt
      end
  end   {method}

method Matrix criaTreinamentoPos(Integer k)
  begin
    A ← E {A é inicializado com o conjunto de treinamento original sem a classe}
    for i ← 1 to n do
      begin
        for j ← 1 to nroNeuronios do
          A[p+j] ← Camadas[j].saidaU(i)
        S ← saida.saidaMU(i)
        if D[i][k] ≠ S[k] and D[i][k] = 1 then
          C^k_A ← 1
        else
          C^k_A ← –1
      end
    return A
  end
end  {Class}
```

Algoritmo 7.5 – Pseudocódigo do algoritmo MPerceptron Cascade

8 capítulo

Uma Avaliação Empírica do Aprendizado Neural Construtivo em Diferentes Domínios de Conhecimento

8.1 Introdução

Neste capítulo são apresentados e discutidos os resultados das avaliações empíricas realizadas com os algoritmos apresentados neste trabalho em vários domínios de conhecimento. Os resultados foram obtidos usando validação cruzada 10 (10-*fold cross validation*) e estão organizados por domínio nas Tabelas 8.1 a 8.120.

Toda tabela apresenta os valores em porcentagem das precisões de treinamento e de teste, com os respectivos desvios padrões seguidas do maior e menor valores obtidos durante o treinamento e teste, respectivamente. As tabelas também mostram o número médio de neurônios, (e respectivo desvio padrão) seguido do maior e do menor número de neurônios obtidos na criação de uma rede. Os algoritmos que constroem a rede em camadas com mais do que um neurônio nelas mostram também o número médio de camadas (e respectivo desvio padrão), bem como os valores referentes ao maior e menor número de camadas obtidas.

O capítulo é dividido em três seções. Na Seção 8.2 são descritos os experimentos realizados com os algoritmos para treinamento de TLUs bem como com os algoritmos construtivos para problemas com duas classes. Os resultados relativos aos algoritmos construtivos têm duas versões: a que usa o PMR e a que usa o BCPMin como algoritmo básico para o treinamento de TLUs.

Na Seção 8.3 são descritos os experimentos realizados com as extensões para o tratamento de problemas multiclasses, dos algoritmos Tower, Pyramid, Tiling, Upstart e Perceptron Cascade.

Os resultados são usados para subsidiar uma avaliação quanto ao desempenho e comportamento de cada algoritmo, em cada domínio de conhecimento.

8.2 Avaliações de Algoritmos para Treinamento de TLUs e de Algoritmos Neurais Construtivos em Nove Domínios com Duas Classes

Esta seção apresenta os resultados das avaliações dos algoritmos para treinamento de TLUs, discutidos nos Capítulos 2 e 3, bem como os resultados dos algoritmos neurais construtivos para duas classes, apresentados nos Capítulos 4 e 5. No que segue, cada subseção focaliza um domínio de conhecimento, na seguinte ordem: Paridade-5, Sistema Vestibular, Ionosphere, Liver, Pima, Breast Cancer (WPBC), Monks1, Monks2 e Monks3. Uma comparação entre os algoritmos Tower, Pyramid, Shift e Perceptron Cascade envolvendo os domínios de conhecimento previamente citados pode ser encontrada em [Bertini *et al.* 2006].

Cada uma das subseções apresenta uma breve descrição do domínio e, então, apresenta uma seqüência de dez tabelas. A primeira delas agrega os resultados dos algoritmos para treinamento de uma TLU, a saber: PMR, BCPMin, MinOver e Thermal Modificado, respectivamente. As tabelas seguintes apresentam os resultados dos algoritmos Tower, Pyramid, Upstart, Shift, Perceptron Cascade, Tiling, PTI, OffTiling e Sequential respectivamente. A tabela referente ao Tiling apresenta também os resultados do Tiling Híbrido (ver Capítulo 4, Subseção 4.4.1). Em cada tabela são mostrados os resultados do respectivo algoritmo construtivo usando o PMR e usando o BCPMin, exeto para as tabelas referentes aos resultados do algoritmo Sequential, no qual são usados os algoritmos BCPMax e IncLp.

No Perceptron e em algoritmos baseados no Perceptron o processamento de um exemplo de treinamento é, muitas vezes, considerado uma iteração. Já algoritmos similares ao BCP, entretanto, sempre atuam sobre todos os exemplos de treinamento, como um todo. A maneira de viabilizar uma comparação não tendenciosa entre essas duas abordagens, adotada neste trabalho foi a de considerar uma época (notada nas tabelas como ep) do Perceptron como equivalente a uma iteração do BCPMin.

As tabelas referentes aos algoritmos Upstart, OffTiling e Sequential não têm os resultados referentes ao N° de Iterações de 10^3ep (com o PMR). Ainda, as tabelas referentes aos algoritmos Upstart e Sequential não têm os resultados referentes ao N° de iterações de 10^3 (com a versão BCP). Os resultados da versão do Sequential com o IncLp, para 10^2ep e 10^3ep, também não são apresentados. Esses dados não foram obtidos experimentalmente em virtude do excessivo tempo de processamento. Nas tabelas os melhores resultados obtidos no treinamento e no teste, bem como o menor número de neurônios criados estão realçados em

negrito. A Tabela 8.1 apresenta o número total de instâncias, o número de instâncias por classe e o número de atributos em cada um dos nove domínios utilizados.

Tabela 8.1 – Especificação dos domínios de conhecimento

Domínio	N° Total Instâncias	N° de Instâncias Positivas	N° de Instâncias Negativas	N° Atributos
Paridade-5	32	16	16	5
Vestibular	198	101	97	6
Iono	351	225	126	34
Liver	345	145	200	6
Pima	768	268	500	8
WPBC	198	47	151	33
Monks1	432	216	216	6
Monks2	432	142	290	6
Monks3	432	228	204	6

8.2.1 O Domínio Paridade-5

O domínio Paridade-5 é um domínio artificial cujas instâncias descrevem o conceito de paridade-5 ímpar. O domínio possui 32 instâncias e todas elas são descritas por 5 atributos boolenanos com uma classe associada. A classe de uma instância é função do número de valores ‘1’ que a descrevem. A classe será 1 se o número de valores ‘1’ que a descrevem for par e será –1 caso seja ímpar. As Tabelas 8.2 a 8.11 mostram os resultados obtidos no domínio Paridade-5.

Tabela 8.2 – PMR, BCPMin, MinOver e Thermal Modificado / Paridade-5

Nº de Iterações	Precisão de Treinamento	Precisão de Teste	Maior Precisão de Treinamento	Menor Precisão de Treinamento	Maior Precisão de Teste	Menor Precisão de Teste
PMR						
1ep	65,3~5,0	55,8~22,9	75,8	58,6	100,0	33,3
10ep	71,2~3,6	**58.3~17,6**	75,9	65,5	75,0	33,3
10^2ep	**74,6~1,6**	46,6~15,3	75,9	72,4	66,6	33,3
10^3ep	**74,6~1,6**	46,6~15,3	75,9	72,4	66,6	33,3
BCPMin						
1	62,8~3.4	35,0~19,6	68,9	58,6	66,7	0,0
10	66,7~1.8	35,8~32,9	69,0	64,3	75,0	0,0
10^2	68,7~3.3	43,3~30,6	75,9	65,5	100,0	0,0
10^3	**69,8~3.0**	**53,3~21,9**	75,9	65,5	100,0	33,3
MinOver						
1ep	58,0~2.9	34,2~16,9	62,1	55,2	66,7	0,0
10ep	57,3~4.6	37,5~19,7	65,5	50,0	66,7	0,0
10^2ep	58,7~5.3	**60,0~11,6**	68,9	51,7	66,7	33,3
10^3ep	**59,0~4.3**	39,2~26,1	65,5	51,7	75,0	0,0
Themal Modificado						
1ep	65,9~4.0	58,3~23,7	72,4	58,6	100,0	33,3
10ep	67,7~4.1	62,5~25,2	72,4	62,1	100,0	33,3
10^2ep	69,5~3.4	53,3~34,9	75,9	62,1	100,0	33,3
10^3ep	**69,8~1.6**	**75,0~20,8**	72,4	67,9	100,0	33,3

Tabela 8.3 – Tower / Paridade-5

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Tower / PMR									
1ep	66,9~5,7	59,2~21,7	**1,6~0,8**	75,8	58,6	100,0	33,3	3,0	1,0
10ep	**100,0~0,0**	71,7~29,2	6,8~3,3	100,0	100,0	100,0	0,0	12,0	2,0
10^2ep	**100,0~0,0**	74,2~26,2	3,1~0,3	100,0	100,0	100,0	33,3	4,0	3,0
10^3ep	**100,0~0,0**	**80,8~22,9**	3,0~0,0	100,0	100,0	100,0	33,3	3,0	3,0
Tower / BCPMin									
1	67,4~3,4	30,0~23,3	**3,5~0,8**	72,4	62,1	66,7	0,0	5,0	2,0
10	88,9~12,4	**54,2~32,7**	4,9~2,0	100,0	69,0	100,0	0,0	7,0	2,0
10^2	94,8~11,3	48,3~35,5	4,2~1,6	100,0	69,0	100,0	0,0	8,0	2,0
10^3	**96,2~9,0**	45,8~23,3	4,7~1,3	100,0	72,4	75,0	0,0	6,0	2,0

Tabela 8.4 – Pyramid / Paridade-5

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Pyramid / PMR									
1ep	64,2~6,5	68,3~16,1	**1,7~0,8**	72,4	53,6	100,0	33,3	3,0	1,0
10ep	91,7~8,2	77,5~15,7	3,7~0,7	100,0	72,4	100,0	66,7	5,0	3,0
10^2ep	**100,0~0,0**	80,0~28,1	3,0~0,0	100,0	100,0	100,0	33,3	3,0	3,0
10^3ep	**100,0~0,0**	**86,7~23,3**	3,0~0,0	100,0	100,0	100,0	33,3	3,0	3,0
Pyramid / BCPMin									
1	60,1~4,6	30,8~25,4	**1,6~0,5**	68,9	55,1	66,7	0,0	2,0	1,0
10	62,1~7,9	42,5~15,9	1,9~0,7	75,9	55,2	75,0	33,3	3,0	1,0
10^2	62,9~9,0	**43,3~14,1**	2,0~0,7	75,9	51,7	66,7	33,3	3,0	1,0
10^3	**66,0~7,2**	37,5~12,0	1,7~0,7	75,9	51,7	66,7	25,0	3,0	1,0

Tabela 8.5 – Upstart / Paridade-5

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Upstart / PMR									
1ep	60,4~6,3	51,7~19,6	44,1~23,6	68,9	50,0	75,0	33,3	62,0	8,0
10ep	94,1~6,5	**76,7~22,5**	7,7~1,9	100,0	79,3	100,0	33,3	10,0	3,0
10^2ep	**95,8~7,0**	75,8~23,7	**5,4~2,2**	100,0	82,8	100,0	33,3	7,0	2,0
Upstart / BCPMin									
1	67,4~3,3	36,7~23,3	10,5~4,3	72,4	62,1	66,7	0,0	17,0	5,0
10	79,2~12,4	51,7~22,8	**6,3~2,3**	96,5	58,6	100,0	33,3	10,0	4,0
10^2	**91,4~10,6**	**53,3~24,9**	6,8~1,5	100,0	68,9	100,0	25,0	9,0	5,0

Tabela 8.6 – Shift / Paridade-5

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Shift / PMR									
1ep	74,3~6,7	59,2~26,8	**2,1~1,1**	89,6	64,2	100,0	33,3	5,0	1,0
10ep	97,2~4,8	76,7~27,4	4,0~1,6	100,0	86,2	100,0	33,3	8,0	3,0
10^2ep	**100,0~0,0**	**90,0~16,1**	3,0~0,0	100,0	100,0	100,0	66,7	3,0	3,0
10^3ep	**100,0~0,0**	84,2~16,8	3,0~0,0	100,0	100,0	100,0	66,7	3,0	3,0
Shift / BCPMin									
1	78,3~5,2	33,3~13,6	5,2~2,6	96,6	65,5	50,0	0,0	9,0	2,0
10	**100,0~0,0**	43,3~30,6	4,8~1,6	100,0	100,0	100,0	0,0	7,0	2,0
10^2	98,9~3,3	46,7~29,4	5,1~2,2	100,0	89,6	100,0	0,0	9,0	3,0
10^3	**100,0~0,0**	**65,0~35,5**	**4,1~1,7**	100,0	100,0	100,0	0,0	8,0	3,0

Tabela 8.7 – Perceptron Cascade / Paridade-5

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Perceptron Cascade / PMR									
1ep	70,8~4,8	**68,3~27,4**	**1,5~1,0**	82,8	64,3	100,0	33,3	4,0	1,0
10ep	84,0~8,7	61,7~24,9	2,2~0,8	100,0	72,4	100,0	33,3	3,0	1,0
10^2ep	85,4~7,2	56,7~26,3	1,7~0,5	93,1	75,0	100,0	33,3	2,0	1,0
10^3ep	**91,3~3,7**	59,2~26,8	2,2~0,4	96,6	86,2	100,0	33,3	3,0	2,0
Perceptron Cascade / BCPMin									
1	66,0~7,7	40,0~19,6	**2,4~1,0**	85,7	58,6	66,7	0,0	4,0	1,0
10	90,6~8,6	47,5~28,3	3,5~1,6	100,0	75,8	100,0	25,0	7,0	2,0
10^2	**100,0~0,0**	**61,7~27,0**	3,5~1,1	100,0	100,0	100,0	33,3	6,0	3,0
10^3	99,3~2,3	57,5~27,9	3,8~1,6	100,0	92,9	100,0	33,3	7,0	2,0

Tabela 8.8 – Tiling / Paridade-5

Nº It.	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de Camadas (CA)	Maior CA	Menor CA
Tiling / PMR												
1ep	65,6~7,5	**55,8~27,8**	**1,0~0,0**	75,9	48,3	100,0	33,3	1,0	1,0	1,0~0,0	1,0	1,0
10ep	**100,0~0,0**	53,3~19,3	10,6~3,8	100,0	100,0	75,0	25,0	18,0	5,0	2,8~0,8	4,0	2,0
10^2ep	**100,0~0,0**	49,2~17,8	6,1~1,4	100,0	100,0	75,0	33,3	9,0	5,0	2,1~0,3	3,0	2,0
10^3ep	**100,0~0,0**	52,5~17,6	6,0~1,6	100,0	100,0	75,0	33,3	10,0	5,0	2,1~0,3	3,0	2,0
Tiling / BCPMin												
1	82,8~18,5	30,0~17,2	8,9~6,6	100,0	58,6	50,0	0,0	19,0	1,0	2,5~1,3	4,0	1,0
10	97,3~8,7	**50,0~25,2**	7,9~3,7	100,0	72,4	100,0	25,0	15,0	1,0	2,5~0,8	4,0	1,0
10^2	**100,0~0,0**	30,8~20,1	7,3~1,7	100,0	100,0	66,7	0,0	10,0	5,0	2,3~0,5	3,0	2,0
10^3	**100,0~0,0**	39,2~13,6	**6,7~0,9**	100,0	100,0	75,0	33,3	8,0	6,0	2,0~0,0	2,0	2,0
Tiling Híbrido												
1ep	82,6~18,6	42,5~27,3	8,1~5,7	100,0	58,6	75,0	0,0	15,0	1,0	2,5~1,2	4,0	1,0
10ep	**100,0~0,0**	46,7~24,9	7,0~2,2	100,0	100,0	75,0	0,0	11,0	4,0	2,4~0,5	3,0	2,0
10^2ep	97,2~8,7	**51,7~27,7**	**5,2~1,9**	100,0	72,4	100,0	33,3	8,0	1,0	2,0~0,5	3,0	1,0
10^3ep	**100,0~0,0**	38,3~21,9	5,5~1,4	100,0	100,0	75,0	0,0	8,0	4,0	2,0~0,0	2,0	2,0

Tabela 8.9 – PTI / Paridade-5

Nº It.	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de Camadas (CA)	Maior CA	Menor CA
PTI / PMR												
1ep	67,7~10,9	**65,8~23,1**	**10,1~12,2**	86,2	51,7	100,0	33,3	29,0	1,0	1,4~0,5	2,0	1,0
10ep	**98,9~3,4**	65,0~27,7	14,7~12,3	100,0	89,3	100,0	33,3	49,0	7,0	2,7~0,5	3,0	2,0
10^2ep	89,9~13,1	52,5~29,6	18,6~29,6	100,0	72,4	75,0	33,3	83,0	1,0	1,7~0,7	3,0	1,0
10^3ep	89,9~13,1	52,5~17,6	16,2~26,5	100,0	72,4	75,2	33,3	68,0	1,0	1,7~0,7	3,0	1,0
PTI / BCPMin												
1	84,3~15,2	34,2~23,1	11,4~9,4	100,0	62,1	66,7	0,0	29,0	1,0	2,5~1,4	5,0	1,0
10	89,5~16,9	27,5~27,2	8,8~7,3	100,0	62,1	66,7	0,0	24,0	1,0	2,4~1,2	4,0	1,0
10^2	**96,9~9,8**	**36,7~28,1**	9,4~3,4	100,0	68,9	66,7	0,0	13,0	1,0	2,7~0,7	3,0	1,0
10^3	94,5~11,7	35,8~24,2	**8,8~5,3**	100,0	68,9	75,0	0,0	17,0	1,0	2,3~0,8	3,0	1,0

Tabela 8.10 – OffTiling / Paridade-5

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
OffTiling / PMR									
1ep	93,4~12,3	**45,0~36,7**	126,1~72,3	100,0	65,0	100,0	0,0	212,0	47,0
10ep	**100,0~0,0**	38,3~26,9	19,6~21,4	100,0	100,0	75,0	0,0	68,0	7,0
10^2ep	**100,0~0,0**	37,5~37,5	**8,1~1,8**	100,0	100,0	100,0	0,0	12,0	6,0
OffTiling / BCPMin									
1	64,6~5,9	40,8~31,0	10,4~1,3	72,4	51,7	100,0	0,0	13,0	8,0
10	84,3~19,1	41,7~23,5	9,0~1,2	100,0	51,7	66,7	0,0	10,0	7,0
10^2	94,8~10,1	52,5~17,6	7,5~1,2	100,0	68,9	75,0	33,0	9,0	6,0
10^3	**95,2~11,3**	**58,3~23,6**	**6,8~0,6**	100,0	65,5	100,0	33,3	8,0	6,0

Tabela 8.11 – Sequential / Paridade-5

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Sequential / BCPMax									
1	58,4~4,9	**49,2~23,7**	14,2~3,1	65,5	51,7	75,0	0,0	19,0	9,0
10	60,7~7,9	45,8~23,3	10,4~2,7	79,3	51,7	75,0	0,0	14,0	6,0
10^2	**65,3~11,7**	43,3~32,8	**8,7~1,6**	96,6	55,2	100,0	0,0	11,0	7,0
Sequential / IncLp									
10ep	**53,1~1,6**	**53,3~15,3**	**11,5~1,2**	55,2	51,7	66,7	33,3	13,0	10,0

Observando os resultados apresentados nas Tabelas 8.2 a 8.11, pode-se notar que o algoritmo que obteve o melhor desempenho, com relação à precisão de teste foi o Shift usando o PMR. Outros algoritmos que tiveram um desempenho razoável nos testes foram o Tower, o Pyramid, e o Upstart todos em suas versões PMR. Nota-se que, na maioria dos algoritmos, os desempenhos no treinamento e no teste foram pior quando usado o algoritmo BCPMin ao invés do PMR; a maior diferença, com relação ao uso desses algoritmos, pode ser verificada na Tabela 8.4, relativa ao algoritmo Pyramid.

As três versões do algoritmo Tiling (Tilig/PMR, Tiling/BCP e Tiling Híbrido) obtiveram bons resultados no treinamento; porém esse resultado não se refletiu nos testes; o que leva a conjecturar que esses algoritmos estejam fazendo *overfitting*[26]. Não pode ser esquecido, entretanto, que o domínio Paridade-5 tem apenas 32 instâncias e cada conjunto de teste é constituído de 3 (ou 4) instâncias apenas.

Como pode ser notado na Tabela 8.10, linha referente a 1ep, o número excessivo de neurônios (126,1~72,3) criados pelo OffTiling se deve ao fato do algoritmo ter sido executado apenas uma época. Pode-se notar na mesma tabela que, à medida que o número de épocas aumenta, o número de neurônios criados diminui.

8.2.2 O Domínio Sistema Vestibular (SV)

Cada instância do domínio Sistema Vestibular descreve dados de um paciente e foram obtidas por meio do teste sacádico fixo, realizado pela equipe do Prof. Dr. José Colafemina do Serviço de Otoneurologia do Hospital das Clínicas da Escola de Medicina da Universidade de São Paulo em Ribeirão Preto [Volpini *et al.* 2002].

Movimentos sacádicos são produzidos quando um paciente tem que olhar, sem mover a cabeça, para um ponto de luz que se alterna, com uma freqüência constante, entre as extremidades de uma barra eletrônica colocada horizontalmente na sua frente. Os potenciais

[26] Situação na qual a rede é treinada em excesso e perde o poder de generalização.

elétricos produzidos pelos movimentos sacádicos são medidos com a ajuda de alguns eletrodos, posicionados próximos aos olhos. Como essas informações podem sugerir problemas no Sistema Vestibular do paciente, as classes que caracterizam as instâncias são: positiva (101 instâncias) e negativa (97 instâncias).

Resultados referentes ao uso dos algoritmos neurais construtivos Tower, Pyramid e DistAl [Yang *et al.* 1998], bem como ao uso do backpropagation [Rumelhart *et al.* 1986], e uma de suas variantes [Moller 1993], na detecção de problemas no sistema vestibular podem ser encontrados em [Figueira *et al.* 2006]. As Tabelas 8.12 e 8.21 mostram os resultados dos experimentos usando o domínio Sistema Vestibular.

Tabela 8.12 – PMR, BCPMin, MinOver e Thermal Modificado / Vestibular

Nº de Iterações	Precisão de Treinamento	Precisão de Teste	Maior Precisão de Treinamento	Menor Precisão de Treinamento	Maior Precisão de Teste	Menor Precisão de Teste
PMR						
1ep	89,9~1,3	**90,8~9,7**	92,7	88,7	100,0	70,0
10ep	90,8~1,7	86,4~16,0	93,8	89,3	100,0	55,0
10^2ep	**91,4~1,8**	86,9~16,0	94,9	89,9	100,0	55,0
10^3ep	**91,4~1,8**	87,9~12,7	94,9	89,9	100,0	65,0
BCPMin						
1	91,4~1,1	**88,9~10,3**	93,3	89,9	100,0	70,0
10	91,7~1,2	87,9~14,2	94,4	90,5	100,0	60,0
10^2	**92,4~1,7**	85,9~15,8	96,1	91,0	100,0	55,0
10^3	92,2~1,9	85,9~15,9	96,1	90,5	100,0	55,0
MinOver						
1ep	88,3~2,1	**89,3~10,2**	91,6	83,7	100,0	70,0
10ep	90,7~1,6	88,4~12,1	94,9	89,3	100,0	65,0
10^2ep	91,2~1,5	87,4~12,8	94,9	89,9	100,0	65,0
10^3ep	**91,4~1,1**	88,4~12,9	93,3	89,9	100,0	65,0
Themal Modificado						
1ep	**54,6~3,3**	11,5~31,5	56,7	45,8	100,0	0,0
10ep	53,7~4,3	21,5~41,6	56,7	45,5	100,0	0,0
10^2ep	52,8~5,0	**31,5~47,5**	56,7	45,5	100,0	0,0
10^3ep	53,7~4,3	21,5~41,6	56,7	45,5	100,0	0,0

Tabela 8.13 – Tower / Vestibular

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Tower / PMR									
1ep	90,2~1,6	**88,4~14,1**	**1,5~1,0**	93,3	88,2	100,0	55,0	4,0	1,0
10ep	90,9~1,7	85,8~16,7	1,7~0,5	94,4	89,3	100,0	55,0	2,0	1,0
10^2ep	91,4~1,8	86,4~14,7	2,0~0,5	94,9	89,9	100,0	60,0	3,0	1,0
10^3ep	**91,5~1,7**	86,4~16,0	2,3~0,5	94,9	89,9	100,0	55,0	3,0	2,0
Tower / BCPMin									
1	91,5~0,9	**89,4~10,2**	1,9~0,3	93,3	90,4	100,0	70,0	2,0	1,0
10	92,0~1,6	86,4~16,0	1,8~0,6	95,5	90,5	100,0	55,0	3,0	1,0
10^2	**92,1~1,8**	85,4~16,1	1,8~0,4	96,1	90,5	100,0	55,0	2,0	1,0
10^3	**92,1~1,8**	86,4~14,9	**1,7~0,5**	96,1	90,5	100,0	60,0	2,0	1,0

Tabela 8.14 – Pyramid / Vestibular

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Pyramid / PMR									
1ep	90,3~1,5	**87,9~14,4**	1,9~0,9	93,3	88,8	100,0	55,0	4,0	1,0
10ep	90,9~1,9	86,4~17,0	2,1~0,3	94,9	89,3	100,0	55,0	3,0	2,0
10^2ep	91,4~1,8	86,9~16,0	**1,9~0,6**	94,9	89,9	100,0	55,0	3,0	1,0
10^3ep	**91,7~1,6**	86,9~14,9	2,6~0,7	95,9	90,4	100,0	60,0	4,0	2,0
Pyramid / BCPMin									
1	91,3~1,2	**88,9~10,3**	1,8~0,4	93,3	89,9	100,0	70,0	2,0	1,0
10	91,5~1,8	86,9~16,2	1,8~0,4	95,5	89,4	100,0	55,0	2,0	1,0
10^2	91,7~2,1	86,4~16,0	1,8~0,4	96,1	89,9	100,0	55,0	2,0	1,0
10^3	**92,1~1,8**	85,3~16,1	**1,7~0,5**	96,1	90,5	100,0	55,0	2,0	1,0

Tabela 8.15 – Upstart / Vestibular

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Upstart / PMR									
1ep	90,0~1,3	**88,9~11,3**	28,3~27,7	92,7	88,2	100,0	65,0	56,0	2,0
10ep	90,8~1,7	85,9~16,8	33,9~27,5	93,8	89,3	100,0	55,0	56,0	2,0
10^2ep	**91,4~1,8**	86,9~14,7	**12,7~22,6**	94,9	89,9	100,0	60,0	56,0	2,0
Upstart / BCPMin									
1	91,5~0,9	**88,9~10,3**	26,5~7,3	93,3	90,4	100,0	70,0	35,0	16,0
10	91,9~1,5	85,4~16,1	30,0~18,3	95,5	90,4	100,0	55,0	60,0	12,0
10^2	**92,0~1,7**	86,4~16,0	**21,1~11,3**	95,5	90,4	100,0	55,0	38,0	10,0

Tabela 8.16 – Shift / Vestibular

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Shift / PMR									
1ep	90,6~1,5	**86,9~15,1**	**1,5~0,7**	93,3	89,3	100,0	60,0	3,0	1,0
10ep	92,5~1,6	83,4~22,6	2,4~0,7	96,1	91,0	100,0	30,0	3,0	1,0
10^2ep	**93,0~1,9**	82,4~22,2	2,1~0,7	97,2	90,4	100,0	30,0	3,0	1,0
10^3ep	92,7~2,3	84,9~23,3	1,7~0,7	97,2	90,0	100,0	25,0	3,0	1,0
Shift / BCPMin									
1	93,7~1,5	83,9~26,6	2,5~0,7	97,2	92,1	100,0	10,0	4,0	2,0
10	93,7~1,5	81,9~25,2	2,0~0,5	97,2	92,1	100,0	20,0	3,0	1,0
10^2	**94,2~1,5**	**86,9~19,2**	**2,0~0,0**	97,2	92,7	100,0	45,0	2,0	2,0
10^3	94,0~1,8	85,9~20,0	2,2~0,4	97,8	92,7	100,0	40,0	3,0	2,0

Tabela 8.17 – Perceptron Cascade / Vestibular

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Perceptron Cascade / PMR									
1ep	90,5~1,5	**86,9~15,1**	**1,8~0,8**	93,3	88,8	100,0	55,0	3,0	1,0
10ep	90,9~1,6	86,4~17,0	2,0~0,5	93,8	89,3	100,0	55,0	3,0	1,0
10^2ep	91,4~1,8	86,4~16,0	1,9~0,6	94,9	89,9	100,0	55,0	3,0	1,0
10^3ep	**92,1~1,6**	86,9~16,0	2,6~1,2	94,9	90,5	100,0	55,0	5,0	1,0
Perceptron Cascade / BCPMin									
1	91,4~1,0	**88,9~10,3**	1,9~0,3	93,3	90,5	100,0	70,0	2,0	1,0
10	92,1~1,6	86,4~16,0	2,0~0,7	95,5	90,5	100,0	55,0	3,0	1,0
10^2	92,1~1,8	85,9~15,8	**1,7~0,5**	96,1	90,4	100,0	60,0	2,0	1,0
10^3	**92,3~1,8**	85,4~16,1	1,9~0,3	96,1	90,5	100,0	55,0	2,0	1,0

Tabela 8.18 – Tiling / Vestibular

Nº It.	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de Camadas (CA)	Maior CA	Menor CA
Tiling / PMR												
1ep	89,7~1,8	88,4~12,5	**1,0~0,0**	92,7	86,5	100,0	60,0	1,0	1,0	1,0~0,0	1,0	1,0
10ep	90,8~1,5	86,4~16,9	**1,0~0,0**	93,8	89,3	100,0	55,0	1,0	1,0	1,0~0,0	1,0	1,0
10^2ep	**91,4~1,8**	86,4~16,0	**1,0~0,0**	94,9	89,9	100,0	55,0	1,0	1,0	1,0~0,0	1,0	1,0
10^3ep	**91,4~1,8**	**86,9~16,0**	**1,0~0,0**	94,9	89,9	100,0	55,0	1,0	1,0	1,0~0,0	1,0	1,0
Tiling / BCPMin												
1	95,4~3,8	84,9~26,8	15,5~9,1	100,0	90,5	100,0	15,0	28,0	1,0	2,4~0,9	4,0	1,0
10	98,7~3,0	84,9~22,3	**14,1~6,3**	100,0	91,0	100,0	25,0	21,0	1,0	2,8~0,8	4,0	1,0
10^2	**100,0~0,0**	**86,4~18,5**	16,6~4,0	100,0	100,0	100,0	40,0	21,0	10,0	3,0~0,0	3,0	3,0
10^3	99,1~2,8	85,0~22,6	14,4~5,7	100,0	91,0	100,0	25,0	19,0	1,0	2,9~0,7	4,0	1,0
Tiling Híbrido												
1ep	94,3~4,1	83,4~28,4	11,7~8,4	100,0	90,5	100,0	10,0	25,0	1,0	2,2~1,0	4,0	1,0
10ep	**99,4~1,8**	85,9~16,3	15,4~3,5	100,0	94,4	100,0	45,0	20,0	9,0	3,0~0,5	4,0	2,0
10^2ep	97,1~4,0	83,9~19,9	**8,8~6,1**	100,0	91,0	100,0	35,0	17,0	1,0	2,3~0,9	3,0	1,0
10^3ep	98,2~3,8	**86,4~24,1**	11,5~6,6	100,0	90,5	100,0	20,0	18,0	1,0	2,7~1,1	4,0	1,0

Tabela 8.19 – PTI / Vestibular

Nº It.	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de Camadas (CA)	Maior CA	Menor CA
PTI / PMR												
1ep	90,1~1,5	**88,4~13,3**	**1,0~0,0**	93,8	88,2	100,0	55,0	1,0	1,0	1,0~0,0	1,0	1,0
10ep	90,7~1,7	87,4~14,0	**1,0~0,0**	94,4	89,3	100,0	60,0	1,0	1,0	1,0~0,0	1,0	1,0
10^2ep	91,2~1,7	87,4~15,1	**1,0~0,0**	94,9	89,9	100,0	55,0	1,0	1,0	1,0~0,0	1,0	1,0
10^3ep	**91,4~1,8**	86,9~16,0	**1,0~0,0**	94,9	89,9	100,0	55,0	1,0	1,0	1,0~0,0	1,0	1,0
PTI / BCPMin												
1	91,8~1,1	**89,4~10,4**	**19,8~33,2**	93,3	90,5	100,0	70,0	97,0	1,0	1,3~0,5	2,0	1,0
10	95,1~3,8	86,4~16,2	21,8~25,1	100,0	90,4	100,0	55,0	73,0	1,0	2,0~1,2	4,0	1,0
10^2	97,4~3,9	85,4~20,6	21,7~14,5	100,0	90,5	100,0	35,0	47,0	1,0	2,4~0,9	4,0	1,0
10^3	**98,7~3,0**	83,4~26,8	24,8~10,5	100,0	91,0	100,0	10,0	37,0	1,0	2,9~1,1	4,0	1,0

Tabela 8.20 – OffTiling / Vestibular

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
OffTiling / PMR									
1ep	96,8~3,9	82,5~3,9	189,0~53,2	100,0	87,6	100,0	5,0	216,0	75,0
10ep	**100,0~0,0**	**86,9~16,5**	24,7~6,1	100,0	100,0	100,0	50,0	32,0	13,0
10^2ep	**100,0~0,0**	85,4~23,6	**15,6~3,8**	100,0	100,0	100,0	25,0	21,0	10,0
OffTiling / BCPMin									
1	**93,3~2,8**	84,4~25,5	15,4~2,6	98,9	87,6	100,0	15,0	18,0	11,0
10	92,8~9,1	**89,5~16,5**	14,7~3,2	99,4	67,9	100,0	50,0	18,0	8,0
10^2	92,9~11,6	86,4~20,8	13,7~3,2	100,0	60,3	100,0	30,0	17,0	8,0
10^3	92,6~11,2	74,9~31,4	**13,0~2,7**	100,0	64,1	95,0	5,0	16,0	10,0

Tabela 8.21 – Sequential / Vestibular

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Sequential / BCPMax									
1	61,6~4,5	**60,0~43,3**	25,8~6,3	67,8	56,2	100,0	0,0	34,0	15,0
10	**61,8~4,3**	55,0~40,9	24,5~5,1	67,5	56,7	100,0	0,0	31,0	13,0
10^2	61,6~4,4	55,0~40,9	**24,2~5,2**	67,6	56,7	100,0	0,0	29,0	13,0
Sequential / IncLp									
10ep	**51,0~5,7**	**51,5~51,3**	**4,0~0,5**	56,7	45,5	100,0	0,0	5,0	3,0

As precisões em treinamento e teste de quase todos os algoritmos foram particularmente boas no domínio SV. Quando o tamanho da rede construída, entretanto, for levado em consideração, as melhores opções talvez sejam as redes construídas pelo Tower ou Pyramid (usando o PMR ou o BCPMin), dado que são menores.

Os algoritmos Thermal Modificado e Sequential tiveram um desempenho bastante ruim tanto no treinamento quanto no teste. Como será evidenciado ao longo desse capítulo, o Thermal Modificado vai se mostrar um algoritmo não muito eficiente nos vários domínios abordados. O desempenho ruim do Sequential talvez se deva à maneira do algoritmo construir a rede, uma vez que, na construção não é levado em consideração o desempenho da rede.

De maneira semelhante ao ocorrido no domínio Paridade-5, o OffTiling com N° de Iterações de 1ep criou um rede (em média) com 189,0~53,2 neurônios. Pode ser notado, também, que à medida que o número de épocas cresce, o número de neurônios criado diminui.

8.2.3 O Domínio Ionosphere

O dominio Ionosphere, extraído do *UCI Repository* [Blake & Merz 1998] contém dados captados por um sistema de radar (ver [Sigillito *et al.* 1989] para mais detalhes) usado para identificar algum tipo de estrutura na ionosfera. O domínio possui 351 instâncias. Cada instância é formada por dois pulsos do radar com 17 entradas cada um, que totalizam 34 entradas por instância. Cada instância é caracterizada, também, por uma classe associada que, quando positiva, indica evidência de alguma estrutura conhecida na ionosfera. As Tabelas 8.22 a 8.31 mostram os resultados dos experimentos no domínio Ionosphere.

Tabela 8.22 – PMR, BCPMin, MinOver e Thermal Modificado / Ionosphere

Nº de Iterações	Precisão de Treinamento	Precisão de Teste	Maior Precisão de Treinamento	Menor Precisão de Treinamento	Maior Precisão de Teste	Menor Precisão de Teste
PMR						
1ep	87,3~1,6	82,3~10,3	89,2	85,1	97,1	71,4
10ep	92,3~1,2	85,8~9,6	94,3	90,5	100,0	72,2
10^2ep	95,1~0,9	87,2~10,0	96,5	93,6	100,0	66,6
10^3ep	**95,6~1,1**	**88,1~9,8**	97,5	94,3	100,0	72,2
BCPMin						
1	84,3~1,8	77,8~11,6	86,7	81,3	94,3	54,3
10	91,1~1,2	**85,5~9,1**	92,7	89,2	100,0	71,4
10^2	92,2~1,0	84,0~9,9	93,7	90,5	97,1	68,6
10^3	**92,3~0,7**	84,6~9,3	93,4	91,5	100,0	68,6
MinOver						
1ep	91,3~0,7	84,7~10,1	92,4	90,2	100,0	66,7
10ep	93,4~1,2	86,9~8,4	95,6	91,8	100,0	75,0
10^2ep	94,6~1,2	**87,8~9,1**	96,5	92,7	100,0	71,4
10^3ep	**94,9~1,2**	87,2~8,7	96,8	93,4	100,0	71,4

Themal Modificado						
1ep	67,1~3,5	66,9~16,9	72,5	61,7	100,0	48,6
10ep	68,4~3,3	66,7~19,4	73,7	62,6	100,0	51,4
10^2ep	81,2~10,1	77,3~17,0	93,0	67,4	100,0	54,3
10^3ep	**84,6~10,9**	**80,4~9,1**	93,7	64,3	97,1	65,7

Tabela 8.23 – Tower / Ionosphere

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Tower / PMR									
1ep	92,1~2,2	85,5~10,6	**3,9~1,2**	94,3	88,3	100,0	71,4	5,0	2,0
10ep	96,7~1,3	**88,4~9,5**	6,0~1,3	98,7	94,9	100,0	72,2	8,0	4,0
10^2ep	98,2~0,7	86,1~10,2	6,1~1,7	99,1	97,5	100,0	69,4	8,0	3,0
10^3ep	**99,0~0,9**	86,3~7,8	5,3~1,6	100,0	97,2	97,1	77,1	8,0	4,0
Tower / BCPMin									
1	84,2~1,6	77,8~10,4	**2,5~0,7**	85,7	81,3	92,2	60,0	4,0	2,0
10	95,1~1,3	**87,8~6,8**	5,2~1,2	96,8	92,1	97,1	77,1	8,0	4,0
10^2	95,9~1,7	86,9~6,1	6,4~1,7	97,8	93,0	97,1	77,1	8,0	4,0
10^3	**96,7~1,6**	86,3~5,6	6,8~1,4	100,0	94,0	94,4	80,0	9,0	5,0

Tabela 8.24 – Pyramid / Ionosphere

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Pyramid / PMR									
1ep	92,7~2,4	85,5~9,8	**5,1~2,3**	95,3	88,0	100,0	72,2	9,0	1,0
10ep	96,9~0,8	**89,2~9,3**	6,1~1,2	97,8	95,3	100,0	72,2	8,0	4,0
10^2ep	98,7~0,7	85,2~7,8	6,1~2,8	99,4	97,5	100,0	75,0	11,0	4,0
10^3ep	**99,9~0,2**	85,6~8,1	5,8~1,2	100,0	99,4	97,1	71,4	8,0	4,0
Pyramid / BCPMin									
1	83,9~1,6	77,8~11,8	2,1~0,7	85,4	80,7	94,3	54,3	3,0	1,0
10	90,5~1,3	**86,1~7,7**	1,6~0,5	92,7	88,3	97,1	77,1	2,0	1,0
10^2	91,5~1,4	84,1~8,6	2,1~0,6	93,4	88,9	100,0	71,4	3,0	1,0
10^3	**91,9~1,2**	84,6~9,2	**1,5~0,5**	93,7	90,5	97,1	68,6	2,0	1,0

Tabela 8.25 – Upstart / Ionosphere

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Upstart / PMR									
1ep	88,7~1,6	80,1~8,9	**51,4~16,7**	90,8	86,0	94,2	65,7	59,0	4,0
10ep	**97,4~1,0**	**87,5~9,9**	52,8~17,6	98,7	96,2	97,1	65,7	62,0	3,0
10^2ep	95,2~2,6	81,8~12,2	95,9~29,5	98,7	91,5	97,1	65,7	107,0	12,0
Upstart / BCPMin									
1	85,0~1,6	80,1~8,8	100,0~0,0	86,4	81,6	94,3	71,4	101,0	100,0
10	95,3~1,2	**86,6~6,1**	46,9~24,6	97,2	93,9	94,3	77,8	101,0	28,0
10^2	**95,5~0,8**	85,2~7,9	**29,2~13,3**	96,8	94,3	97,1	74,3	58,0	15,0

Tabela 8.26 – Shift / Ionosphere

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Shift / PMR									
1ep	94,6~1,9	88,6~6,9	4,5~1,6	96,8	90,5	97,1	80,0	7,0	2,0
10ep	97,4~0,8	**90,0~5,0**	**3,1~1,0**	98,7	96,2	100,0	82,8	5,0	2,0
10^2ep	98,9~0,7	87,5~9,4	3,5~0,9	99,7	98,1	100,0	72,2	5,0	2,0
10^3ep	**99,5~0,4**	84,3~6,7	3,4~1,2	99,7	98,4	97,1	74,3	6,0	2,0

Shift / BCPMin									
1	91,2~1,6	81,8~10,9	**2,6~0,7**	94,6	89,5	97,1	62,9	4,0	2,0
10	94,3~1,5	85,8~7,4	2,8~0,8	96,8	92,4	97,1	77,1	4,0	2,0
10^2	**96,3~1,4**	**87,5~9,2**	2,9~0,9	98,4	94,0	97,1	71,4	4,0	2,0
10^3	95,8~1,6	86,9~7,8	2,7~1,0	98,1	93,7	97,4	74,3	5,0	2,0

Tabela 8.27 – Perceptron Cascade / Ionosphere

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Perceptron Cascade / PMR									
1ep	89,1~1,5	83,5~9,3	1,8~0,6	91,1	86,4	100,0	71,4	3,0	1,0
10ep	92,7~1,2	86,9~8,1	**1,4~0,7**	94,9	91,2	100,0	77,1	3,0	1,0
10^2ep	95,2~1,1	**87,2~8,1**	1,7~0,7	97,2	94,0	100,0	72,2	3,0	1,0
10^3ep	**96,3~1,6**	86,9~9,7	1,8~0,8	99,7	94,6	100,0	74,3	3,0	1,0
Perceptron Cascade / BCPMin									
1	84,4~1,4	80,3~8,8	**1,6~0,5**	85,8	82,3	94,3	68,6	2,0	1,0
10	96,9~2,5	**88,6~8,0**	4,4~2,0	99,1	90,8	100,0	80,0	8,0	1,0
10^2	**98,8~1,2**	86,1~10,5	4,6~2,0	100,0	95,9	100,0	69,4	8,0	2,0
10^3	98,2~2,3	84,9~10,4	4,4~2,4	100,0	94,3	100,0	68,6	9,0	1,0

Tabela 8.28 – Tiling / Ionosphere

Nº It.	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de Camadas (CA)	Maior CA	Menor CA
Tiling / PMR												
1ep	87,5~1,3	82,9~11,7	**1,0~0,0**	89,9	86,1	100,0	68,5	1,0	1,0	1,0~0,0	1,0	1,0
10ep	96,5~3,3	84,1~8,5	54,8~49,8	100,0	92,1	97,1	74,3	148,0	1,0	3,1~1,7	5,0	1,0
10^2ep	97,2~3,0	84,4~10,7	6,0~5,9	100,0	93,4	100,0	69,4	17,0	1,0	2,0~1,2	4,0	1,0
10^3ep	**98,2~2,3**	**87,5~8,1**	4,9~3,6	100,0	94,3	100,0	75,0	10,0	1,0	2,0~0,9	3,0	1,0
Tiling / BCPMin												
1	88,7~6,1	81,8~10,2	17,5~21,3	100,0	84,8	100,0	68,6	44,0	1,0	2,1~1,5	5,0	1,0
10	97,1~3,9	85,8~8,3	17,4~12,1	100,0	90,8	100,0	77,1	38,0	1,0	2,6~1,1	4,0	1,0
10^2	97,9~3,5	**88,3~6,5**	**13,3~6,6**	100,0	91,5	97,1	77,1	22,0	1,0	2,6~0,8	4,0	1,0
10^3	**98,6~2,9**	86,0~9,7	13,4~9,9	100,0	93,0	100,0	68,6	28,0	1,0	2,8~1,2	4,0	1,0
Tiling Híbrido												
1ep	87,6~5,5	82,6~8,8	12,0~15,1	100,0	81,6	94,3	68,5	40,0	1,0	1,7~1,1	4,0	1,0
10ep	95,2~4,3	87,2~9,6	**9,5~7,7**	100,0	90,5	100,0	74,3	19,0	1,0	2,3~1,3	4,0	1,0
10^2ep	**98,4~3,3**	85,2~10,1	10,1~5,2	100,0	91,1	100,0	71,4	17,0	1,0	2,9~1,0	4,0	1,0
10^3ep	97,8~3,6	**87,2~6,8**	9,9~6,1	100,0	91,5	97,1	77,1	17,0	1,00	2,6~0,9	4,0	1,0

Tabela 8.29 – PTI / Ionosphere

Nº It.	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de Camadas (CA)	Maior CA	Menor CA
PTI / PMR												
1ep	87,2~1,6	81,8~9,5	**1,0~0,0**	89,6	84,8	97,1	69,4	1,0	1,0	1,0~0,0	1,0	1,0
10ep	93,1~2,2	84,9~9,5	1,8~2,5	98,4	90,8	97,1	72,2	9,0	1,0	1,1~0,3	2,0	1,0
10^2ep	96,6~2,9	**86,9~9,2**	8,2~10,8	100,0	93,7	100,0	69,4	32,0	1,0	1,7~0,9	3,0	1,0
10^3ep	**96,9~2,4**	86,6~7,5	3,3~3,7	100,0	94,3	100,0	75,0	9,0	1,0	1,5~0,8	3,0	1,0
PTI / BCPMin												
1	87,5~4,3	77,2~10,7	23,3~31,4	93,4	80,7	91,4	54,3	88,0	1,0	1,4~0,5	2,0	1,0
10	95,8~4,7	87,8~7,6	20,2~21,5	100,0	90,2	100,0	75,0	66,0	1,0	2,4~1,5	5,0	1,0
10^2	**97,7~3,7**	**88,3~7,7**	10,7~8,5	100,0	91,5	100,0	77,1	29,0	1,0	2,6~1,2	4,0	1,0
10^3	96,6~3,9	86,6~8,4	**6,6~5,0**	100,0	90,5	97,1	68,5	13,0	1,0	1,7~0,7	3,0	1,0

Tabela 8.30 – OffTiling / Ionosphere

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
OffTiling / PMR									
1ep	84,6~12,7	75,5~11,9	221,1~2,9	98,1	63,3	94,2	54,3	227,0	100,0
10ep	99,9~0,2	86,4~10,3	81,9~54,7	100,0	99,3	100,0	71,4	215,0	25,0
10^2ep	**100,0~0,0**	**88,0~6,3**	**17,6~6,9**	100,0	100,0	97,1	77,1	31,0	11,0
OffTiling / BCPMin									
1	78,3~12,6	70,4~10,4	24,0~4,0	91,5	58,5	82,9	54,3	29,0	18,0
10	91,8~9,1	86,1~7,9	16,5~4,5	98,1	67,1	97,1	74,3	24,0	12,0
10^2	**97,2~2,6**	**86,6~10,0**	11,2~3,1	100,0	93,7	100,0	62,8	18,0	8,0
10^3	93,3~6,9	82,9~11,3	**10,7~2,0**	100,0	79,7	97,1	60,0	14,0	7,0

Tabela 8.31 – Sequential / Ionosphere

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Sequential / BCPMax									
1	**75,5~1,9**	**73,8~16,4**	24,8~3,7	77,5	72,7	100,0	54,3	30,0	17,0
10	72,5~9,4	73,8~17,2	24,8~3,7	78,2	46,5	100,0	51,4	30,0	20,0
10^2	71,2~13,9	70,9~19,2	**23,8~4,3**	87,7	45,6	100,0	45,7	33,0	18,0
Sequential / IncLp									
10ep	**64,1~2,5**	**64,1~22,4**	**3,0~0,0**	65,8	60,1	100,0	48,6	3,0	3,0

Nas Tabelas 8.22 a 8.31 pode ser observado que no domínio Ionosphere todos os algoritmos tiveram um bom desempenho, sendo que o Thermal Modificado e o Sequential tiveram os piores desempenhos dentre todos.

Na Tabela 8.28 relativa ao Tiling usando o PMR, pode ser evidenciado que com 1ep apenas, o algoritmo Tiling induziu uma rede com 1 neurônio e, consequentemente, o Tiling nessa situação é o próprio PMR. Já com 10ep, o Tiling em média criou redes com 3,1~1,7 camadas com um total de 58,8~49,8 em média neurônios (mestres e auxiliares).

8.2.4 O Domínio Liver

O domínio Liver, extraído do *UCI Repository* [Blake & Merz 1998] contém 345 instâncias. Cada instância descreve dados de um paciente e é composta por 6 atributos, sendo que 5 deles são índices de análises sanguíneas e um representa a quantidade média de álcool ingerida por dia. Tais instâncias são divididas em duas classes, positiva (indica a presença de doenças no fígado), e negativa (indica a ausência de doenças no fígado). As Tabelas 8.32 a 8.41 mostram os resultados dos experimentos no domínio Liver.

Tabela 8.32 – PMR, BCPMin, MinOver e Thermal Modificado / Liver

Nº de Iterações	Precisão de Treinamento	Precisão de Teste	Maior Precisão de Treinamento	Menor Precisão de Treinamento	Maior Precisão de Teste	Menor Precisão de Teste
PMR						
1ep	64,7~5,6	62,9~14,5	72,3	57,9	94,1	47,0
10ep	71,7~1,2	67,0~8,8	73,3	70,1	79,4	54,3
10^2ep	72,7~1,1	**69,3~7,5**	75,2	71,3	82,9	54,3
10^3ep	**73,5~0,7**	66,9~9,4	74,5	72,3	85,7	51,4
BCPMin						
1	66,8~1,7	61,3~15,5	68,7	64,3	91,2	44,1
10	72,2~2,8	64,4~10,0	74,9	65,2	79,4	48,6
10^2	73,9~1,2	**67,0~10,7**	75,6	72,0	80,0	42,9
10^3	**74,4~1,5**	65,2~11,7	76,1	71,3	82,9	40,0
MinOver						
1ep	64,5~3,2	59,2~10,4	70,9	60,5	82,4	50,0
10ep	70,3~1,3	**68,1~10,3**	72,0	68,1	82,8	54,3
10^2ep	71,4~1,5	64,0~12,8	74,8	69,0	88,6	47,1
10^3ep	**71,8~1,4**	65,5~7,1	75,5	70,3	76,5	54,3
Themal Modificado						
1ep	49,8~8,9	30,6~22,7	62,9	40,8	76,5	5,7
10ep	50,8~8,8	42,3~25,8	60,8	37,1	85,7	5,7
10^2ep	**51,8~8,7**	**65,4~21,7**	60,8	37,1	94,3	32,5
10^3ep	47,9~8,6	53,5~26,8	59,2	37,1	85,7	5,7

Tabela 8.33 – Tower / Liver

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Tower / PMR									
1ep	66,8~2,8	60,8~9,4	**1,5~0,7**	70,6	61,1	76,5	47,1	3,0	1,0
10ep	72,6~1,7	66,7~7,8	1,7~0,7	75,5	70,1	76,5	55,8	3,0	1,0
10^2ep	74,2~1,0	66,7~12,1	3,4~1,4	76,2	72,9	82,8	40,0	6,0	1,0
10^3ep	**78,2~1,5**	**67,6~7,3**	7,9~1,7	80,7	75,6	79,4	57,1	10,0	5,0
Tower / BCPMin									
1	67,5~1,4	62,9~16,5	1,9~0,9	69,0	65,8	94,1	41,7	4,0	1,0
10	73,1~1,7	**67,6~8,9**	1,8~1,0	75,2	69,7	85,3	54,3	3,0	1,0
10^2	74,6~1,3	63,8~12,2	**1,8~0,6**	76,5	72,6	80,0	40,0	3,0	1,0
10^3	**75,5~1,9**	64,1~10,8	2,6~2,0	78,5	72,9	82,4	42,9	7,0	1,0

Tabela 8.34 – Pyramid / Liver

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Pyramid / PMR									
1ep	69,2~2,3	67,8~14,4	2,1~0,9	71,9	65,9	97,1	44,1	4,0	1,0
10ep	72,6~1,4	**69,0~11,2**	**2,0~0,9**	74,5	69,7	85,7	48,6	3,0	1,0
10^2ep	74,5~0,8	66,7~10,1	4,6~1,5	76,2	73,2	85,7	54,3	7,0	2,0
10^3ep	**77,6~2,0**	66,7~8,8	7,1~2,4	81,3	75,2	82,9	48,6	11,0	4,0
Pyramid / BCPMin									
1	67,4~1,8	62,7~17,2	**1,7~1,2**	70,0	63,5	94,1	41,2	4,0	1,0
10	72,2~2,2	62,4~11,0	1,9~0,7	74,3	67,4	82,4	48,6	3,0	1,0
10^2	72,8~1,5	64,7~11,9	2,0~0,7	75,5	71,1	82,4	47,1	3,0	1,0
10^3	**73,7~1,6**	**65,3~9,2**	2,0~0,8	76,1	70,7	79,4	51,4	3,0	1,0

Tabela 8.35 – Upstart / Liver

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Upstart / PMR									
1ep	66,9~4,6	59,8~17,5	51,6~17,7	74,8	60,3	82,9	22,8	63,0	2,0
10ep	71,7~1,6	65,8~11,4	56,2~2,7	74,8	69,4	82,9	42,9	60,0	53,0
10^2ep	**72,9~0,9**	**67,0~8,6**	**56,4~1,9**	74,3	71,7	88,6	57,1	59,0	53,0
Upstart / BCPMin									
1	67,0~1,6	61,8~15,7	**100,0~0,0**	69,4	64,9	91,2	44,1	100,0	100,0
10	70,4~2,9	63,5~11,1	**100,0~0,0**	73,9	66,1	82,4	44,1	100,0	100,0
10^2	**73,9~1,2**	**64,7~7,2**	**100,0~0,0**	76,2	72,3	79,4	52,9	100,0	100,0

Tabela 8.36 – Shift / Liver

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Shift / PMR									
1ep	67,4~2,9	56,3~9,7	1,9~1,0	72,3	63,5	73,5	37,1	4,0	1,0
10ep	74,8~0,9	66,1~6,0	5,0~1,4	75,8	72,9	77,1	58,8	7,0	3,0
10^2ep	78,0~1,7	65,5~5,7	7,1~2,8	80,6	74,6	74,3	58,8	11,0	3,0
10^3ep	**79,5~2,0**	**66,4~7,7**	**7,7~2,5**	83,3	76,2	82,4	55,8	12,0	4,0
Shift / BCPMin									
1	71,3~2,6	**66,1~7,9**	2,1~0,7	76,1	66,5	79,4	54,3	3,0	1,0
10	72,7~1,6	63,8~12,9	**1,8~0,8**	75,2	70,4	79,4	34,3	3,0	1,0
10^2	75,3~1,3	64,4~13,2	2,2~0,6	77,7	73,3	82,9	40,0	3,0	1,0
10^3	**75,3~1,0**	63,8~10,2	1,9~0,6	77,1	74,2	80,0	42,9	3,0	1,0

Tabela 8.37 – Perceptron Cascade / Liver

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Perceptron Cascade / PMR									
1ep	67,0~3,0	**69,6~14,4**	1,9~0,7	71,4	62,6	85,7	42,9	3,0	1,09
10ep	72,6~1,4	67,5~8,0	2,0~0,8	75,2	70,7	77,1	54,3	4,0	1,0
10^2ep	73,0~1,1	66,1~7,9	**1,7~1,1**	74,5	71,4	80,0	57,1	4,0	1,0
10^3ep	**75,7~2,2**	66,1~7,6	3,5~2,3	79,7	73,3	74,3	51,4	7,0	1,0
Perceptron Cascade / BCPMin									
1	67,5~1,9	61,2~15,9	1,8~0,8	70,0	65,0	91,2	44,1	3,0	1,0
10	72,5~2,7	**66,2~8,5**	**1,6~0,7**	75,5	65,5	74,3	48,6	3,0	1,0
10^2	74,4~1,0	63,8~11,1	1,8~0,9	75,8	72,9	82,4	42,8	4,0	1,0
10^3	**74,9~1,1**	62,1~7,9	1,9~0,9	75,9	72,9	71,4	48,6	3,0	1,0

Tabela 8.38 – Tiling / Liver

Nº It.	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de (CA)	Maior CA	Menor CA
Tiling / PMR												
1ep	67,1~2,4	59,2~16,6	**1,0~0,0**	70,4	62,9	82,4	22,8	1,0	1,0	1,0~0,0	1,0	1,0
10ep	71,5~1,2	**68,1~9,0**	**1,0~0,0**	73,0	69,0	85,7	55,9	1,0	1,0	1,0~0,0	1,0	1,0
10^2ep	72,8~1,0	65,5~9,8	**1,0~0,0**	74,2	71,3	85,7	51,4	1,0	1,0	1,0~0,0	1,0	1,0
10^3ep	**97,2~4,8**	62,3~10,7	104,7~16,3	100,0	86,7	82,4	44,1	126,0	68,0	6,2~1,5	8,0	4,0
Tiling / BCPMin												
1	73,9~3,5	**63,9~11,9**	106,1~68,8	80,0	69,5	82,4	42,9	220,0	36,0	3,7~1,9	7,0	2,0
10	88,6~10,4	62,6~10,2	105,9~43,6	100,0	75,5	80,0	42,9	164,0	36,0	4,8~1,9	7,0	2,0
10^2	92,7~8,7	62,0~10,3	84,3~25,7	100,0	80,7	77,1	48,6	124,0	39,0	4,3~1,4	6,0	2,0
10^3	**94,0~8,3**	58,3~11,5	**82,1~23,1**	100,0	78,4	74,3	41,2	109,0	39,0	4,6~1,5	6,0	2,0
Tiling Híbrido												
1ep	74,4~2,1	**62,6~11,9**	98,0~39,2	76,8	71,4	82,4	44,1	189,0	44,0	3,5~1,1	6,0	2,0
10ep	89,5~12,0	59,8~9,2	102,3~60,4	100,0	70,6	70,6	40,0	196,0	1,0	4,9~2,5	9,0	1,0
10^2ep	**94,7~7,8**	57,1~6,2	95,8~18,9	100,0	79,7	67,6	45,7	128,0	75,0	5,2~1,2	7,0	3,0
10^3ep	90,5~8,7	60,9~11,6	**72,5~25,9**	100,0	80,4	77,1	42,8	119,0	40,0	3,9~1,5	6,0	2,0

Tabela 8.39 – PTI / Liver

Nº It.	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de Camadas (CA)	Maior CA	Menor CA
PTI / PMR												
1ep	66,4~2,2	60,3~18,9	**1,0~0,0**	69,5	63,0	85,7	25,7	1,0	1,0	1,0~0,0	1,0	1,0
10ep	71,9~1,2	**66,4~8,9**	**1,0~0,0**	73,5	70,3	76,5	48,6	1,0	1,0	1,0~0,0	1,0	1,0
10^2ep	72,8~0,8	66,4~10,6	**1,0~0,0**	73,6	71,6	80,0	45,7	1,0	1,0	1,0~0,0	1,0	1,0
10^3ep	**73,3~0,9**	66,4~11,8	**1,0~0,0**	74,6	72,0	82,8	45,7	1,0	1,0	1,0~0,0	1,0	1,0
PTI / BCPMin												
1	66,7~1,3	62,9~16,1	**1,0~0,0**	68,4	64,8	91,2	41,2	1,0	1,0	1,0~0,0	1,0	1,0
10	74,2~3,1	63,8~9,8	25,2~38,9	80,0	69,7	79,4	52,9	84,0	1,0	1,3~0,5	2,0	1,0
10^2	**77,2~7,1**	**65,8~7,1**	34,7~43,9	93,5	70,6	80,0	55,9	95,0	1,0	1,5~0,7	3,0	1,0
10^3	76,4~3,7	60,6~10,9	37,9~47,8	86,5	74,2	77,1	40,0	102,0	1,0	1,5~0,7	3,0	1,0

Tabela 8.40 – OffTiling / Liver

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
OffTiling / PMR									
1ep	84,7~2,5	**60,8~5,9**	243,0~2,1	89,4	81,3	68,5	52,9	247,0	100,0
10ep	**100,0~0,0**	54,8~8,8	159,3~31,6	100,0	100,0	68,6	40,0	194,0	100,0
10^2ep	**100,0~0,0**	60,5~7,7	**82,2~5,5**	100,0	100,0	68,6	41,2	95,0	76,0
OffTiling / BCPMin									
1	72,3~1,1	62,7~11,5	45,6~4,8	74,3	70,7	76,5	40,0	53,0	38,0
10	76,2~2,4	**67,0~8,4**	41,2~2,7	79,0	70,7	79,4	54,3	45,0	35,0
10^2	78,8~1,3	61,2~6,8	**41,1~5,1**	80,4	77,2	71,4	48,6	49,0	34,0
10^3	**79,2~1,4**	66,4~7,1	42,4~3,4	82,0	77,1	76,5	55,9	49,0	38,0

Tabela 8.41 – Sequential / Liver

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Sequential / BCPMax									
1	46,4~7,5	45,9~26,3	108,0~6,9	62,3	38,4	85,7	5,7	119,0	98,0
10	46,4~6,2	**56,4~26,4**	93,1~4,8	57,7	40,0	85,7	5,7	101,0	87,0
10^2	**48,8~8,3**	49,4~26,7	**82,7~7,2**	65,8	42,1	85,3	5,7	95,0	72,0
Sequential / IncLp									
10ep	**42,0~2,9**	**42,0~25,7**	**19,0~2,9**	46,1	36,1	85,7	5,7	26,0	15,0

Os resultados de todos os algoritmos no domínio Liver não foram muito bons; o desempenho nos testes foi por volta de 66,0%. As precisões de treinamento e de teste do Sequential foram péssimas. Um dos motivos pelo qual o desempenho médio dos algoritmos não tenha sido bom talvez se deva à presença de ruídos no conjunto de treinamento.

8.2.5 O Domínio Pima

Este domínio, extraído do *UCI Repository* [Blake & Merz 1998], tem 768 instâncias de treinamento, cada uma delas definida por oito atributos numéricos. As instâncias são classificadas em duas classes, dependendo da paciente mostrar (ou não) sinais de diabetes de acordo com a *World Health Organization*.

Os resultados referentes aos algoritmos OffTiling/PMR e Sequential/IncLp não são apresentados para este domínio, uma vez que não foram obtidos experimentalmente devido ao excessivo tempo de processamento As Tabelas 8.42 a 8.51 mostram os resultados dos experimentos no domínio Pima.

Tabela 8.42 – PMR, BCPMin, MinOver e Thermal Modificado / Pima

Nº de Iterações	Precisão de Treinamento	Precisão de Teste	Maior Precisão de Treinamento	Menor Precisão de Treinamento	Maior Precisão de Teste	Menor Precisão de Teste
PMR						
1ep	67,1~1,0	65,4~5,4	69,2	65,7	75,3	58,4
10ep	69,1~0,6	68,4~3,7	69,9	68,3	74,0	63,6
10^2ep	70,7~0,7	68,2~6,6	71,9	69,5	79,2	58,4
10^3ep	**74,5~0,9**	**72,3~6,0**	75,7	73,0	80,5	59,7
BCPMin						
1	68,1~0,6	67,0~7,2	68,9	67,0	77,9	57,9
10	75,3~1,7	72,5~4,1	77,0	72,5	79,2	66,2
10^2	77,8~0,5	**75,8~5,7**	78,6	77,0	81,8	66,2
10^3	**78,5~0,6**	75,4~5,3	79,7	77,7	81,8	66,2
MinOver						
1ep	65,8~1,0	65,1~7,4	67,6	63,9	80,5	57,1
10ep	66,1~1,6	64,9~5,2	69,9	64,1	72,7	58,4
10^2ep	66,8~1,3	64,7~4,6	70,0	64,8	72,7	58,4
10^3ep	**70,1~0,9**	**69,3~6,1**	72,1	69,1	77,9	59,7
Themal Modificado						
1ep	49,7~15,9	52,2~17,5	65,6	33,9	81,8	28,6
10ep	53,7~15,5	47,3~17,4	66,1	35,2	64,9	18,2
10^2ep	**59,0~12,8**	59,4~14,6	66,1	34,4	81,8	32,5
10^3ep	**59,0~12,8**	**59,6~14,4**	65,8	33,9	81,8	28,6

Tabela 8.43 – Tower / Pima

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Tower / PMR									
1ep	67,8~0,9	66,5~7,2	2,0~0,9	69,2	66,7	81,8	59,7	3,0	1,0
10ep	69,6~0,8	67,4~5,3	**1,4~0,7**	70,9	68,0	76,6	59,7	3,0	1,0
10^2ep	71,5~0,8	67,6~4,7	2,7~1,1	73,2	70,6	75,0	58,4	5,0	2,0
10^3ep	**79,6~1,5**	**72,7~5,1**	12,9~4,4	81,8	76,7	80,5	66,2	19,0	5,0
Tower / BCPMin									
1	68,1~0,6	67,2~6,7	1,9~0,7	68,8	66,8	77,9	60,5	3,0	1,0
10	76,3~0,9	75,3~5,4	1,8~0,9	77,6	75,0	80,5	66,2	3,0	1,0
10^2	78,0~0,5	75,8~5,5	1,9~0,6	78,6	77,1	81,8	66,2	3,0	1,0
10^3	**78,5~0,8**	**76,1~6,5**	**1,4~0,5**	80,0	77,6	84,4	65,6	2,0	1,0

Tabela 8.44 – Pyramid / Pima

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Pyramid / PMR									
1ep	67,7~0,8	67,3~6,7	**1,6~0,8**	68,7	66,2	81,8	57,1	3,0	1,0
10ep	69,3~0,9	67,3~5,3	1,7~1,1	70,8	67,7	75,3	58,4	4,0	1,0
10^2ep	71,7~0,8	68,6~3,5	3,5~1,7	73,1	62,3	75,3	62,3	6,0	1,0
10^3ep	**78,7~1,9**	**73,4~4,9**	10,4~3,8	81,5	75,1	83,1	68,4	15,0	3,0
Pyramid / BCPMin									
1	68,1~0,7	67,3~6,8	1,7~0,5	68,9	66,8	77,9	60,5	2,0	1,0
10	75,0~2,2	74,1~6,1	1,8~0,8	76,8	69,2	81,8	62,3	3,0	1,0
10^2	77,7~0,9	75,9~4,2	**1,4~0,5**	78,7	76,3	81,8	68,8	2,0	1,0
10^3	**78,2~1,1**	**76,3~6,0**	1,5~0,7	80,0	75,8	84,4	66,2	3,0	1,0

Tabela 8.45 – Upstart / Pima

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Upstart / PMR									
1ep	67,7~0,7	65,5~5,1	**46,2~22,8**	68,5	66,1	74,0	58,4	61,0	2,0
10ep	68,9~0,7	66,1~4,9	56,9~2,4	70,3	67,9	72,7	58,4	61,0	53,0
10^2ep	**70,8~0,8**	**68,5~5,4**	100,0~0,0	71,9	69,3	75,3	58,4	100,0	100,0
Upstart / BCPMin									
1	68,1~0,7	67,0~7,4	**100,0~0,0**	68,9	66,7	77,9	56,6	101,0	100,0
10	74,1~2,0	73,0~6,2	**100,0~0,0**	77,1	70,6	80,5	58,4	101,0	100,0
10^2	**77,7~0,6**	**75,8~4,4**	**100,0~0,0**	78,6	76,9	81,8	68,8	101,0	100,0

Tabela 8.46 – Shift / Pima

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Shift / PMR									
1ep	69,1~1,0	67,4~5,3	**4,1~1,8**	70,0	66,9	77,9	56,2	7,0	2,0
10ep	71,6~1,5	67,5~4,1	6,2~1,0	73,8	70,2	72,2	61,0	9,0	4,0
10^2ep	75,3~1,6	68,6~5,0	10,9~4,1	77,4	73,2	79,2	61,0	19,0	5,0
10^3ep	**80,0~1,3**	**73,2~5,4**	12,2~2,9	82,1	77,6	81,8	64,9	18,0	8,0
Shift / BCPMin									
1	72,9~1,3	70,0~4,8	**2,0~0,0**	75,5	71,5	79,2	63,2	2,0	2,0
10	75,4~1,1	72,1~5,6	2,3~0,7	77,0	73,8	79,2	63,2	3,0	1,0
10^2	78,1~0,7	**76,6~5,6**	2,0~0,7	79,2	76,8	83,1	63,6	3,0	1,0
10^3	**78,8~0,7**	75,4~5,7	2,0~0,8	79,9	77,9	81,8	66,2	3,0	1,0

Tabela 8.47 – Perceptron Cascade / Pima

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Perceptron Cascade / PMR									
1ep	67,8~0,5	66,9~6,0	**1,7~0,8**	68,6	67,0	80,5	59,7	3,0	1,0
10ep	69,7~0,8	69,8~6,2	**1,7~0,8**	70,7	68,0	81,8	61,0	3,0	1,0
10^2ep	72,3~1,5	70,1~6,4	2,8~1,9	76,0	70,8	80,5	58,4	7,0	1,0
10^3ep	**78,4~1,3**	**75,0~6,7**	3,7~1,5	80,9	76,7	84,4	63,6	7,0	2,0
Perceptron Cascade / BCPMin									
1	68,1~0,6	67,3~6,7	2,1~0,7	69,0	67,0	77,9	60,5	3,0	1,0
10	76,0~1,0	74,5~4,8	2,1~1,4	77,3	74,2	80,5	66,2	5,0	1,0
10^2	77,8~0,6	75,4~5,3	1,7~0,9	78,7	77,1	81,8	67,5	4,0	1,0
10^3	**78,5~0,7**	**75,5~5,5**	**1,3~0,5**	79,9	77,6	81,8	66,2	2,0	1,0

Tabela 8.48 – Tiling / Pima

Nº It.	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de (CA)	Maior CA	Menor CA
Tiling / PMR												
1ep	67,9~1,7	**67,7~5,3**	**1,0~0,0**	70,5	65,1	77,9	57,1	1,0	1,0	1,0~0,0	1,0	1,0
10ep	69,2~1,3	66,7~6,8	**1,0~0,0**	71,2	66,4	80,5	58,4	1,0	1,0	1,0~0,0	1,0	1,0
10^2ep	70,7~0,9	67,3~7,2	**1,0~0,0**	71,9	69,3	81,8	54,5	1,0	1,0	1,0~0,0	1,0	1,0
10^3ep	**90,5~11,5**	66,5~7,1	208,0~151,9	100,0	74,7	79,2	57,1	369,0	1,0	8,2~5,5	14,0	1,0
Tiling / BCPMin												
1	76,0~1,1	74,2~5,8	164,9~73,8	77,4	74,3	83,1	64,9	358,0	100,0	3,8~1,2	7,0	3,0
10	80,4~5,7	**74,6~8,3**	**152,5~134,8**	91,8	76,8	84,4	59,7	407,0	1,0	4,0~3,1	10,0	1,0
10^2	88,7~9,0	71,9~4,9	198,2~91,1	100,0	78,4	77,9	63,2	312,0	66,0	5,7~2,6	9,0	2,0
10^3	**91,1~8,6**	70,7~7,6	229,4~130,3	100,0	78,7	81,8	55,8	410,0	1,0	6,8~3,7	12,0	1,0
Tiling Híbrido												
1ep	76,3~1,2	**74,6~6,1**	151,1~73,4	78,7	74,2	81,8	63,6	330,0	71,0	3,5~1,4	7,0	2,0
10ep	79,4~2,0	74,5~6,4	**134,7~55,9**	82,6	75,9	83,1	64,9	213,0	59,0	3,4~1,1	5,0	2,0
10^2ep	88,9~9,2	70,0~7,6	180,6~107,5	100,0	79,3	83,1	61,0	316,0	60,0	5,7~3,5	10,0	2,0
10^3ep	**91,5~8,0**	66,9~5,6	253,4~90,2	100,0	81,2	75,3	57,8	361,0	100,0	7,4~2,9	11,0	3,0

Tabela 8.49 – PTI / Pima

Nº It.	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de Camadas (CA)	Maior CA	Menor CA
PTI / PMR												
1ep	67,8~0,8	66,3~7,6	**1,0~0,0**	68,6	66,3	76,6	53,2	1,0	1,0	1,0~0,0	1,0	1,0
10ep	69,2~0,7	68,2~5,3	**1,0~0,0**	70,1	67,9	76,6	58,4	1,0	1,0	1,0~0,0	1,0	1,0
10^2ep	70,6~0,8	69,4~6,2	**1,0~0,0**	71,9	69,2	77,6	57,2	1,0	1,0	1,0~0,0	1,0	1,0
10^3ep	**74,3~0,7**	**73,2~4,3**	**1,0~0,0**	75,8	73,7	81,8	67,5	1,0	1,0	1,0~0,0	1,0	1,0
PTI / BCPMin												
1	68,0~0,6	66,9~6,4	**1,0~0,0**	68,9	67,0	77,9	57,9	1,0	1,0	1,0~0,0	1,0	1,0
10	74,3~2,1	74,7~4,5	**1,0~0,0**	76,8	70,8	81,8	66,2	1,0	1,0	1,0~0,0	1,0	1,0
10^2	77,8~0,5	75,9~5,4	**1,0~0,0**	78,5	77,2	81,8	63,6	1,0	1,0	1,0~0,0	1,0	1,0
10^3	**78,5~0,7**	**76,7~5,1**	**1,0~0,0**	80,0	77,7	84,4	68,8	1,0	1,0	1,0~0,0	1,0	1,0

Tabela 8.50 – OffTiling / Pima

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
OffTiling / BCPMin									
1ep	74,7~1,0	73,6~4,7	75,0~5,5	75,9	73,5	79,2	64,9	85,0	63,0
10ep	77,2~0,9	72,3~8,2	67,2~8,9	78,5	75,3	83,1	57,1	81,0	58,0
10^2ep	79,6~1,2	73,4~6,3	**64,3~4,4**	81,5	78,0	81,8	63,6	73,0	56,0
10^3ep	**80,1~1,2**	**74,5~6,7**	65,8~4,4	81,5	78,0	83,1	63,6	73,0	61,0

Tabela 8.51 – Sequential / Pima

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Sequential / BCPMax									
1	38,7~9,5	38,8~12,8	230,0~13,0	65,3	34,1	64,9	18,2	244,0	100,0
10	45,6~10,7	45,5~13,9	190,5~5,5	65,7	35,9	68,4	19,9	199,0	100,0
10^2	**49,9~13,6**	**52,1~16,8**	**167,4~10,5**	66,5	36,0	81,8	29,9	186,0	100,0

Como pode ser observado nas tabelas, os algoritmos que investem na construção de cada uma as camadas intermediárias (a saber, o Tiling, o PTI, e o OffTiling), bem como o Shift/BCPMin e o Perceptron Cascade/BCPMin tiveram os melhores resultados. Se considerado o número de neurônios, entretanto, os melhores resultados foram obtidos pelos algoritmos Shift, Perceptron Cascade, PTI (todos na versão BCPMin) e pelo próprio BCPMin.

Apesar do PTI/BCPMin ter obtido o melhor desempenho nos conjuntos de testes (ver Tabela 4.49), é importante comentar que o PTI não conseguiu tornar a primeira camada confiável e, consequentemente, o PTI/BCPMin é equivalente ao BCPMin.

Os resultados obtidos com o Sequential, tanto no treinamento quanto no teste, ficaram novamente abaixo da média, com relação aos demais algoritmos. O segundo melhor resultado, obtido pelo Shift/BCPMin foi devido não apenas ao uso do BCPMin no treinamento das TLUs mas, também, pela estratégia de criação da rede implementada pelo Shift.

8.2.6 O Domínio Breast Cancer

O dominio Breast Câncer (WPBC), extraído do *UCI Repository* [Blake & Merz 1998], possui 198 instancias, sendo que, cada uma delas é um registro de um caso de câncer de mama. As instâncias são formadas por 33 atributos relativos a cada paciente, a classe é positiva se o câncer voltar a ocorrer e negativa caso contrário. As Tabelas 8.52 a 8.61 mostram os resultados dos experimentos no domínio Breast Cancer.

Tabela 8.52 – PMR, BCPMin, MinOver e Thermal Modificado / Breast Cancer

Nº de Iterações	Precisão de Treinamento	Precisão de Teste	Maior Precisão de Treinamento	Menor Precisão de Treinamento	Maior Precisão de Teste	Menor Precisão de Teste
PMR						
1ep	77,1~1,5	**77,4~11,3**	79,2	74,8	94,7	60,0
10ep	79,2~1,4	75,4~10,8	81,5	77,1	89,5	65,0
10^2ep	82,9~1,2	74,7~14,6	84,3	81,5	95,0	47,4
10^3ep	**83,8~1,2**	74,3~13,6	85,4	82,0	94,7	60,0
BCPMin						
1	78,3~1,4	76,9~12,6	80,3	76,5	94,7	60,0
10	**78,4~1,3**	76,9~12,0	80,3	76,5	94,7	60,0
10^2	**78,4~1,3**	76,9~12,0	80,3	76,5	94,7	60,0
10^3	78,3~1,4	**77,9~12,6**	80,3	76,5	94,7	60,0
MinOver						
1ep	76,7~1,3	76,9~12,2	78,7	74,8	94,7	60,0
10ep	79,7~1,3	76,8~10,3	82,0	78,1	95,0	60,0
10^2ep	80,9~2,2	76,3~10,5	84,8	78,1	89,5	60,0
10^3ep	**82,8~2,1**	**79,9~11,3**	85,9	79,8	95,0	65,0
Themal Modificado						
1ep	**76,8~1,3**	**76,9~12,2**	78,7	74,9	94,7	60,0
10ep	**76,8~1,3**	**76,9~12,2**	78,7	74,9	94,7	60,0
10^2ep	**76,8~1,3**	**76,9~12,2**	78,7	74,9	94,7	60,0
10^3ep	**76,8~1,3**	**76,9~12,2**	78,7	74,9	94,7	60,0

Tabela 8.53 – Tower / Breast Cancer

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Tower / PMR									
1ep	76,8~1,4	**77,4~11,5**	1,9~0,3	79,2	75,3	94,7	60,0	2,0	1,0
10ep	80,2~0,6	76,9~12,9	2,2~0,6	81,5	79,2	95,0	65,0	3,0	1,0
10^2ep	83,1~0,9	76,8~11,5	2,1~0,6	84,3	81,6	95,0	63,2	3,0	1,0
10^3ep	**84,0~1,1**	75,8~13,1	**1,8~0,4**	85,4	82,0	94,7	60,0	2,0	1,0
Tower / BCPMin									
1	78,3~1,4	76,9~12,6	2,0~0,0	80,3	76,5	94,7	60,0	2,0	2,0
10	78,4~1,4	**76,9~12,0**	2,1~0,3	80,3	76,5	94,7	60,0	3,0	2,0
10^2	78,4~1,4	76,4~12,4	2,0~0,0	80,3	76,5	94,7	60,0	2,0	2,0
10^3	**78,6~1,4**	76,4~12,4	**1,9~0,3**	80,3	76,5	94,7	60,0	2,0	1,0

Tabela 8.54 – Pyramid / Breast Cancer

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Pyramid / PMR									
1ep	77,2~1,3	73,9~11,8	1,8~0,4	78,7	74,9	94,7	60,0	2,0	1,0
10ep	79,3~0,9	**76,9~11,7**	**1,5~0,5**	80,9	78,1	94,7	60,0	2,0	1,0
10^2ep	83,2~1,3	76,8~10,2	2,1~1,0	84,8	81,5	90,0	63,2	4,0	1,0
10^3ep	**84,0~1,1**	76,3~11,4	1,9~0,3	85,4	82,0	94,7	65,0	2,0	1,0
Pyramid / BCPMin									
1	**78,4~1,4**	**77,4~12,2**	**2,0~0,0**	80,3	76,5	94,7	60,0	2,0	2,0
10	78,3~1,5	76,9~12,0	**2,0~0,0**	80,3	76,4	94,7	60,0	2,0	2,0
10^2	78,3~1,5	76,4~12,4	2,1~0,3	80,3	76,4	94,7	60,0	3,0	2,0
10^3	78,3~1,5	76,4~12,4	**2,0~0,0**	80,3	76,4	94,7	60,0	2,0	2,0

Tabela 8.55 – Upstart / Breast Cancer

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Upstart / PMR									
1ep	76,9~1,4	75,4~12,0	**7,7~16,9**	78,7	74,9	94,7	60,0	54,0	2,0
10ep	79,1~0,8	74,9~12,1	23,3~27,5	80,3	78,1	90,0	60,0	58,0	2,0
10^2ep	**82,6~1,0**	**76,3~10,5**	34,1~27,7	83,7	81,0	90,0	60,0	57,0	2,0
Upstart / BCPMin									
1	78,3~1,4	76,9~12,6	**100,0~0,0**	80,3	76,5	94,7	60,0	100,0	100,0
10	**78,5~1,3**	76,9~12,0	**100,0~0,0**	80,3	76,5	94,7	60,0	100,0	100,0
10^2	78,3~1,4	**77,9~12,6**	**100,0~0,0**	80,3	76,5	94,7	60,0	100,0	100,0

Tabela 8.56 – Shift / Breast Cancer

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Shift / PMR									
1ep	77,0~1,1	**75,4~11,0**	**1,1~0,3**	78,7	75,8	89,5	75,8	2,0	1,0
10ep	80,2~1,3	73,8~11,1	1,6~0,7	82,6	78,2	90,0	60,0	3,0	1,0
10^2ep	84,2~1,0	74,8~13,0	2,9~1,4	85,4	82,0	95,0	63,2	5,0	1,0
10^3ep	**84,7~1,3**	72,3~12,6	1,9~0,7	86,5	82,6	90,0	60,0	3,0	1,0
Shift / BCPMin									
1	78,5~1,3	75,4~11,6	1,1~0,3	80,3	76,5	94,7	60,0	2,0	1,0
10	78,4~1,3	**77,9~12,6**	**1,0~0,0**	80,3	76,5	94,7	60,0	1,0	1,0
10^2	78,5~1,3	76,9~12,0	1,1~0,3	80,3	77,1	94,7	60,0	2,0	1,0
10^3	**78,6~1,5**	76,9~11,7	1,1~0,3	81,5	76,5	94,7	60,0	2,0	1,0

Tabela 8.57 – Perceptron Cascade / Breast Cancer

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Perceptron Cascade / PMR									
1ep	76,9~1,3	**75,4~11,3**	1,8~0,4	78,7	74,9	94,7	60,0	2,0	1,0
10ep	79,5~0,7	75,4~12,0	1,6~0,8	90,0	78,1	90,0	60,0	3,0	1,0
10^2ep	83,2~1,1	75,2~11,9	**1,5~0,7**	84,8	81,5	90,0	57,9	3,0	1,0
10^3ep	**83,8~1,0**	74,8~13,0	1,9~0,6	84,8	82,0	94,7	60,0	3,0	1,0
Perceptron Cascade / BCPMin									
1	78,3~1,4	76,9~12,6	2,0~0,0	80,3	76,5	94,7	60,0	2,0	2,0
10	**78,5~1,3**	76,4~12,4	**1,9~0,3**	80,3	76,5	94,7	60,0	2,0	1,0
10^2	**78,5~1,3**	**76,9~12,0**	2,0~0,0	80,3	76,5	94,7	60,0	2,0	2,0
10^3	78,5~1,4	76,4~12,4	2,0~0,0	80,3	76,5	94,7	60,0	2,0	2,0

Tabela 8.58 – Tiling / Breast Cancer

Nº It.	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de Camadas (CA)	Maior CA	Menor CA
Tiling / PMR												
1ep	76,8~1,4	**77,4~11,5**	**1,0~0,0**	79,2	74,5	94,7	60,0	1,0	1,0	1,0~0,0	1,0	1,0
10ep	79,2~1,0	74,3~10,8	**1,0~0,0**	80,4	77,5	90,0	65,0	1,0	1,0	1,0~0,0	1,0	1,0
10^2ep	82,8~1,0	76,8~10,3	**1,0~0,0**	84,3	81,5	90,0	60,0	1,0	1,0	1,0~0,0	1,0	1,0
10^3ep	**85,6~5,3**	73,3~15,1	7,1~13,6	100,0	82,0	94,7	50,0	41,0	1,0	1,4~0,9	4,0	1,0
Tiling / BCPMin												
1	82,7~6,7	70,7~12,7	38,6~23,2	100,0	77,5	85,0	50,0	7,0	1,0	2,2~0,9	4,0	1,0
10	89,7~9,2	**71,3~13,4**	36,8~14,0	100,0	77,5	90,0	50,0	51,0	1,0	2,7~1,2	4,0	1,0
10^2	**100,0~0,0**	70,2~12,8	40,9~4,2	100,0	100,0	90,0	52,6	50,0	36,0	3,4~0,5	4,0	3,0
10^3	95,7~9,1	69,3~9,9	**36,5~13,2**	100,0	77,5	85,0	55,0	48,0	1,0	3,1~1,0	4,0	1,0
Tiling Híbrido												
1ep	82,4~6,7	**72,2~9,8**	38,0~22,4	100,0	77,5	85,0	60,0	63,0	1,0	2,3~1,2	5,0	1,0
10ep	89,0~10,5	68,3~13,8	**32,4~17,9**	100,0	77,5	85,0	55,0	41,0	1,0	2,7~1,4	5,0	1,0
10^2ep	**100,0~0,0**	67,3~8,9	40,3~3,6	100,0	100,0	80,0	55,0	48,0	35,0	3,8~0,8	5,0	3,0
10^3ep	97,9~6,7	68,2~9,6	33,0~12,3	100,0	78,7	80,0	50,0	45,0	1,0	3,2~0,9	4,0	1,0

Tabela 8.59 – PTI / Breast Cancer

Nº It.	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de (CA)	Maior CA	Menor CA
PTI / PMR												
1ep	77,0~1,1	74,4~11,5	**1,0~0,0**	78,7	75,8	94,7	60,0	1,0	1,0	1,0~0,0	1,0	1,0
10ep	78,7~1,3	75,4~11,0	**1,0~0,0**	81,5	77,1	89,5	60,0	1,0	1,0	1,0~0,0	1,0	1,0
10^2ep	82,9~0,9	75,8~9,9	**1,0~0,0**	84,3	81,5	90,0	650	1,0	1,0	1,0~0,0	1,0	1,0
10^3ep	**83,7~1,2**	**78,8~10,7**	**1,0~0,0**	88,4	82,0	95,0	65,0	1,0	1,0	1,0~0,0	1,0	1,0
PTI / BCPMin												
1	78,4~1,4	**77,4~12,2**	**1,0~0,0**	80,3	76,5	94,7	60,0	1,0	1,0	1,0~0,0	1,0	1,0
10	81,7~7,0	75,9~14,9	25,8~33,4	100,0	76,5	94,7	45,0	87,0	1,0	1,7~1,3	5,0	1,0
10^2	82,7~9,2	74,9~11,9	15,3~30,6	100,0	76,5	94,7	60,0	83,0	1,0	1,6~1,3	4,0	1,0
10^3	**83,5~8,8**	72,9~14,4	18,6~28,4	100,0	76,5	94,7	50,0	63,0	1,0	1,5~0,8	3,0	1,0

Tabela 8.60 – OffTiling / Breast Cancer

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
OffTiling / PMR									
1ep	90,5~2,8	68,9~10,8	240,4~1,9	94,4	86,5	85,0	55,0	224,0	110,0
10ep	**100,0~0,0**	**69,2~9,8**	60,6~7,8	100,0	100,0	80,0	55,0	74,0	49,0
10^2ep	**100,0~0,0**	65,2~3,1	**41,3~3,1**	100,0	100,0	80,0	40,0	45,0	37,0
OffTiling / BCPMin									
1	79,0~1,4	**77,9~12,2**	41,8~2,9	80,9	77,1	94,7	55,0	47,0	38,0
10	81,2~3,1	74,8~11,4	34,5~3,8	86,1	75,3	85,0	50,0	42,0	27,0
10^2	**87,8~4,6**	70,3~9,4	33,7~2,6	94,9	79,8	85,0	55,0	39,0	30,0
10^3	84,2~6,0	67,2~14,5	**31,8~3,1**	91,1	75,4	90,0	40,0	37,0	26,0

Tabela 8.61 – Sequential / Breast Cancer

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Sequential / BCPMax									
1	**27,9~1,2**	**27,6~11,5**	60,6~2,9	29,6	26,4	40,0	10,5	66,0	57,0
10	**27,9~1,2**	**27,6~11,5**	50,3~11,4	29,6	26,4	40,0	10,5	56,0	45,0
10^2	**27,9~1,2**	**27,6~11,5**	**46,8~2,8**	29,6	26,4	40,0	10,5	50,0	41,0
Sequential / IncLp									
10ep	**76,8~1,3**	**76,9~12,2**	**4,0~0,5**	78,7	74,9	94,7	60,0	5,0	3,0

Neste domínio, com exceção do Sequential, todos os algoritmos tiveram resultados similares, com relação à precisão de testes. O algoritmo que obteve o melhor resultado foi o PTI/PMR. O algoritmo OffTiling/PMR obteve os melhores resultados no treinamento, porém seus resultados nos testes ficaram abaixo da média, possivelmente pelo fato do algoritmo criar redes muito grandes. Novamente, como já esperado os resultados obtidos pelo Sequential o tornam um candidato a nunca ser usado.

8.2.7 Os Domínios Monks

Os domínios Monks1, Monks2 e Monks3 são domínios artificiais e foram extraídos do *UCI Repository* [Blake & Merz 1998]. Cada domínio possui 432 instâncias descritas por 6 atributos e distribuídas em duas classes diferentes. As instâncias que compõem os três domínios são exatamente iguais, salvo pela classe associada a cada uma delas. Os domínios foram construídos da seguinte maneira:

- O primeiro, o segundo e o quarto atributo podem assumir os valores 1, 2 ou 3.
- O terceiro e o sexto atributo podem assumir valor 1 ou 2.
- O quinto atributo pode assumir os valores 1, 2, 3 e 4.

A partir da construção das instâncias de treinamento, cada versão (Monks1, Monks2 ou Monks3) estabelece uma regra para a determinação das classes. Às instâncias que satisfazem a regra são atribuídas a classe positiva; às que não satisfazem a classe negativa.

- No domínio Monks1, se $A1 = A2$ ou $A5 = 1$ então a classe é positiva.
- No domínio Monks2, a classe será positiva se, em uma instância, exatamente dois atributos possuírem valor 1.
- No domínio Monks3 a classe será positiva quando $A5 = 3$ e $A4 = 1$ ou quando $A5 \neq 4$ e $A2 \neq 3$.

8.2.7.1 O Domínio Monks1

As Tabelas 8.62 a 8.71 mostram os resultados dos experimentos no domínio Monks1.

Tabela 8.62 – PMR, BCPMin, MinOver e Thermal Modificado / Monks1

Nº de Iterações	Precisão de Treinamento	Precisão de Teste	Maior Precisão de Treinamento	Menor Precisão de Treinamento	Maior Precisão de Teste	Menor Precisão de Teste
PMR						
1ep	72,3~3,3	66,7~26,5	77,6	68,1	90,9	25,0
10ep	74,7~3,4	**68,2~37,8**	80,9	71,0	100,0	0,0
10^2ep	76,3~3,9	67,7~37,3	83,8	73,0	100,0	0,0
10^3ep	**77,2~3,4**	62,1~33,7	83,4	74,3	95,3	0,0
BCPMin						
1	76,8~3,8	68,4~27,2	83,8	73,2	100,0	23,3
10	76,7~3,9	68,6~29,2	83,8	73,0	100,0	23,3
10^2	77,3~3,7	**68,8~27,8**	83,8	74,6	100,0	23,3
10^3	**78,3~3,0**	46,8~28,1	83,8	75,3	100,0	0,0
MinOver						
1ep	66,2~8,2	49,4~25,3	76,3	52,0	86,0	11,6
10ep	70,1~7,6	**59,6~30,5**	80,4	52,1	100,0	23,3
10^2ep	69,1~2,9	54,2~29,1	73,5	63,8	90,9	6,8
10^3ep	**71,2~2,9**	57,2~33,2	77,3	67,9	100,0	4,5
Themal Modificado						
1ep	**50,8~3,4**	**42,9~30,3**	52,9	44,3	100,0	23,3
10ep	50,8~3,8	**42,9~30,3**	52,9	44,3	100,0	23,3
10^2ep	52,6~2,1	26,4~18,8	55,7	48,1	67,4	0,0
10^3ep	52,2~2,5	29,9~22,8	55,7	48,1	67,4	0,0

Tabela 8.63 – Tower / Monks1

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Tower / PMR									
1ep	77,0~4,7	64,4~33,8	12,5~7,8	83,8	70,2	97,7	4,5	20,0	1,0
10ep	82,9~4,5	66,1~30,0	8,3~3,7	90,5	77,1	97,7	9,0	14,0	1,0
10^2ep	90,2~2,3	65,1~28,1	9,1~3,6	95,4	87,4	93,1	23,3	15,0	4,0
10^3ep	**91,3~2,1**	**72,8~26,4**	**6,3~2,1**	94,6	88,2	100,0	34,9	11,0	4,0
Tower / BCPMin									
1	76,9~3,8	**70,2~28,1**	**2,6~0,8**	83,8	73,0	100,0	23,3	4,0	2,0
10	81,8~5,6	65,6~34,1	4,6~1,9	90,5	73,2	100,0	0,0	8,0	2,0
10^2	87,0~5,2	65,8~22,0	5,7~2,8	91,8	74,8	100,0	34,9	10,0	2,0
10^3	**90,3~5,7**	65,1~32,2	4,3~1,2	100,0	81,7	100,0	0,0	7,0	3,0

Tabela 8.64 – Pyramid / Monks1

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Pyramid / PMR									
1ep	74,7~5,1	67,2~31,0	**3,9~2,3**	85,1	70,6	100,0	4,5	10,0	2,0
10ep	85,7~6,2	63,9~34,1	10,0~4,0	98,2	78,9	100,0	0,0	17,0	4,0
10^2ep	92,8~4,0	73,6~25,6	7,4~2,8	100,0	89,5	100,0	25,6	12,0	4,0
10^3ep	**93,1~3,0**	**82,8~17,9**	5,5~1,6	100,0	90,7	100,0	51,2	9,0	4,0
Pyramid / BCPMin									
1	76,6~3,9	69,7~27,6	2,2~0,8	83,8	73,0	97,7	23,3	4,0	1,0
10	76,9~3,4	**70,0~28,3**	2,2~0,8	83,8	73,0	100,0	23,3	4,0	1,0
10^2	76,7~4,6	56,2~30,5	**1,6~0,5**	83,8	68,3	100,0	23,3	2,0	1,0
10^3	**77,7~3,8**	56,9~30,6	1,8~0,6	83,8	72,8	100,0	28,3	3,0	1,0

Tabela 8.65 – Upstart / Monks1

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Upstart / PMR									
1ep	74,9~4,9	**66,7~31,0**	51,7~27,9	83,4	68,9	97,7	13,9	87,0	2,0
10ep	84,0~6,2	57,5~26,3	**42,0~27,9**	90,7	70,9	95,3	11,4	62,0	1,0
10^2ep	**86,3~9,2**	61,4~32,8	61,4~51,1	100,0	73,2	100,0	0,0	100,0	2,0
Upstart / BCPMin									
1	77,3~3,7	**66,0~27,8**	**100,0~0,0**	83,8	74,0	100,0	23,5	100,0	98,0
10	89,9~7,9	59,8~7,9	**100,0~0,0**	100,0	78,7	97,7	0,0	100,0	100,0
10^2	**94,3~3,7**	57,7~25,5	**100,0~0,0**	100,0	88,4	100,0	25,0	100,0	82,0

Tabela 8.66 – Shift / Monks1

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Shift / PMR									
1ep	76,9~6,5	59,4~30,4	**3,2~1,8**	86,1	69,2	95,3	4,5	6,0	1,0
10ep	84,4~6,7	68,1~32,1	4,6~1,8	99,2	76,6	100,0	22,7	8,0	3,0
10^2ep	88,9~2,7	66,7~30,5	5,2~3,3	94,3	85,3	100,0	13,9	14,0	3,0
10^3ep	**93,3~4,0**	**74,5~26,3**	4,2~0,6	99,5	88,9	100,0	34,9	5,0	3,0
Shift / BCPMin									
1	79,5~6,1	**71,4~29,8**	1,9~1,0	90,0	73,0	100,0	23,3	4,0	1,0
10	80,6~5,7	63,5~27,4	2,1~0,7	90,0	73,2	100,0	23,3	3,0	1,0
10^2	81,6~5,6	68,3~28,6	**1,7~0,5**	90,0	73,2	100,0	23,3	2,0	1,0
10^3	**85,3~6,8**	64,9~27,6	2,1~0,3	92,8	77,1	100,0	23,3	3,0	2,0

Tabela 8.67 – Perceptron Cascade / Monks1

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Perceptron Cascade / PMR									
1ep	73,5~3,1	68,4~30,2	**1,9~0,7**	78,9	69,7	93,0	13,6	3,0	1,0
10ep	79,4~5,5	66,6~28,6	2,9~1,5	90,0	72,2	100,0	13,6	5,0	1,0
10^2ep	88,9~3,8	70,4~32,0	4,4~1,6	91,0	78,4	100,0	23,3	7,0	2,0
10^3ep	**91,0~2,3**	**70,8~29,1**	4,0~0,9	94,9	87,4	100,0	34,9	6,0	3,0
Perceptron Cascade / BCPMin									
1	76,9~3,9	69,0~28,8	**2,1~0,6**	83,8	73,5	100,0	23,3	3,0	1,0
10	84,3~9,5	69,1~34,3	3,0~1,8	100,0	73,2	100,0	4,5	6,0	1,0
10^2	90,0~6,9	61,7~32,0	3,4~1,2	100,0	78,4	100,0	9,1	5,0	1,0
10^3	**91,6~6,1**	**69,3~25,6**	4,0~1,3	100,0	80,7	100,0	27,9	7,0	3,0

Tabela 8.68 – Tiling / Monks1

Nº It.	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de Camadas (CA)	Maior CA	Menor CA
Tiling / PMR												
1ep	71,7~3,8	64,4~25,6	**1,0~0,0**	79,6	68,4	100,0	25,0	1,0	1,0	1,0~0,0	1,0	1,0
10ep	83,4~12,6	71,9~35,9	13,9~20,1	100,0	71,9	100,0	2,3	61,0	1,0	1,8~1,0	4,0	1,0
10^2ep	91,4~12,2	**81,5~16,8**	9,8~8,0	100,0	73,0	100,0	53,5	27,0	1,0	2,5~1,1	4,0	1,0
10^3ep	**92,4~10,1**	61,6~40,5	9,6~8,7	100,0	74,6	97,7	0,0	28,0	1,0	2,1~1,0	4,0	1,0
Tiling / BCPMin												
1	85,1~11,4	**84,2~24,3**	10,2~9,9	100,0	73,0	100,0	23,3	23,0	1,0	2,1~1,2	4,0	1,0
10	98,2~5,7	83,6~21,7	10,1~2,4	100,0	82,0	100,0	44,2	13,0	7,0	2,7~0,7	4,0	2,0
10^2	98,4~5,0	81,3~21,5	15,4~6,7	100,0	84,3	100,0	44,2	25,0	7,0	3,0~0,8	4,0	2,0
10^3	**100,0~0,0**	81,0~20,4	**8,4~2,2**	100,0	100,0	100,0	46,5	12,0	6,0	2,6~0,5	3,0	2,0
Tiling Híbrido												
1ep	88,5~12,6	69,6~27,1	11,8~10,7	100,0	72,3	100,0	27,3	32,0	1,0	2,6~1,6	5,0	1,0
10ep	92,9~11,5	69,1~26,9	**8,1~6,4**	100,0	74,8	100,0	30,2	19,0	1,0	2,3~1,1	4,0	1,0
10^2ep	97,9~6,4	71,8~3,4	8,5~3,5	100,0	79,7	100,0	30,2	16,0	5,0	2,7~0,5	3,0	2,0
10^3ep	**98,0~6,3**	**78,8~21,9**	9,5~5,1	100,0	79,9	100,0	41,9	16,0	1,0	2,7~0,8	4,0	1,0

Tabela 8.69 – PTI / Monks1

Nº It.	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de Camadas (CA)	Maior CA	Menor CA
PTI / PMR												
1ep	71,5~3,9	68,4~32,6	**1,0~0,0**	77,8	67,8	100,0	0,0	1,0	1,0	1,0~0,0	1,0	1,0
10ep	74,9~3,5	**70,7~34,1**	**1,0~0,0**	81,2	71,9	100,0	4,5	1,0	1,0	1,0~0,0	1,0	1,0
10^2ep	**81,2~10,4**	66,1~36,5	2,6~3,4	100,0	72,8	100,0	0,0	9,0	1,0	1,4~0,8	3,0	1,0
10^3ep	79,6~7,9	68,6~36,7	1,5~1,6	100,0	73,8	100,0	0,0	6,0	1,0	1,1~0,3	2,0	1,0
PTI / BCPMin												
1	79,4~8,1	70,8~28,7	**1,4~1,3**	100,0	73,0	100,0	23,4	5,0	1,0	1,1~0,3	2,0	1,0
10	**95,9~8,9**	73,3~28,3	15,5~12,7	100,0	75,3	100,0	27,7	37,0	1,0	2,4~0,9	4,0	1,0
10^2	93,8~10,1	**79,3~24,9**	16,2~18,4	100,0	76,9	100,0	23,5	57,0	1,0	2,1~0,7	3,0	1,0
10^3	93,1~11,2	62,5~31,0	19,9~22,2	100,0	75,1	100,0	27,9	68,0	1,0	2,0~0,9	4,0	1,0

Tabela 8.70 – OffTiling / Monks1

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
OffTiling / PMR									
1ep	84,1~8,5	67,8~17,5	206,0~18,1	100,0	73,3	95,3	40,9	215,0	100,0
10ep	99,9~0,4	65,9~22,2	63,8~70,4	100,0	98,7	100,0	27,9	214,0	11,0
10^2ep	**100,0~0,0**	**73,9~21,6**	**14,6~8,2**	100,0	100,0	100,0	40,9	34,0	7,0
OffTiling / BCPMin									
1	75,3~5,8	69,9~27,8	14,1~2,4	88,2	68,6	100,0	29,5	19,0	10,0
10	83,4~9,7	65,9~23,7	12,5~3,0	100,0	72,2	100,0	25,6	16,0	8,0
10^2	87,0~12,2	80,4~21,3	10,9~3,0	100,0	68,9	100,0	43,2	17,0	6,0
10^3	**91,5~8,2**	**82,4~18,1**	**10,7~3,5**	100,0	79,2	100,0	53,5	17,0	6,0

Tabela 8.71 – Sequential / Monks1

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Sequential / BCPMax									
1	**76,2~4,3**	**74,8~30,7**	**22,6~14,1**	83,8	72,2	100,0	23,3	46,0	4,0
10	76,1~4,4	74,6~31,2	26,2~13,4	83,8	72,2	100,0	23,3	48,0	4,0
10^2	76,0~4,5	74,4~31,4	23,9~11,7	83,8	72,2	100,0	23,3	39,0	4,0
Sequential / IncLp									
10ep	**50,0~3,5**	**49,9~31,2**	**19,6~3,2**	52,9	44,3	100,0	23,3	26,0	14,0

O algoritmo que obteve os melhores resultados nos testes foi Tiling/BCP, que também obteve bons resultados no treinamento. O OffTiling/ BCPMin (ver Tabela 8.70), quando criou as redes com 10 neurônios em média, obteve resultados equivalentes aos obtidos pelo Tiling/BCPMin.

Note que, nos algoritmos que constroem a rede em camadas com mais de um neurônio por camada, quando o número total de neurônios da rede fica em torno de 10, o desempenho no teste melhora. Conclui-se que redes com 2,5 camadas em média e com 10 neurônios distribuídos nessas camadas, caracterizam uma boa arquitetura para esse domínio.

Surpreendentemente o algoritmo Sequential/BCPMax somente foi superado, com relação ao desempenho nos teste, pelas três versões do Tiling e pelas versões BCPMin dos algoritmos Pyramid, PTI e OffTiling.

8.2.7.2 O Domínio Monks2

As Tabelas 8.72 a 8.81 mostram os resultados dos experimentos no domínio Monks2.

Tabela 8.72 – PMR, BCPMin, MinOver e Thermal Modificado / Monks2

Nº de Iterações	Precisão de Treinamento	Precisão de Teste	Maior Precisão de Treinamento	Menor Precisão de Treinamento	Maior Precisão de Teste	Menor Precisão de Teste
PMR						
1ep	67,2~1,2	65,4~12,4	69,1	64,6	90,9	46,5
10ep	68,0~1,2	**65,5~10,2**	69,9	65,5	81,8	51,2
10^2ep	68,3~0,9	60,2~13,8	69,7	66,8	76,7	27,9
10^3ep	**69,1~1,2**	61,3~6,9	70,4	66,8	69,8	53,5
BCPMin						
1	66,7~1,0	64,7~11,3	68,1	65,2	86,4	46,5
10	67,1~1,2	**67,1~10,4**	68,9	64,7	90,9	53,5
10^2	67,0~1,5	66,2~10,3	69,9	64,7	88,6	53,5
10^3	**67,4~1,3**	66,6~10,9	69,9	95,2	90,9	53,5
MinOver						
1ep	67,6~1,4	67,1~10,5	70,1	64,7	90,9	53,5
10ep	67,3~0,9	67,1~10,5	68,6	65,7	90,9	53,5
10^2ep	**67,9~1,1**	**67,3~11,5**	69,2	66,1	90,9	51,2
10^3ep	67,4~1,5	67,1~11,7	69,7	64,3	90,9	51,2
Themal Modificado						
1ep	57,1~16,5	55,0~20,2	68,4	31,4	76,7	9,1
10ep	59,9~14,8	**63,8~14,9**	68,4	31,4	90,9	37,2
10^2ep	**63,7~10,9**	64,1~14,7	68,6	32,6	90,9	34,9
10^3ep	57,1~16,5	54,7~20,3	68,4	31,4	76,7	9,1

Tabela 8.73 – Tower / Monks2

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Tower / PMR									
1ep	68,0~1,5	**64,1~7,0**	**6,9~4,5**	70,7	65,2	76,7	53,5	15,0	1,0
10ep	73,9~3,6	62,7~10,7	12,1~5,9	79,4	69,7	79,1	40,9	20,0	4,0
10^2ep	84,9~1,2	59,0~12,0	30,8~5,6	86,8	82,5	76,7	39,5	41,0	24,0
10^3ep	**88,9~1,8**	60,2~9,1	27,3~4,1	91,5	85,1	72,1	48,8	37,0	23,0
Tower / BCPMin									
1	66,7~1,0	64,9~12,8	**2,0~1,2**	68,1	65,2	90,9	44,2	4,0	1,0
10	67,4~1,0	66,6~10,1	2,7~1,3	69,4	65,9	88,6	53,4	6,0	2,0
10^2	67,5~1,3	67,1~10,5	2,2~0,9	69,4	65,2	90,9	53,5	4,0	1,0
10^3	**67,7~0,9**	**67,3~10,8**	2,5~0,5	68,9	65,7	90,9	53,5	3,0	2,0

Tabela 8.74 – Pyramid / Monks2

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Pyramid / PMR									
1ep	67,2~1,3	**67,1~10,5**	**2,3~0,7**	69,2	64,4	90,9	53,5	4,0	2,0
10ep	70,9~2,2	62,0~7,2	7,0~3,7	75,1	66,8	72,1	51,2	16,0	3,0
10^2ep	86,7~3,5	56,5~10,6	31,4~3,5	92,8	79,9	79,1	46,5	44,0	21,0
10^3ep	**90,6~1,2**	53,5~8,5	30,8~3,8	93,1	88,7	65,1	34,8	36,0	24,0
Pyramid / BCPMin									
1	66,6~1,1	65,2~11,5	**1,6~0,7**	68,1	64,7	88,6	48,8	3,0	1,0
10	67,0~0,9	**67,1~10,5**	1,7~0,5	68,4	64,9	90,9	53,5	2,0	1,0
10^2	**67,4~1,7**	65,5~10,3	1,8~0,9	70,7	64,4	86,4	53,5	3,0	1,0
10^3	67,3~1,0	66,8~10,0	1,7~0,8	68,4	65,5	88,6	53,5	3,0	1,0

Tabela 8.75 – Upstart / Monks2

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Upstart / PMR									
1ep	67,1~1,2	**67,0~10,5**	**17,1~25,9**	68,6	64,4	90,9	54,5	56,0	1,0
10ep	80,4~2,1	57,2~14,0	58,0~6,4	83,4	76,3	74,4	30,2	75,0	54,0
10^2ep	**89,6~1,6**	56,3~11,9	100,0~0,0	91,5	86,1	69,8	38,6	100,0	100,0
Upstart / BCPMin									
1	66,7~0,9	66,4~11,1	**100,0~0,0**	68,1	65,2	90,9	53,5	100,0	100,0
10	**67,2~1,0**	**67,1~10,5**	**100,0~0,0**	68,9	65,6	90,9	53,5	100,0	100,0
10^2	67,2~1,1	66,8~9,9	**100,0~0,0**	68,6	64,7	88,6	53,5	100,0	100,0

Tabela 8.76 – Shift / Monks2

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Shift / PMR									
1ep	67,3~1,3	**64,1~14,1**	**1,0~0,0**	68,9	64,4	90,9	37,2	1,0	1,0
10ep	72,0~2,9	54,4~15,7	6,6~3,7	76,3	67,9	74,4	23,2	12,0	1,0
10^2ep	78,4~4,3	55,8~11,0	12,2~5,3	87,4	70,7	67,4	31,8	21,0	2,0
10^3ep	**83,4~3,8**	62,1~14,0	13,1~3,5	88,1	74,0	83,0	34,9	19,0	6,0
Shift / BCPMin									
1	66,9~1,5	65,5~11,8	**1,3~0,7**	68,4	63,4	90,9	51,2	3,0	1,0
10	67,2~1,2	66,8~10,7	1,5~0,7	68,6	64,9	90,9	53,5	3,0	1,0
10^2	67,3~1,1	64,6~7,0	1,4~0,7	68,4	65,2	76,7	53,5	3,0	1,0
10^3	**67,5~1,4**	**67,1~10,3**	2,1~0,9	68,9	64,2	90,9	54,5	3,0	1,0

Tabela 8.77 – Perceptron Cascade / Monks2

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Perceptron Cascade / PMR									
1ep	67,6~1,2	61,8~13,8	**2,0~0,5**	69,4	65,5	79,5	37,2	3,0	1,0
10ep	69,3~0,7	**63,4~5,8**	2,6~0,8	70,4	68,4	70,5	53,5	4,0	1,0
10^2ep	77,2~4,1	54,9~10,8	8,3~3,4	83,0	68,9	74,4	32,6	14,0	1,0
10^3ep	**82,6~6,3**	62,3~7,3	10,8~5,3	87,9	71,0	72,1	51,2	19,0	2,0
Perceptron Cascade / BCPMin									
1	66,7~1,1	65,5~11,5	1,7~0,5	68,1	64,9	88,6	48,8	2,0	1,0
10	67,5~1,1	67,1~10,5	1,8~0,8	69,4	65,5	90,9	53,5	3,0	1,0
10^2	**67,6~1,1**	**67,8~11,2**	**1,6~0,8**	69,2	65,5	90,9	53,5	3,0	1,0
10^3	67,5~0,9	66,4~9,2	2,2~0,9	68,6	65,7	86,4	53,5	4,0	1,0

Tabela 8.78 – Tiling / Monks2

Nº It.	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de (CA)	Maior CA	Menor CA
Tiling / PMR												
1ep	67,2~1,2	**65,9~10,9**	**1,0~0,0**	68,6	64,4	90,9	53,5	1,0	1,0	1,0~0,0	1,0	1,0
10ep	67,8~1,2	62,0~10,9	**1,0~0,0**	69,4	65,2	76,7	39,5	1,0	1,0	1,0~0,0	1,0	1,0
10^2ep	**90,3~14,9**	63,0~4,5	107,0~77,5	100,0	68,1	72,1	53,5	200,0	1,0	7,2~4,4	11,0	1,0
10^3ep	84,9~14,9	64,8~5,7	54,6~51,7	100,0	67,4	72,1	53,5	137,0	1,0	4,1~2,8	7,0	1,0
Tiling / BCPMin												
1	70,9~10,5	67,8~14,8	**18,2~30,8**	100,0	64,7	97,7	53,5	91,0	1,0	2,2~2,3	8,0	1,0
10	83,6~15,1	77,5~15,7	34,1~29,2	100,0	66,1	100,0	55,8	79,0	1,0	3,2~1,5	5,0	1,0
10^2	90,3~15,7	**80,3~17,5**	22,8~19,6	100,0	64,4	100,0	55,8	67,0	1,0	3,1~1,5	6,0	1,0
10^3	**93,7~11,2**	76,7~19,8	26,9~18,7	100,0	67,9	100,0	41,9	69,0	1,0	3,0~1,1	5,0	1,0
Tiling Híbrido												
1ep	67,6~2,6	65,6~12,0	**6,8~10,6**	74,0	64,9	90,9	48,8	32,0	1,0	1,4~0,7	3,0	1,0
10ep	77,8~13,4	70,3~14,7	32,4~33,6	100,0	66,6	100,0	46,5	98,0	1,0	3,1~2,0	6,0	1,0
10^2ep	90,0~14,0	81,7~13,0	26,4~18,2	100,0	67,1	100,0	58,1	52,0	1,0	3,1~1,3	5,0	1,0
10^3ep	**96,5~11,0**	**86,1~13,7**	28,9~18,7	100,0	65,2	100,0	62,7	64,0	1,0	3,9~1,7	7,0	1,0

Tabela 8.79 – PTI / Monks2

Nº It.	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de Camadas (CA)	Maior CA	Menor CA
PTI / PMR												
1ep	67,4~0,9	**63,9~6,3**	**1,0~0,0**	68,6	65,7	76,7	53,5	1,0	1,0	1,0~0,0	1,0	1,0
10ep	68,3~1,4	62,9~5,6	**1,0~0,0**	70,1	65,2	72,1	53,5	1,0	1,0	1,0~0,0	1,0	1,0
10^2ep	68,5~1,2	60,8~11,7	**1,0~0,0**	70,2	66,2	77,3	44,2	1,0	1,0	1,0~0,0	1,0	1,0
10^3ep	**69,0~1,2**	63,6~7,3	**1,0~0,0**	71,2	67,0	69,8	46,5	1,0	1,0	1,0~0,0	1,0	1,0
PTI / BCPMin												
1	66,5~1,2	64,7~12,3	**1,0~0,0**	68,1	64,7	88,6	44,2	1,0	1,0	1,0~0,0	1,0	1,0
10	67,0~1,1	67,0~10,5	**1,0~0,0**	68,1	64,9	90,9	53,5	1,0	1,0	1,0~0,0	1,0	1,0
10^2	67,0~1,5	67,1~10,5	**1,0~0,0**	69,2	64,2	90,9	53,4	1,0	1,0	1,0~0,0	1,0	1,0
10^3	**67,2~1,1**	**67,3~10,5**	**1,0~0,0**	68,4	64,9	90,9	53,4	1,0	1,0	1,0~0,0	1,0	1,0

Tabela 8.80 – OffTiling / Monks2

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
OffTiling / PMR									
1ep	81,5~4,5	62,0~13,5	215,3~2,3	87,3	73,0	81,4	34,9	219,0	100,0
10ep	98,9~1,4	77,9~19,2	200,3~26,9	100,0	96,7	100,0	41,9	218,0	100,0
10^2ep	**100,0~0,0**	**77,9~14,7**	**64,7~12,2**	100,0	100,0	93,0	41,8	77,0	39,0
OffTiling / BCPMin									
1	65,6~2,4	63,4~14,3	17,3~2,5	68,6	60,9	81,4	32,6	21,0	13,0
10	**71,1~3,9**	62,9~13,5	17,6~2,4	76,9	64,5	86,0	41,9	22,0	15,0
10^2	70,3~3,3	**68,2~16,7**	16,7~2,9	74,2	65,8	93,0	37,2	23,0	13,0
10^3	69,2~4,8	64,3~16,0	**16,7~1,2**	77,9	62,7	90,9	39,5	18,0	15,0

Tabela 8.81 – Sequential / Monks2

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Sequential / BCPMax									
1	40,9~3,1	36,2~12,7	91,5~19,2	45,2	36,2	53,5	9,0	131,0	66,0
10	49,7~2,3	49,9~14,3	63,9~9,2	52,9	46,5	67,4	32,6	85,0	56,0
10^2	**51,2~2,3**	**51,7~15,3**	**58,3~5,1**	54,8	48,6	84,9	34,9	66,0	50,0
Sequential / IncLp									
10ep	**32,9~1,2**	**32,9~10,5**	**29,6~2,2**	35,6	31,4	46,5	9,1	33,0	27,0

A grande maioria dos algoritmos analisados obteve resultados com média abaixo de 70% com relação à precisão de teste. Assim como nas tabelas anteriores, o algoritmo que obteve os melhores resultados no treinamento foi o OffTiling/PMR. Desta vez no entanto, o bom desempenho no treinamento ajudou a obter um desempenho razoável nos testes. Os melhores desempenhos, entretanto, foram obtitos pelos algoritmos Tiling/BCPMin e Tiling/Híbrido (ver Tabela 8.78). Com base nesses resultados, pode ser inferido que, os melhores desempenhos foram obtidos por meio do uso do algoritmo BCPMin na arquitetura de uma rede Tiling.

8.2.7.3 O Domínio Monsk3

As Tabelas 8.82 a 8.91 mostram os resultados dos experimentos no domínio Monks3.

Tabela 8.82 – PMR, BCPMin, MinOver e Thermal Modificado / Monks3

Nº de Iterações	Precisão de Treinamento	Precisão de Teste	Maior Precisão de Treinamento	Menor Precisão de Treinamento	Maior Precisão de Teste	Menor Precisão de Teste
PMR						
1ep	77,4~1,5	75,9~11,0	79,9	75,3	88,6	54,5
10ep	84,4~1,8	76,6~12,3	87,7	82,0	100,0	58,2
10^2ep	89,2~0,7	**81,7~9,1**	90,2	88,2	95,3	62,8
10^3ep	**89,5~0,7**	81,5~9,0	90,5	88,4	93,0	60,5
BCPMin						
1	81,6~2,0	73,5~17,9	85,1	78,4	100,0	46,5
10	88,5~1,4	82,4~11,4	90,5	85,6	100,0	62,8
10^2	89,0~1,4	**83,6~6,4**	90,5	86,6	95,3	74,4
10^3	**89,4~0,8**	82,7~8,7	90,5	88,2	95,4	65,1
MinOver						
1ep	78,6~3,1	73,1~13,5	83,5	74,6	95,4	46,5
10ep	**82,2~2,5**	62,3~17,9	86,4	78,2	93,0	41,8
10^2ep	80,1~4,2	71,3~16,7	84,1	69,9	97,7	34,9
10^3ep	80,7~3,8	**73,6~15,1**	86,6	74,7	93,0	53,4
Themal Modificado						
1ep	52,7~2,7	**52,7~24,8**	57,8	50,0	77,3	6,9
10ep	**52,8~2,7**	47,2~24,7	57,8	50,0	74,4	6,9
10^2ep	**52,8~2,7**	**52,7~24,8**	57,8	50,0	77,3	0,0
10^3ep	**52,8~2,7**	47,2~24,7	57,8	50,0	74,4	6,9

Tabela 8.83 – Tower / Monks3

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Tower / PMR									
1ep	80,4~2,5	74,8~15,5	4,8~2,4	85,8	77,4	90,9	40,9	11,0	3,0
10ep	88,1~2,4	82,2~14,0	5,1~1,9	93,8	85,8	100,0	59,1	8,0	3,0
10^2ep	91,7~3,5	**86,6~9,0**	3,4~1,6	96,9	88,2	100,0	74,4	7,0	2,0
10^3ep	**93,6~3,6**	84,2~17,9	**3,0~0,9**	97,4	89,7	100,0	48,8	5,0	2,0
Tower / BCPMin									
1	81,9~2,7	73,8~18,5	**2,4~1,0**	87,1	78,4	100,0	51,2	5,0	2,0
10	89,9~3,8	87,3~11,1	2,9~1,0	96,9	86,3	100,0	72,1	5,0	2,0
10^2	90,7~3,3	87,3~13,2	2,9~1,1	96,9	87,2	100,0	63,6	5,0	2,0
10^3	**92,1~3,6**	**88,7~12,1**	2,7~0,7	96,9	88,4	100,0	65,1	4,0	2,0

Tabela 8.84 – Pyramid / Monks3

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Pyramid / PMR									
1ep	82,2~3,1	79,0~18,4	5,2~1,6	88,1	77,9	100,0	36,4	7,0	3,0
10ep	91,4~4,2	84,1~14,5	4,6~1,8	96,4	85,1	100,0	58,1	9,0	3,0
10^2ep	93,1~3,6	87,3~11,9	**2,9~0,9**	96,9	88,9	100,0	70,5	4,0	2,0
10^3ep	**93,3~3,7**	**88,4~10,8**	3,0~0,8	97,4	88,4	100,0	69,8	4,0	2,0
Pyramid / BCPMin									
1	81,4~2,4	71,7~17,7	1,7~0,5	85,1	77,6	100,0	46,5	2,0	1,0
10	87,6~2,9	**84,3~7,7**	1,9~1,0	90,2	79,9	100,0	76,7	4,0	1,0
10^2	**89,4~0,7**	82,2~9,3	**1,6~0,5**	90,5	88,2	95,3	63,6	2,0	1,0
10^3	89,2~1,6	83,1~6,2	2,0~0,0	90,5	85,1	93,1	74,4	2,0	2,0

Tabela 8.85 – Upstart / Monks3

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Upstart / PMR									
1ep	81,5~1,4	76,9~12,9	52,0~17,6	84,0	79,7	93,0	54,5	67,0	3,0
10ep	91,8~3,0	80,2~13,1	41,0~26,3	95,6	86,4	97,7	52,3	60,0	2,0
10^2ep	**92,6~3,7**	**87,8~8,8**	**31,6~47,9**	96,9	88,2	100,0	74,4	100,0	1,0
Upstart / BCPMin									
1	83,9~1,8	**70,3~17,9**	100,0~0,0	87,1	82,0	95,5	46,5	100,0	100,0
10	94,2~2,7	88,0~8,0	**61,3~43,8**	96,9	88,2	100,0	77,3	100,0	4,0
10^2	**96,3~3,3**	87,7~12,2	67,7~45,1	100,0	91,0	100,0	62,8	100,0	4,0

Tabela 8.86 – Shift / Monks3

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Shift / PMR									
1ep	78,3~2,7	72,1~14,4	**1,4~0,7**	82,5	75,1	90,7	43,2	3,0	1,0
10ep	88,4~3,6	75,9~14,1	2,1~0,7	92,3	83,3	95,3	55,8	3,0	1,0
10^2ep	92,1~3,6	88,7~11,7	1,6~11,7	96,9	87,7	100,0	69,8	3,0	1,0
10^3ep	**93,4~3,9**	**89,6~11,3**	1,7~0,7	97,4	88,4	100,0	74,4	3,0	1,0
Shift / BCPMin									
1	87,8~5,3	79,6~19,0	1,6~0,5	96,4	82,0	100,0	53,5	2,0	1,0
10	88,9~2,5	83,9~12,0	**1,3~0,5**	95,4	86,9	100,0	63,6	2,0	1,0
10^2	89,8~2,6	86,3~13,1	1,5~0,7	96,9	87,7	100,0	55,8	3,0	1,0
10^3	**91,6~3,7**	**87,0~10,5**	1,8~0,6	96,9	88,2	100,0	74,4	3,0	1,0

Tabela 8.87 – Perceptron Cascade / Monks3

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Perceptron Cascade / PMR									
1ep	80,2~3,1	77,4~11,3	2,6~1,3	86,1	75,8	90,7	52,3	5,0	1,0
10ep	87,2~1,3	79,2~9,6	2,3~1,2	89,5	85,6	95,3	62,8	4,0	1,0
10^2ep	**89,3~0,8**	**82,4~6,1**	**1,2~0,4**	90,5	88,2	95,3	74,4	2,0	1,0
10^3ep	89,7~0,7	79,7~9,2	1,4~0,8	90,5	88,4	93,0	59,1	3,0	1,0
Perceptron Cascade / BCPMin									
1	83,7~3,0	73,6~17,7	2,6~1,3	88,2	78,9	95,5	46,5	4,0	1,0
10	94,5~5,1	**89,8~11,5**	2,8~1,8	100,0	86,9	100,0	69,8	6,0	1,0
10^2	93,5~5,2	88,9~10,0	**2,3~1,4**	100,0	88,4	100,0	74,4	5,0	1,0
10^3	**94,6~5,1**	86,3~13,9	3,1~1,5	100,0	88,4	100,0	65,1	6,0	1,0

Tabela 8.88 – Tiling / Monks3

Nº It.	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de Camadas (CA)	Maior CA	Menor CA
Tiling / PMR												
1ep	77,2~1,4	70,9~14,2	**1,0~0,0**	79,4	74,6	86,4	38,6	1,0	1,0	1,0~0,0	1,0	1,0
10ep	83,9~1,6	72,9~11,9	**1,0~0,0**	86,6	81,7	88,4	54,5	1,0	1,0	1,0~0,0	1,0	1,0
10^2ep	90,2~3,6	**85,9~13,2**	5,3~13,6	100,0	87,6	100,0	55,8	44,0	1,0	1,1~0,3	2,0	1,0
10^3ep	**92,7~5,1**	84,8~9,8	5,7~10,4	100,0	88,9	100,0	72,1	34,0	1,0	1,6~1,3	5,0	1,0
Tiling / BCPMin												
1	92,8~8,2	85,4~10,1	9,2~4,5	100,0	82,0	100,0	67,4	13,0	1,0	2,3~0,8	3,0	1,0
10	**100,0~0,0**	96,1~5,6	8,0~2,9	100,0	100,0	100,0	86,1	15,0	5,0	2,2~0,4	3,0	2,0
10^2	98,8~3,9	**98,4~3,1**	5,9~2,5	100,0	87,6	100,0	90,7	10,0	1,0	1,9~0,3	2,0	1,0
10^3	**100,0~0,0**	96,1~6,9	**5,2~1,6**	100,0	100,0	100,0	81,4	8,0	4,0	2,1~0,3	3,0	2,0
Tiling Híbrido												
1ep	93,6~8,4	82,9~21,8	8,0~6,4	100,0	812	100,0	46,5	21,0	1,0	2,6~1,3	5,0	1,0
10ep	97,5~5,6	94,9~7,9	7,2~5,2	100,0	86,4	100,0	74,4	15,0	1,0	2,3~0,8	3,0	1,0
10^2ep	99,0~3,0	96,3~8,0	6,0~4,0	100,0	90,5	100,0	74,4	14,0	1,0	2,1~0,6	3,0	1,0
10^3ep	**100,0~0,0**	**97,2~8,8**	**4,9~2,8**	100,0	100,0	100,0	72,1	12,0	3,0	2,1~0,3	3,0	2,0

Tabela 8.89 – PTI / Monks3

Nº It.	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de (CA)	Maior CA	Menor CA
PTI / PMR												
1ep	76,8~2,4	70,6~14,4	**1,0~0,0**	80,9	72,8	86,0	47,7	1,0	1,0	1,0~0,0	1,0	1,0
10ep	84,7~2,1	78,7~14,9	**1,0~0,0**	88,2	82,0	100,0	51,2	1,0	1,0	1,0~0,0	1,0	1,0
10^2ep	88,8~1,1	**86,1~8,7**	**1,0~0,0**	90,5	86,9	100,0	74,4	1,0	1,0	1,0~0,0	1,0	1,0
10^3ep	**89,6~0,7**	81,9~7,8	**1,0~0,0**	90,5	88,4	93,0	65,1	1,0	1,0	1,0~0,0	1,0	1,0
PTI / BCPMin												
1	87,6~8,8	80,3~15,2	**6,4~8,1**	100,0	78,9	100,0	58,2	24,0	1,0	1,7~0,9	3,0	1,0
10	96,2~5,1	87,1~10,1	12,5~9,9	100,0	88,6	100,0	72,1	29,0	1,0	2,1~0,9	3,0	1,0
10^2	**100,0~0,0**	97,0~3,9	19,3~13,5	100,0	100,0	100,0	88,4	41,0	5,0	2,1~0,3	3,0	2,0
10^3	**100,0~0,0**	**97,4~4,5**	18,0~9,2	1,00	100,0	100,0	86,1	36,0	9,0	2,1~0,3	3,0	2,0

Tabela 8.90 – OffTiling / Monks3

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
OffTiling / PMR									
1ep	92,9~8,4	83,7~11,2	208,0~2,8	98,9	71,9	100,0	69,8	214,0	90,0
10ep	**100,0~0,0**	97,4~5,6	13,1~10,2	100,0	100,0	100,0	83,7	33,0	5,0
10^2ep	**100,0~0,0**	**97,9~4,4**	**8,2~3,9**	100,0	100,0	100,0	88,5	17,0	5,0
OffTiling / BCPMin									
1	86,9~7,0	80,7~16,5	10,1~1,6	95,4	75,3	100,0	48,8	13,0	7,0
10	97,1~6,3	96,9~7,3	8,8~2,8	100,0	79,6	100,0	76,7	15,0	5,0
10^2	**99,9~0,1**	**97,7~5,0**	**8,3~3,2**	100,0	99,7	100,0	86,0	13,0	5,0
10^3	97,4~4,6	92,1~12,2	8,8~2,8	100,0	86,9	100,0	62,8	13,0	5,0

Tabela 8.91 – Sequential / Monks3

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
Sequential / BCPMax									
1	73,9~4,8	72,0~13,7	9,7~2,5	81,4	66,7	93,0	50,0	15,0	7,0
10	73,6~4,1	**72,3~13,7**	11,8~4,9	81,4	67,8	93,0	50,0	23,0	6,0
10^2	**74,4~4,7**	71,6~13,4	**8,5~2,5**	84,0	68,9	90,7	68,9	12,0	5,0
Sequential / IncLp									
10ep	**52,7~2,7**	**52,7~24,7**	**9,0~2,8**	57,8	50,0	77,3	6,9	15,0	5,0

Neste domínio, nota-se claramente que os algoritmos que constrem a rede em camadas, com mais de um neurônio por camada (a saber: Tiling, PTI e OffTiling), obtiveram significativa vantagem nos desempenhos relativos ao treinamento e ao teste. É importante ressaltar que, os algoritmos PTI/PMR (Tabela 8.89) e Tiling/PMR (Tabela 8.88) quando executados com 1ep e 10ep, não tiveram bons resultados por não conseguirem tornar a primeira camada confiável; dessa forma os resultados obtidos por esses algoritmos foram, na realidade, obtidos pelo PMR.

8.3 Avaliação dos Algoritmos Neurais Construtivos Multiclasses

Nesta seção são apresentados os resultados das avaliações realizadas com os algoritmos multiclasses discutidos nos Capítulos 6 e 7 deste trabalho. Os domínios utilizados foram: Íris, E.Coli, Balance, Contraceptive e Glass. Todos esses domínios possuem no mínimo 3 classes (ver Tabela 8.92) e foram extraídos do *UCI Repository* [Blake & Merz 1998].

Tabela 8.92 – Especificação dos domínios multiclasses

Dominio	**N° de instâncias**	**N° atributos**	**N° de Classes**
Íris	150	4	3
Ecoli	336	7	8
Balance	625	4	3
Contraceptive	1473	9	3
Glass	214	9	6

8.3.1 O Domínio Íris

O dominio Íris é um dos domínios mais conhecidos e utilizados em aprendizado indutivo de máquina [Duda & Hart 1973]. Este domínio possui três classes, cada uma delas representa 50 instâncias de treinamento, caracterizando um tipo de planta da espécie Íris. A classe Íris-Setosa, é linearmente separável das classes Íris-Versicolor e Íris-Virginica, entretanto, estas duas últimas não são linearmente separáveis entre si. Este domínio é descrito por 4 atributos numéricos e contínuos referentes a medidas (cm) feitas nas plantas pertencentes em cada uma das classes.

Tabela 8.93 – PMRWTA, BCPWTA, PMR Individual, BCP Individual / Íris

Nº de Iterações	Precisão de Treinamento	Precisão de Teste	Maior Precisão de Treinamento	Menor Precisão de Treinamento	Maior Precisão de Teste	Menor Precisão de Teste
PMRWTA						
1ep	84,1~12,6	74,0~37,1	97,0	66,7	100,0	0,0
10ep	98,1~0,7	94,7~10,8	99,3	97,0	100,0	66,7
10^2ep	98,7~0,4	**95,3~8,9**	99,3	98,5	100,0	73,3
10^3ep	**98,8~0,4**	**95,3~8,9**	99,3	98,5	100,0	73,3
BCPWTA						
1	49,3~4,9	47,3~42,4	55,5	42,2	100,0	0,0
10	57,8~11,2	52,0~37,7	87,4	47,4	100,0	6,7
10^2	87,9~1,4	**84,0~13,4**	89,6	85,9	100,0	53,3
10^3	**88,4~1,4**	82,7~13,8	90,4	86,7	100,0	53,3
PMR Individual						
1ep	83,9~11,4	52,7~49,5	98,5	70,4	100,0	0,0
10ep	**90,7~4,3**	82,7~20,7	95,5	82,9	100,0	33,3

10^2ep	89,3~2,6	**86,7~18,1**	94,1	85,9	100,0	46,7
10^3ep	90,2~3,5	84,6~18,9	94,8	85,2	100,0	53,3
BCP Individual						
1	85,0~7,3	**72,7~41,3**	94,1	75,6	100,0	6,7
10	86,4~10,9	64,6~42,8	97,8	68,8	100,0	0,0
10^2	84,5~11,9	50,0~45,2	97,1	70,4	100,0	0,0
10^3	**87,1~12,1**	54,0~47,4	97,8	71,1	100,0	0,0

Tabela 8.94 – MTower / Íris

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
MTower / PMRWTA									
10ep	98,2~0,6	94,7~9,3	**3,3~0,9**	99,3	97,8	100,0	73,3	6,0	3,0
10^2ep	**98,9~0,4**	**96,7~6,5**	3,6~1,3	99,3	98,5	100,0	80,0	6,0	3,0
MTower / BCPWTA									
10	58,1~10,3	52,7~37,1	**3,0~0,0**	85,9	49,6	100,0	6,7	3,0	3,0
10^2	**87,8~1,6**	**83,3~14,5**	3,3~0,9	89,6	85,2	100,0	53,3	6,0	3,0

Tabela 8.95 – MPyramid / Íris

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
MPyramid / PMRWTA									
10ep	98,1~0,5	95,3~7,7	**3,3~0,9**	99,3	97,8	100,0	80,0	6,0	3,0
10^2ep	**98,8~0,4**	**96,0~8,4**	**3,3~0,9**	99,3	98,5	100,0	73,3	6,0	3,0
MPyramid / BCPWTA									
10	58,0~11,0	51,3~38,4	**3,0~0,0**	88,2	47,7	100,0	6,7	3,0	3,0
10^2	**88,3~1,4**	**82,7~13,8**	3,6~1,3	90,4	85,9	100,0	53,3	6,0	3,0

Tabela 8.96 – MUpstart / Íris

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
MUpstart / PMRWTA									
10ep	98,4~0,6	93,3~13,3	3,2~0,4	99,3	97,8	100,0	60,0	4,0	3,0
10^2ep	**98,8~0,4**	**93,3~12,9**	**3,0~0,0**	99,3	98,5	100,0	60,0	3,0	3,0
MUpstart / BCPWTA									
10	59,0~10,8	53,3~36,9	3,7~0,7	88,1	48,1	100,0	6,7	5,0	3,0
10^2	**88,6~1,6**	**80,7~13,9**	**3,6~0,5**	91,1	86,7	100,0	53,3	4,0	3,0

Tabela 8.97 – MPerceptron Cascade / Íris

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
MPerceptron Cascade / PMRWTA									
10ep	98,6~0,5	**95,3~7,1**	3,4~0,7	99,3	97,8	100,0	80,0	5,0	3,0
10^2ep	**98,8~0,4**	95,3~8,9	**3,0~0,0**	99,2	98,5	100,0	73,3	3,0	3,0
MPerceptron Cascade / BCPWTA									
10	58,9~11,0	54,7~35,2	3,4~0,5	88,1	47,4	100,0	6,7	4,0	3,0
10^2	**88,5~1,4**	**80,7~13,5**	**3,3~0,5**	90,4	87,7	100,0	53,3	4,0	3,0

Tabela 8.98 – MTiling / Íris

Nº It,	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de Camadas (CA)	Maior CA	Menor CA
MTiling / PMRWTA												
10ep	98,2~0,8	**95,3~8,9**	**3,0~0,0**	99,3	97,0	100,0	73,3	3,0	3,0	1,0~0,0	1,0	1,0
10^2ep	**98,9~0,7**	94,0~12,7	4,6~3,7	100,0	97,8	100,0	60,0	14,0	3,0	1,2~0,4	2,0	1,0

MTiling / BCPWTA												
10	67,1~18,9	62,7~40,3	23,3~38,1	97,0	48,1	100,0	0,0	118,0	3,0	1,4~0,7	3,0	1,0
10^2	**89,6~4,7**	**84,0~14,5**	**7,0~8,8**	100,0	85,2	100,0	53,3	28,0	3,0	1,3~0,7	3,0	1,0

Como pode ser notado, as versões dos algoritmos multiclasses que usaram o PMRWTA ao invés do BCPWTA como algoritmo básico, obtiveram melhores resultados. Note que os algoritmos que usaram o PMRWTA obtiveram desempenhos semelhantes, principalmente considerando a precisão de treinamento. Os melhores desempenhos no conjunto de teste foram obtidos pelos algoritmos MTower e MPyramid.

8.3.2 O Domínio E.Coli

O domínio E.Coli (ver [Nakai & Kanehisa 1991] para informações sobre este domínio) possui 336 instâncias divididas em 7 atributos com dados relativos a bactérias gram-negativas. Este domínio está dividido em 8 classes que determinam a localização de sítios, nessas bactérias, de seqüência de proteínas, os quais, são determinados pela seqüência de aminoácidos nas células.

Tabela 8.99 – PMRWTA, BCPWTA, PMR Individual, BCP Individual / E.Coli

Nº de Iterações	Precisão de Treinamento	Precisão de Teste	Maior Precisão de Treinamento	Menor Precisão de Treinamento	Maior Precisão de Teste	Menor Precisão de Teste
PMRWTA						
1ep	74,4~6,2	55,1~43,7	83,8	63,6	100,0	0,0
10ep	88,2~1,7	76,5~22,1	90,8	85,4	100,0	36,4
10^2ep	**90,8~1,3**	77,8~18,9	93,0	89,4	100,0	52,9
10^3ep	90,5~1,5	**78,7~19,8**	93,1	89,1	100,0	45,5
BCPWTA						
1	24,3~2,6	22,4~32,4	28,4	17,8	90,9	0,0
10	32,5~5,4	26,7~30,5	45,2	26,1	87,8	0,0
10^2	76,5~3,0	69,2~15,0	80,5	72,5	97,1	42,4
10^3	**82,6~2,2**	**72,8~15,2**	86,8	79,9	97,1	54,5
PMR Individual						
1ep	74,3~6,4	62,4~34,4	85,5	65,9	100,0	0,0
10ep	82,0~4,3	66,4~29,4	89,1	76,5	100,0	9,1
10^2ep	87,2~2,6	**73,8~24,0**	91,1	84,4	100,0	42,4
10^3ep	**87,2~1,7**	71,2~26,6	89,7	84,5	100,0	39,4
BCP Individual						
1	81,2~3,7	68,9~31,5	88,8	77,8	100,0	15,2
10	85,1~3,3	**72,0~28,2**	89,4	79,9	100,0	27,3
10^2	84,8~4,4	69,4~25,2	90,1	75,1	100,0	36,4
10^3	**85,7~3,2**	69,9~3,2	90,1	80,8	100,0	36,4

Tabela 8.100 – MTower / E.Coli

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
MTower / PMRWTA									
10ep	90,2~1,5	**78,4~22,2**	**27,8~14,1**	92,4	88,1	100,0	30,3	56,0	6,0
10^2ep	**93,5~1,5**	76,3~18,1	37,2~13,8	96,4	91,7	100,0	48,5	48,0	8,0
MTower / BCPWTA									
10	32,1~5,4	25,1~30,3	**7,7~0,7**	44,2	26,1	84,8	0,0	8,0	8,0
10^2	**76,6~2,9**	**69,2~14,4**	9,2~3,2	80,5	72,5	97,1	48,5	16,0	8,0

Tabela 8.101 – MPyramid / E.Coli

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
MPyramid / PMRWTA									
10ep	90,1~1,3	79,6~18,3	**29,2~10,3**	92,1	88,1	100,0	42,4	48,0	14,0
10^2ep	**92,8~1,3**	**76,9~18,9**	35,1~11,6	95,4	91,1	100,0	48,5	56,0	16,0
MPyramid / BCPWTA									
10	31,8~4,3	25,4~31,0	**7,7~0,7**	39,9	26,7	90,9	0,0	8,0	8,0
10^2	**76,6~2,8**	**69,5~16,2**	8,4~2,1	80,2	73,2	97,1	36,4	14,0	6,0

Tabela 8.102 – MUpstart / E.Coli

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
MUpstart / PMRWTA									
10ep	88,3~1,3	76,0~23,6	**8,1~0,7**	89,8	86,4	100,0	27,7	9,0	7,0
10^2ep	**90,7~1,6**	**76,9~19,7**	8,4~1,3	93,4	88,7	100,0	50,0	10,0	6,0
MUpstart / BCPWTA									
10	32,3~5,9	25,4~32,0	**7,7~0,7**	46,9	25,7	93,9	0,0	8,0	6,0
10^2	**80,5~2,8**	**75,2~10,4**	8,9~1,3	84,5	76,5	97,1	60,7	11,0	6,0

Tabela 8.103 – MPerceptron Cascade / E.Coli

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
MPerceptron Cascade / PMRWTA									
10ep	88,8~1,2	**82,9~16,2**	**8,6~1,3**	90,8	86,8	100,0	57,8	11,0	8,0
10^2ep	**90,7~1,5**	77,5~19,7	8,8~0,8	93,4	88,4	100,0	45,5	10,0	8,0
MPerceptron Cascade / BCPWTA									
10	33,0~5,7	26,6~30,9	**7,7~0,7**	46,5	27,4	87,9	0,0	8,0	6,0
10^2	**79,6~2,7**	**70,1~15,3**	9,0~1,7	82,5	73,5	97,0	42,4	12,0	6,0

Tabela 8.104 – MTiling / E.Coli

Nº It,	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de Camadas (CA)	Maior CA	Menor CA
MTiling / PMRWTA												
10ep	87,9~2,2	**76,3~20,7**	**7,7~0,7**	91,7	85,8	100,0	38,2	8,0	6,0	1,0~0,0	1,0	1,0
10^2ep	**93,6~2,7**	74,5~17,8	64,0~94,7	97,5	89,7	97,1	50,0	318,0	6,0	1,5~0,5	2,0	1,0
MTiling / BCPWTA												
10	33,7~6,3	24,5~31,7	38,3~64,7	45,9	27,1	87,8	0,0	164,0	6,0	1,4~0,8	3,0	1,0
10^2	**76,3~3,3**	**67,3~18,0**	**25,3~55,4**	81,5	71,5	97,1	33,3	183,0	6,0	1,1~0,3	2,0	1,0

Considerando as versões PMRWTA, os algoritmos MTower, MPyramid e MTiling obtiveram, em sua maioria, redes relativamente grandes. Esse fato talvez tenha colaborado na

diminuição do poder de generalização dessas redes. Note que os algoritmos MUpstart e MPerceptron Cascade criaram redes menores e, consequentemente, obtiveram melhores resultados no conjunto de teste, principalmente no caso do MPerceptron Cascade.

Nas versões BCPWTA dos algoritmos multiclasses, apesar do tamanho da rede criada ter se mantido na média de 8,0 neurônios, salvo no caso do MTiling/BCPWTA (ver Tabela 8.104), nota-se uma pequena vantagem dos algoritmos MUpstart e MPerceptron Cascade sobre os demais algorimtos. Este fato permite concluir que não só o tamanho da rede criada, mas também a sua arquitetura influenciaram positivamente nos resultados relativos a este domínio.

8.3.3 O Domínio Balance

O domínio Balance [Siegler 1976], foi gerado a partir de resultados experimentais de um modelo psicológico denominado *balance scale phenomena*. O domínio possui 625 instâncias com 4 atributos e separados em 3 classes. Os atributos são: peso esquerdo, distância do peso esquerdo, peso direito e distância do peso direito. Cada exemplo é classificado de acordo com uma escala de balanceamento, que pode ser direita, esquerda, ou balanceado. O modo correto de definir a classe é fazendo (distância do peso esquerdo * peso esquerdo) e (distância do peso direito * peso direito).

Tabela 8.105 – PMRWTA, BCPWTA, PMR Individual, BCP Individual / Balance

Nº de Iterações	Precisão de Treinamento	Precisão de Teste	Maior Precisão de Treinamento	Menor Precisão de Treinamento	Maior Precisão de Teste	Menor Precisão de Teste
PMRWTA						
1ep	88,9~1,0	87,5~5,0	90,7	87,7	93,5	79,0
10ep	91,8~0,6	89,4~7,6	92,5	90,6	98,4	75,8
10^2ep	**92,2~0,5**	**90,1~3,7**	93,1	91,3	96,8	84,1
10^3ep	92,2~0,6	90,1~4,3	93,2	91,5	98,4	84,1
BCPWTA						
1	76,4~8,7	74,1~8,3	89,7	58,9	91,9	66,1
10	75,2~8,8	73,1~9,5	88,9	56,3	91,9	0,0
10^2	80,1~4,4	77,9~9,9	90,4	76,4	91,9	65,1
10	**83,8~2,8**	**81,8~5,8**	90,7	81,2	90,5	74,6
PMR Individual						
1ep	**90,7~0,9**	89,1~4,7	92,2	89,5	95,2	79,0
10ep	89,1~1,6	**89,3~4,8**	92,4	87,4	98,4	84,1
10^2ep	88,8~0,8	86,8~3,8	89,7	87,4	91,9	80,9
10^3ep	88,8~0,9	86,7~3,7	90,1	87,4	92,1	80,9
BCP Individual						
1	88,9~1,3	86,6~3,5	90,7	86,9	91,9	80,9
10	**89,5~1,7**	88,0~3,5	92,2	86,3	93,5	82,3
10^2	87,4~4,3	87,4~3,6	92,0	78,3	93,5	82,5
10^3	88,3~2,2	**88,3~3,1**	91,5	84,9	92,1	82,5

Tabela 8.106 – MTower / Balance

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
MTower / PMRWTA									
10ep	**94,8~1,1**	**90,6~6,2**	20,4~5,8	96,6	93,1	98,4	79,4	30,0	12,0
10^2ep	91,5~2,0	90,2~5,2	**8,7~4,8**	94,8	88,3	98,4	80,9	15,0	3,0
MTower / BCPWTA									
10	78,9~8,7	76,9~8,3	**5,4~2,8**	90,3	61,3	91,9	67,7	12,0	3,0
10^2	**83,6~4,8**	**80,8~8,0**	9,0~3,5	92,0	73,0	91,9	65,1	15,0	6,0

Tabela 8.107 – MPyramid / Balance

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
MPyramid / PMRWTA									
10ep	95,1~0,9	**90,1~6,3**	**24,0~6,2**	96,8	93,8	96,8	76,2	33,0	15,0
10^2ep	**95,9~1,1**	90,1~7,4	25,8~10,3	97,3	94,1	95,2	71,4	48,0	9,0
MPyramid / BCPWTA									
10	79,0~8,2	79,6~7,4	**6,3~2,2**	88,6	61,6	91,9	66,7	9,0	3,0
10^2	**83,3~4,1**	**83,1~5,8**	**6,3~2,2**	89,9	77,6	93,5	76,2	9,0	3,0

Tabela 8.108 – MUpstart / Balance

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
MUpstart / PMRWTA									
10ep	91,8~0,4	**91,2~4,7**	3,4~0,9	92,4	90,9	98,4	85,5	6,0	3,0
10^2ep	**92,3~0,6**	90,4~4,5	**3,4~0,7**	93,2	91,5	98,4	84,1	5,0	3,0
MUpstart / BCPWTA									
10	79,0~8,3	76,9~8,7	4,4~1,4	90,6	63,8	91,9	64,5	7,0	3,0
10^2	**82,1~4,9**	**81,1~6,9**	**4,0~0,8**	91,3	77,1	91,9	69,8	5,0	3,0

Tabela 8.109 – MPerceptron Cascade / Balance

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
MPerceptron Cascade / PMRWTA									
10ep	91,7~0,8	89,9~5,9	3,9~1,0	92,5	90,2	98,4	77,8	6,0	3,0
10^2ep	**92,1~0,6**	**90,0~4,5**	**3,2~0,4**	93,2	91,3	98,4	84,1	4,0	3,0
MPerceptron Cascade / BCPWTA									
10	78,6~7,9	76,2~6,9	**3,7~0,7**	89,2	61,5	91,9	66,7	5,0	3,0
10^2	**81,6~4,7**	**78,6~9,9**	4,0~0,8	91,5	76,4	91,9	66,7	5,0	3,0

Tabela 8.110 – MTiling / Balance

Nº It,	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de Camadas (CA)	Maior CA	Menor CA
MTiling / PMRWTA												
10ep	94,8~1,9	90,9~5,4	177,8~111,7	97,3	91,7	98,4	80,9	338,0	3,0	1,9~0,6	3,0	1,0
10^2ep	**95,5~2,9**	**92,3~3,3**	**28,1~21,8**	98,9	91,7	96,8	88,9	49,0	3,0	1,6~0,5	2,0	1,0
MTiling / BCPWTA												
10	77,3~9,1	74,3~9,8	20,2~54,4	89,2	58,3	91,9	58,1	175,0	3,0	1,3~0,9	4,0	1,0
10^2	**79,8~4,5**	**76,4~8,4**	**3,0~0,0**	88,4	71,9	91,9	66,7	3,0	3,0	1,0~0,0	1,0	1,0

Note que o MTiling/PMRWTA, apesar de ter obtido uma média relativamente alta no número de neurônios, foi o algoritmo que obteve o melhor desempenho quanto à precisão de

teste. Considerando todos os algoritmos, o número de neurônios não parece ter influenciado na precisão de teste, pois algoritmos como o MPyramid e MTiling, construíram suas redes com média superior a 20,0 neurônios e, no entanto, tiveram desempenho, quanto à precisão nos testes, comparável à redes com menor número de neurônio.

8.3.4 O Domínio Contraceptive

O domínio Contraceptive representa o problema de predição do método contraceptivo de uma mulher baseado em suas características demográficas e socioeconômicas. O domínio possui 1473 instâncias relativas à mulheres casadas e que ainda não tiveram filho. As instâncias são dadas por 9 atributos e divididas em 3 classes. As classes são referentes à escolha de um tipo de método contraceptivo, de curto prazo, de longo prazo ou nenhum.

Tabela 8.111 – PMRWTA, BCPWTA, PMR Individual, BCP Individual / Contraceptive

Nº de Iterações	Precisão de Treinamento	Precisão de Teste	Maior Precisão de Treinamento	Menor Precisão de Treinamento	Maior Precisão de Teste	Menor Precisão de Teste
PMRWTA						
1ep	49,7~1,8	**39,0~25,6**	52,2	46,4	76,3	0,0
10ep	53,4~1,3	32,4~14,0	56,3	52,2	48,6	0,0
10^2ep	53,6~1,4	33,5~16,6	56,7	51,8	51,7	0,0
10^3ep	**53,7~1,0**	30,0~15,3	56,2	52,5	46,9	0,0
BCPWTA						
1	36,0~4,8	**34,4~42,1**	42,5	28,2	98,6	0,0
10	36,1~4,8	**34,4~42,1**	42,5	28,3	98,6	0,0
10^2	43,3~5,8	31,1~26,4	49,5	35,4	82,3	0,0
10^3	**45,4~3,3**	26,5~16,4	49,5	39,3	50,3	0,0
PMR Individual						
1ep	41,5~9,0	37,1~24,2	49,2	16,9	70,1	0,0
10ep	48,8~3,9	**50,6~23,2**	55,6	42,8	82,3	2,0
10^2ep	52,6~1,3	41,2~17,7	54,4	50,3	68,2	7,5
10^3ep	**53,9~0,8**	40,6~12,8	55,1	52,3	62,2	21,8
BCP Individual						
1	32,7~4,1	33,3~25,3	37,7	27,2	66,1	4,1
10	33,8~4,6	41,6~21,5	46,3	29,1	65,5	0,7
10^2	**34,3~5,7**	37,4~34,8	46,8	25,5	85,0	0,7
10^3	33,4~6,6	**44,5~31,6**	46,9	22,8	77,6	0,0

Tabela 8.112 – MTower / Contraceptive

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
MTower / PMRWTA									
10ep	**54,7~1,1**	**33,6~17,2**	10,5~3,8	57,5	53,5	51,4	0,7	18,0	3,0
10^2ep	36,4~6,3	26,0~29,6	**5,1~2,5**	50,2	26,5	73,5	0,0	9,0	3,0
MTower / BCPWTA									
10	36,1~4,8	**34,4~42,0**	**3,3~0,9**	42,5	28,3	98,6	0,0	6,0	3,0
10^2	**43,3~5,8**	31,6~26,3	4,5~1,6	49,5	34,4	82,3	0,0	6,0	3,0

Tabela 8.113 – MPyramid / Contraceptive

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
MPyramid / PMRWTA									
10ep	54,4~2,1	30,9~15,2	**8,7~3,6**	58,7	52,6	52,7	5,4	15,0	3,0
10^2ep	**55,8~1,3**	**32,9~15,4**	15,3~9,1	58,5	54,1	50,0	2,0	33,0	3,0
MPyramid / BCPWTA									
10	36,2~4,8	**34,5~4,8**	**3,3~0,9**	42,5	28,3	98,6	0,0	6,0	3,0
10^2	**43,3~5,6**	31,6~26,1	3,6~1,3	49,8	35,7	82,3	0,0	6,0	3,0

Tabela 8.114 – MUpstart / Contraceptive

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
MUpstart / PMRWTA									
10ep	53,7~1,4	31,2~14,9	**3,9~1,1**	56,7	52,0	47,9	0,0	6,0	3,0
10^2ep	**54,5~1,4**	**38,3~17,6**	4,5~1,5	57,7	53,1	58,8	0,0	8,0	3,0
MUpstart / BCPWTA									
10	36,2~4,8	**34,4~42,0**	**3,1~0,3**	42,5	28,3	98,6	0,0	4,0	3,0
10^2	**43,6~5,5**	30,9~26,2	3,9~0,7	49,5	35,9	81,6	0,0	5,0	3,0

Tabela 8.115 – MPerceptron Cascade / Contraceptive

MPerceptron Cascade / PMRWTA									
Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
10ep	53,4~1,3	30,8~14,3	**3,6~0,7**	56,1	51,6	42,6	0,0	5,0	3,0
10^2ep	**54,5~1,2**	**34,6~15,9**	4,0~1,4	57,1	53,1	52,4	1,4	7,0	3,0
MPerceptron Cascade / BCPWTA									
10	36,1~4,8	**34,4~42,0**	**3,1~0,3**	42,5	28,3	98,6	0,0	4,0	3,0
10^2	**43,4~5,7**	31,1~26,3	3,3~0,5	49,5	35,4	82,3	0,0	4,0	3,0

Tabela 8.116 – MTiling / Contraceptive

Nº It,	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de Camadas (CA)	Maior CA	Menor CA
MTiling / PMRWTA												
10ep	52,9~1,6	**33,7~14,1**	**3,0~0,0**	56,3	49,8	47,6	8,2	3,0	3,0	1,0~0,0	1,0	1,0
10^2ep	**53,5~1,1**	30,7~16,9	**3,0~0,0**	55,7	52,,3	53,7	0,0	3,0	3,0	1,0~0,0	1,0	1,0
MTiling / BCPWTA												
10	36,2~4,8	**34,4~42,0**	**3,0~0,0**	42,5	28,3	98,6	0,0	3,0	3,0	1,0~0,0	1,0	1,0
10^2	**43,1~5,6**	31,7~26,1	**3,0~0,0**	49,7	35,2	82,3	0,0	3,0	3,0	1,0~0,0	1,0	1,0

Um fato interessante verificado na precisão de teste para este domínio foi que os algoritmos PMR individual e BCP individual obtiveram desempenho superior aos demais algoritmos, sendo que o melhor foi o PMR individual. Com relação ao desempenho no treinamento, no entanto, este fato não se confirma, pois os algoritmos que usaram o PMRWTA tiveram os melhores resultados.

Nota-se que os resultados de todos os algoritmos foram ruins, tanto no treinamento quanto no teste, fato devido, talvez, à ruídos e inconsistências presentes no conjunto de treinamento.

8.3.5 O Domínio Glass

O domínio Glass é composto por 214 instâncias formadas por 9 atributos relativos à quantidade de certos elementos químicos que podem fazer parte da composição do vidro. O domínio é dividido em 6 classes e cada classe é um tipo especifico de vidro, como vidro de carro, vidro de lâmpada, etc. O domínio foi usado em criminalística, pois quando é encontrado vidro na cena do crime este pode servir de evidência.

Tabela 8.117 – PMRWTA, BCPWTA, PMR Individual, BCP Individual / Glass

Nº de Iterações	Precisão de Treinamento	Precisão de Teste	Maior Precisão de Treinamento	Menor Precisão de Treinamento	Maior Precisão de Teste	Menor Precisão de Teste
PMRWTA						
1ep	71,4~10,5	63,4~47,6	89,6	60,0	100,0	0,0
10ep	86,2~4,1	59,3~45,8	93,8	81,3	100,0	0,0
10^2ep	99,9~0,2	90,6~15,3	100,0	99,5	100,0	57,1
10^3ep	**100,0~0,0**	**93,4~13,5**	100,0	100,0	100,0	57,1
BCPWTA						
1	44,5~5,8	42,0~47,7	52,3	37,3	100,0	0,0
10	98,5~0,5	**87,7~22,1**	99,5	97,9	100,0	33,3
10^2	99,0~1,0	81,5~27,4	100,0	97,4	100,0	28,6
10^3	**99,2~0,8**	83,9~26,2	100,0	97,9	100,0	23,8
PMR Individual						
1ep	59,1~13,3	1,8~5,7	76,5	39,9	18,2	0,0
10ep	62,8~10,9	13,2~31,2	75,6	51,0	100,0	0,0
10^2ep	77,4~3,7	49,9~40,9	82,9	72,9	100,0	0,0
10^3ep	**81,6~3,3**	**58,9~40,6**	86,5	77,6	100,0	0,0
BCP Individual						
1	**72,1~3,7**	**68,2~47,1**	75,6	64,6	100,0	0,0
10	68,7~11,7	64,9~45,5	80,8	39,1	100,0	0,0
10^2	68,3~11,9	64,4~45,6	83,4	39,1	100,0	0,0
10^3	69,4~11,6	64,9~45,5	80,3	39,1	100,0	0,0

Tabela 8.118 – MTower / Glass

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
MTower / PMRWTA									
10ep	89,8~3,9	78,0~33,8	11,3~5,4	94,3	83,9	100,0	0,0	24,0	6,0
10^2ep	**100,0~0,0**	**92,9~16,9**	**6,0~0,0**	100,0	100,0	100,0	47,6	6,0	6,0
MTower / BCPWTA									
10	98,6~0,6	83,9~25,1	**8,2~2,9**	100,0	97,9	100,0	28,6	12,0	6,0
10^2	**99,6~0,6**	**85,8~25,8**	9,4~2,9	100,0	98,4	100,0	23,8	12,0	6,0

Tabela 8.119 – MPyramid / Glass

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
MPyramid / PMRWTA									
10ep	89,4~2,8	64,2~40,7	9,3~4,5	93,8	85,4	100,0	0,0	18,0	6,0
10^2ep	**100,0~0,0**	**95,4~10,6**	**7,7~2,9**	100,0	100,0	100,0	68,2	12,0	5,0
MPyramid / BCPWTA									
10	98,8~0,7	**84,8~24,7**	**7,7~2,9**	99,5	97,4	100,0	23,8	12,0	5,0
10^2	**99,6~0,6**	83,4~25,7	8,3~4,3	100,0	98,4	100,0	23,8	18,0	5,0

Tabela 8.120 – MUpstart / Glass

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
MUpstart / PMRWTA									
10ep	88,0~2,9	74,6~34,3	**6,5~0,8**	93,3	84,5	100,0	0,0	8,0	6,0
10^2ep	**97,3~1,8**	**88,3~18,6**	6,6~0,8	99,5	94,7	100,0	54,5	8,0	5,0
MUpstart / BCPWTA									
10	98,7~0,6	82,0~27,8	**6,1~0,6**	99,5	97,9	100,0	23,8	7,0	5,0
10^2	**99,7~0,4**	**86,3~24,2**	6,5~0,9	100,0	98,9	100,0	23,8	8,0	5,0

Tabela 8.121 – MPerceptron Cascade / Glass

Nº de Iterações	Precisão de Treinamento (TR)	Precisão de Teste (TE)	Nº de Neurônios (NE)	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE
MPerceptron Cascade / PMRWTA									
10ep	88,3~2,7	56,2~42,7	6,9~0,9	92,2	84,9	100,0	0,0	8,0	6,0
10^2ep	**96,6~1,4**	**79,2~32,1**	**6,7~0,8**	98,5	94,8	100,0	9,5	8,0	5,0
MPerceptron Cascade / BCPWTA									
10	98,7~0,7	**82,9~26,6**	**6,0~0,5**	100,0	97,9	100,0	28,6	7,0	5,0
10^2	**99,3~0,7**	82,5~26,6	6,2~0,8	100,0	98,5	100,0	23,8	8,0	5,0

Tabela 8.122 – MTiling / Glass

Nº It,	TR	TE	NE	Maior TR	Menor TR	Maior TE	Menor TE	Maior NE	Menor NE	Nº de Camadas (CA)	Maior CA	Menor CA
MTiling / PMRWTA												
10ep	90,2~7,4	75,1~36,7	24,2~32,9	100,0	81,4	100,0	0,0	93,0	5,0	1,4~0,7	3,0	1,0
10^2ep	**100,0~0,0**	**94,4~9,2**	**7,5~3,4**	100,0	100,0	100,0	77,3	14,0	5,0	1,2~0,4	2,0	1,0
MTiling / BCPWTA												
10	**100,0~0,0**	81,0~27,3	18,2~5,1	100,0	100,0	100,0	33,3	30,0	10,0	2,1~0,3	3,0	2,0
10^2	**100,0~0,0**	**82,5~26,6**	**12,7~5,1**	100,0	100,0	100,0	23,8	19,0	5,0	1,7~0,5	2,0	1,0

Neste domínio, nota-se que as versões dos algoritmos que usaram o BCPWTA como algoritmo básico foram, em vários casos, melhores que as versões PMRWTA. Esse fato é evidenciado nos algoritmos MPerceptron Cascade e MUpstart. Apesar disso o melhor desempenho nos testes foi obtido pelo MPyramid na versão PMRWTA.

9 capítulo Conclusões

Este trabalho de pesquisa investigou empiricamente três frentes de pesquisa em aprendizado neural:

- Algoritmos para o treinamento de uma única TLU, a saber, o Thermal, o Thermal Modificado e o BCP.
- Algoritmos neurais construtivos, a saber: Shift, Offset, PTI, Perceptron Cascade e Sequential.
- Extensões para problemas multiclasses de algoritmos neurais construtivos duas classes, já avaliados empiricamente em [Palma Neto & Nicoletti 2005], a saber, Tower, Pyramid, Tiling, Upstart, foi também investigada a versão multiclasse do algoritmo Perceptron Cascade, cuja versão para duas classes foi também objeto de pesquisa desta dissertação.

9.1 Conclusões sobre Algoritmos de Treinamento de TLUs

A investigação dos algoritmos para o treinamento de TLUs foi conduzida com o objetivo de evidenciar outras alternativas ao PMR, para serem usadas por algoritmos construtivos. A avaliação empírica conduzida (e descrita no Capítulo 8) não considerou o Thermal, devido à forte dependência desse algoritmo dos valores iniciais associados ao parâmetro temperatura.

Com relação à avaliação empírica dos demais algoritmos, nos nove domínios de conhecimento, o Thermal Modificado foi o que obteve o pior resultado tanto no treinamento quanto no teste. O Thermal Modificado obteve resultados equivalentes aos resultados dos demais algoritmos somente no domínio Breast Cancer; nos outros oito domínios o algoritmo ficou abaixo da média com relação ao desempenho nos testes.

Na maioria dos domínios o algoritmo MinOver teve um desempenho um pouco inferior aos algoritmos PMR e BCPMin. Nos domínios Paridade-5, Ionosphere e Monks2, entretanto, o algoritmo obteve resultados equivalentes ou superiores aos demais algoritmos de treinamento de TLU. O algoritmo MinOver implementado incorpora duas pequenas alterações, propostas neste trabalho, que o tornam ligeiramente diferente da versão original.

O algoritmo PMR obteve o melhor resultado, no domínio Vestibular, entre todos os algoritmos avaliados, com relação ao desempenho no teste. Nos demais domínios os desempenhos do PMR e BCP foram similares. O algoritmo BCP implementado, assim como o MinOver, também teve seu desempenho melhorado por meio de uma alteração da versão original, proposta neste trabalho.

9.2 Conclusões sobre Algoritmos Neurais Construtivos

Com relação à investigação de algoritmos neurais construtivos, a avaliação empírica de cada um deles foi conduzida considerando duas versões: uma usando o PMR e outra usando o BCPMin para o treinamento de TLUs. Apesar do trabalho não ter investido na investigação dos algoritmos Tower, Pyramid, Tiling e Upstart, tais algoritmos (cada um em suas duas versões, PMR e BCP) foram também avaliadas empiricamente neste trabalho, com o objetivo de fornecer um amplo panorama de desempenhos de algoritmos neurais construtivos.

Os algoritmos Tower e Pyramid tiveram desempenhos bastante semelhantes tanto em suas versões PMR quanto em suas versões BCPMin (tal fato tinha já sido identificado em [Palma Neto 2004] na versão PMR). No domínio Paridade-5, entretanto, isso não se verificou. A versão Pyramid/BCPMin teve um desempenho consideravelmente inferior ao da versão Tower/BCPMin no treinamento. No teste essa tendência é também verificada, mas não de maneira tão acentuada. Os resultados obtidos evidenciam que a estratégia de construção da rede usada pelo Pyramid, de criar mais conexões, aumentando as dimensões do vetor de pesos a cada camada, não resulta em melhores resultados que aqueles obtidos pelo Tower.

Considerando os resultados de todos os algoritmos neurais construtivos, pode-se dizer que os obtidos pelo Upstart estão na média. Pode ser observado nas tabelas relativas ao desempenho do Upstart, no Capítulo 8, que esse algoritmo criou, na fase de treinamento, redes razoavelmente grandes que atingiram, em algumas situações, o limite máximo pré-estabelecido de 100 neurônios. Os resultados obtidos pelas duas versões do Upstart (PMR e

BCPMin) são semelhantes, exceto no domínio Paridade-5, no qual a versão PMR obteve resultados sensivelmente melhores.

Uma tendência que pode ser notada é a da versão Upstart/BCPMin criar redes maiores que as criadas pela versão Upstart/PMR; esse fato, entretanto, parece não interferir nos desempenhos dos algoritmos. As avaliações com o Upstart foram realizadas com o limite de 100 iterações (i.e, 100 épocas), dado o tempo excessivo que esse algoritmo leva para construir a rede.

O algoritmo Shift obteve o melhor resultado nos testes em dois dos nove domínios avaliados, a saber, Paridade-5 e Ionosphere. Esses resultados foram obtidos com a versão PMR. No restante dos domínios, no entanto, o uso do PMR ao invés do BCP não se mostrou vantajoso; na verdade o uso do BCP foi superior em pelo menos dois dos nove domínios.

O algoritmo Perceptron Cascade na versão PMR foi o algoritmo que obteve o melhor resultado no domínio Liver. No restante dos domínios o algoritmo manteve-se na média com relação aos resultados apresentados pelos outros algoritmos construtivos. Quanto ao uso do PMR ou do BCPMin repetiu-se a tendência verificada com os outros algoritmos; o uso de um ao invés do outro pode ser ligeiramente mais eficiente em um domínio, porém a diferença de desempenhos não chega a ser significante.

O algoritmo Tiling foi avaliado em três versões: Tiling/PMR, Tiling/BCPMin e Tiling/Híbrido esta última é proposta deste trabalho (Capítulo 4, Subseção 4.4.1). O Tiling/Híbrido pode usar tanto o PMR quanto o BCPMin; sempre que termina a construção de uma camada, o Tiling/Híbrido verifica o melhor algoritmo para treinar o próximo mestre. Qualquer que tenha sido o algoritmo usado para treinar o mestre, se for necessário tornar a camada confiável, o Tiling/Híbrido usará sempre o BCPMin para treinar os neurônios auxiliares.

O Tiling na versão BCPMin obteve o melhor desempenho na fase de teste, nos domínios Monks1 e Monks3. O Tiling/Híbrido obteve o melhor desempenho na fase de teste, no domínio Monks2. Nos domínios Monks as versões Tiling/BCPMin e Tiling/Híbrido ficaram acima da média quanto ao desempenho, tanto com relação ao treinamento quanto ao teste. Note que a versão Tiling/PMR, em muitos casos, não conseguiu criar uma rede com mais de uma camada. Focalizando os desempenhos das três versões no treinamento e teste, pode-se dizer que as versões Tiling/BCPMin e Tiling/Híbrido tiveram desempenhos semelhantes e superiores ao desempenho da versão Tiling/PMR.

Assim como no algoritmo Tiling, a versão PTI/PMR nem sempre conseguiu construir uma rede com mais de uma camada, apesar de que em dois dos nove domínios, a versão

PTI/BCPMin também não construiu uma rede com mais de uma camada. A versão PTI/BCPMin teve desempenho superior à versão PTI/PMR tanto na fase de treinamento quanto na fase de teste; a superioridade, entretanto, se mostrou mais acentuada no treinamento.

O PTI obteve os melhores resultados no teste em dois dos nove domínios, no Pima com a versão BCP e no Breast Cancer na versão PMR. Estes resultados, no entanto, não podem ser atribuídos ao PTI, pois ambos os resultados não foram obtidos por uma rede criada pelo PTI, e sim por apenas um neurônio treinado com o PMR, no caso do Breast Cancer, ou com o BCP, no caso do PTI.

Como visto anteriormente (Capítulo 5, Subseção 5.3.6), o algoritmo OffTiling é uma proposta deste trabalho que pode ser caracterizado como um algoritmo híbrido que integra dois outros algoritmos, a saber, o Tiling e o Offset. Inicialmente o Tiling é usado para obter uma representação confiável e booleana do conjunto de treinamento, por meio da criação da primeira camada da rede. O treinamento então prossegue usando o Offset. O algoritmo OffTiling é apresentado neste trabalho nas versões BCPMin e PMR. Tanto a versão OffTiling/BCPMin quanto a OffTiling/PMR usam o BCPMin para criar a primeira camada da rede. Na versão OffTiling/BCPMin o BCPMin é usado pelo Offset enquanto que na versão OffTiling/PMR, o PMR é usado pelo Offset.

A versão OffTiling/PMR conseguiu 100% de precisão no treinamento em todos os domínios com exceção do domínio Pima; neste domínio o algoritmo não terminou a execução em tempo hábil para nenhum valor de iteração. Com relação ao desempenho nos testes, o algoritmo manteve-se na média, salvo algumas exceções em que houve *overfitting*. O OffTiling foi o algoritmo que construiu redes com o maior número de neurônios dentre todas as redes construídas pelos demais algoritmos.

A proposta do algoritmo Sequential além de usar outro tipo de algoritmo de treinamento de TLU (no caso, o algoritmo IncLp – ver Capítulo 5, Subseção 5.6.2), também constrói a rede usando uma abordagem diferenciada das demais. Como comentado anteriormente, o Sequential foi o algoritmo com o pior desempenho dentre os algoritmos avaliados. Além do péssimo desempenho tanto no treinamento quanto no teste, o Sequential/IncLp é extremamente lento e, por essa razão, esta versão foi executada apenas dez épocas; mesmo assim, não foi possível obter resultados com dados do domínio Pima. É importante lembrar que este trabalho experimentou a versão Sequential/BCPMax que, embora tenha tido desempenho melhor que a versão Sequential/IncLp, ainda assim, os resultados estão aquém dos resultados obtidos pelos outros métodos.

Com relação ao uso do PMR ou do BCPMin em algoritmos construtivos pode ser verificado que o PMR foi melhor sucedido em algoritmos que constroem a rede a partir da saída, tias como o Upstart, Perceptron Cascade e, principalmente, o Shift.

O único domínio de dados booleanos é o Paridade-5. Neste domínio pode ser observado que todas as versões de algoritmos usando o PMR tiveram desempenho superior àquelas usando o BCPMin. Isso sugere que em domínios booleanos que descrevem o conceito de paridade, o PMR terá melhor desempenho que o BCPMin. Esse fato, entretanto, precisa ser confirmado.

O algoritmo BCPMin se mostrou superior ao PMR quando usado em algoritmos neurais construtivos que constroem a rede em camadas com mais de um neurônio, como o Tiling, o PTI e o OffTiling. O algoritmo BCPMin parece ser muito mais apropriado para tornar uma camada confiável, que o algoritmo PMR. Por essa razão o OffTiling/PMR e o Tiling/Híbrido são fortemente baseados no BCPMin.

Resumindo, quanto aos algoritmos construtivos pode-se dizer que Tiling/BCP e Shift/PMR foram os algoritmos que obtiveram os melhores desempenhos em teste dentre todos os algoritmos avaliados. Apesar do algoritmo PTI ter obtido os melhores resultados no teste em dois dos nove domínios, o mérito na obtenção desses resultados foi do algoritmo de treinamento da TLU e não do PTI.

Se considerado os resultados obtidos pelo PTI, como sendo mérito do algoritmo que treinou o neurônio, então como um resultado foi obtido pelo PMR e o outro pelo BCPMin, pode-se dizer que dependendo do conjunto de treinamento, o uso de algoritmo construtivo pode ser dispensado, pois somente com algum algoritmo de treinamento de TLU é possível obter bons resultados com relação à precisão de teste.

9.3 Conclusões sobre Algoritmos de Treinamento Multiclasses

A avaliação dos algoritmos multiclasses foi feita usando cinco domínios de dados (Íris, E.Coli, Balance, Contraceptive e Glass). Os algoritmos multiclasses podem ser abordados em dois grupos. O primeiro é constituído por aqueles algoritmos identificados neste trabalho como algoritmos multiclasses básicos, a saber: o PMRWTA, BCPWTA, PMR Individual e BCP Individual. O segundo é constituído pelos algoritmos construtivos multiclasses, a saber: MTower, MPyramid, MTiling, MUpstart e MPerceptron Cascade.

Focalizando apenas os algoritmos multiclasses básicos, o PMRWTA foi o algoritmo que obteve os melhores resultados, tanto no treinamento quanto no teste. Os algoritmos PMR Individual e BCP Individual obtiveram resultados semelhantes em três dos cinco domínios (a saber: E.Coli, Balance, Contraceptive), com relação aos outros dois o PMR Individual foi superior no domínio Íris e o BCP Individual levou uma pequena vantagem no domínio Glass.

Os algoritmos PMR Individual e o BCP Individual obtiveram desempenho superior quando comparados ao algoritmo BCPWTA. Apesar deste fato o BCPWTA foi escolhido para ser usado na comparação entre os algoritmos construtivos multiclasses por dois motivos: viabilizar uma comparação não tendenciosa com o PMRWTA, uma vez que ambos os algoritmos são WTA, e pelas vantagens garantidas pela abordagem WTA, tais como preservar o relacionamento entre classes e garantir que uma instância pertença a uma única classe.

Quanto aos algoritmos construtivos multiclasses, as versões que usaram o PMRWTA tiveram desempenhos superiores em três (Íris, E.Coli, Balance) dos cinco domínios quando comparadas às versões que usaram o BCPWTA. Nos outros dois domínios observou-se um desempenho semelhante entre as versões.

Os algoritmos construtivos multiclasses, quando comparados na mesma versão, obtiveram resultados, em sua maioria, semelhantes com relação ao desempenho no treinamento e no teste. Pode ser observado que em três domínios (E.Coli, Balance, Contraceptive) os algoritmos MUpstart e MPerceptron Cascade obtiveram melhor desempenho que os demais algoritmos. Em dois dos três desses domínios, no entanto, os algoritmos MTower, MPyramid e MTiling obtiveram redes grandes quando comparadas às redes geradas pelo MUpstart e pelo MPerceptron Cascade e, por essa razão os últimos foram considerados superiores nesses domínios.

Nos dois domínios restantes nota-se que os algoritmos MTower, MPyramid e o MTiling obtiveram desempenho superior ao MUpstart e ao MPeceptron Cascade. Nestes dois domínios não se observa a criação de redes grandes por parte dos algoritmos MTower e MPyramid.

Considerando o número reduzido de domínios utilizados e a semelhança dos resultados obtidos, é impossível identificar alguma tendência que aponte um candidato a ser o melhor (ou o pior). É possível notar, no entanto, que nos domínios em que os algoritmos MUpstart e MPerceptron Cascade tiveram os melhores desempenhos, os algoritmos MTower, MPyramid e MTiling saíram-se pior, e nos demais domínios ocorreu o contrário.

Tendo em vista as semelhanças das arquiteturas de redes geradas pelos algoritmos MUpstart e MPerceptron Cascade e das arquiteturas de redes geradas pelos demais

algoritmos, pode-se concluir que, dependendo do domínio, o desempenho da rede criada será melhor quando os algoritmos MUpstart ou MPerceptron Cascade forem usados ao invés dos algoritmos MTower, MPyramid e MTiling, e vice-versa.

9.4 Trabalhos Futuros

Como este trabalho abordou três frentes igualmente importantes no contexto de aprendizado neural construtivo, uma continuação natural deste trabalho envolveria investigações nas três frentes.

A pesquisa focalizando algoritmos para o treinamento de TLUs pode ter continuidade por meio da investigação dos algoritmos Loss Minimization [Hrycej 1992] e AdaTron [Anlauf & Biehl 1989], que foram citados em [Yang *et al.* 1998] como possíveis candidatos a treinarem TLUs como parte de um algoritmo construtivo.

Ao longo do desenvolvimento desse trabalho de mestrado e devido ao constante acompanhamento da literatura na área, foram evidenciados um número razoável de algoritmos neurais construtivos para duas classes que mereceriam uma investigação e avaliação cuidadosas. Entre eles, Patch [Barkema *et al.* 1993], SELF [Zhang 1994], Target Switch [Campbell & Perez 1995], Learn++ [Polikar *et al.* 2001], [Liu *et al.* 2001] e [Ghiassi & Saidane 2005].

Uma das propostas deste trabalho, que é a do uso de diferentes algoritmos para o treinamento de TLUs, durante a construção de uma rede usando um algoritmo neural construtivo, merece ser investigada com mais detalhes. A proposta descrita no trabalho envolveu apenas um algoritmo construtivo (o Tiling) e dois algoritmos para o treinamento de TLUs (PMR e BCP). Devido ao fato do relativo sucesso dessa abordagem híbrida, outras combinações mereceriam investigação.

Uma outra área de pesquisa é a da articulação de diferentes algoritmos, de maneira a obter uma abordagem híbrida que agregue as vantagens dos algoritmos envolvidos. Esse trabalho contemplou apenas uma possibilidade, a da combinação do Tiling e Offset no algoritmo OffTiling.

Os algoritmos apresentados e discutidos no Capítulo 5, com exceção do Perceptron Cascade e do Sequential, não têm versões para domínios multiclasses e, consequentemente, uma continuação dos trabalhos nessa área seria a da pesquisa das versões multiclasses desses algoritmos.

Referências Bibliográficas

[Amaldi 1994] Amaldi, E. "From Finding Maximum Feasible Subsystems of Linear System to Feedfoward Neural Network Design", Ph.D. Thesis No. 1282, Department of Mathematics, Swiss Federal Institute of Technology, Lausanne, 1994.

[Amaldi & Kann 1995] Amaldi, E.; Kann, V. "The Complexity and Approximability of Finding Maximum Feasible Subsystems of Linear Relations", *Theoretical Computer Science*, Vol. 147, Pages 181 – 210, 1995.

[Amaldi & Guenin 1997] Amaldi, E.; Guenin, B. "Two Constructive Methods for Design Compact Feedforward Networks of Threshold Units", *International Journal on Neural Systems*, Vol. 8, N. 5 & 6, Pages 629 – 646, 1997.

[Anderson 1995] Anderson, J. A. "*An Introduction to Neural Networks*", The MIT Press, Cambridge, M.A, 1995.

[Anlauf & Biehl 1989] Anlauf, J. K.; Biehl, M. "The AdaTron: An Adaptative Perceptron Algorithm", *Europhysics Letters*, Vol. 10, Pages 687 – 692, 1989.

[Barkema *et al.* 1993] Barkema, G.; Andree, H.; Taal, A. "The Patch Algorithm: Fast Design of Binary Feedforward Neural Networks", *Network: Computation in Neural Systems*, Vol. 4, Pages 393 – 407, 1993.

[Baum & Haussler 1989] Baum, E. B.; Haussler, D. "What Size Net Gives Valid Generalizaton?", *Neural Computation*, Vol. 1, Pages 151 – 160, 1989.

[Bertini *et al.* 2006] Bertini Jr., J. R.; Nicoletti, M. C.; Hruschka Jr., E. R. "A Comparative Evaluation of Constructive Neural Networks Methods using PMR and BCP as TLU Training Algorithms", In: *Proceedingns of the IEEE International Conference on Systems, Man and Cybernetics*, Taipei, Taiwan, Los Alamitos: IEEE Press, 2006.

[Bishop 1995] Bishop, C. M. "*Neural Networks for Pattern Recognition*", Oxford University Press, 1995.

[Blake & Merz 1998] Blake, C. L.; Merz, C. J. "UCI Repository of Machine Learning Databases", [http://www.ics.uci.edu/~mlearn/MLRepository.html], University of California, Department of Information and Computer Science, Irvine, CA, 1998.

[Block 1999] Block, J. "Tutorial: Collections", [http://java.sun.com/docs/books/tutorial/index.html], 1999.

[Burgess 1994] Burgess, N. "A Constructive Algorithm that Converges for Real-Valued Input Patterns", *International Journal on Neural Systems* Vol.5, N. 1, Pages 59 – 66, 1994.

[Campbell 1997] Campbell, C. "*Constructive Learning Techniques for Designing Neural Networks Systems*", Academic Press, San Diego, 1997.

[Campbell & Perez 1995] Campbell, C.; Perez, C. V. "The Target Switch Algorithm: A Constructive Procedure for Feedforward Neural Networks", *Neural Computation*, Vol. 7, Pages 1245-1264, 1995.

[Chen *et al.* 1995] Chen, C-H.; Parekh, R.; Yang, J.; Balakrishnan, K.; Honovar, V. "Analysis of Decision Boundaries Generated by Constructive Neural Learning Algorithms", In: *Proceedings of the World Congress on Neural Networks (WCNN'95)*, Pages 628 – 635, Washington D.C., July 17 – 21, 1995.

[Cun *et al.* 1990] Cun, Y. Le.; Denker, J.S.; Solla, S.A. "Optimal Brain Damage", In: *Advances in Neural Information Processing (2)*, Pages 598 – 605, Morgan Kaufmann, 1990.

[de Berg *et al.* 2000] de Berg, M.; van Kreveld, M.; Overmans, M.; and Schwarzkopf, O. "Convex Hulls: Mixing Things", Ch. 11 in "*Computational Geometry: Algorithms and Applications*", 2nd rev., Pages 235 – 250, Springer-Verlag, Berlin, 2000.

[Dougherty *et al.* 1995] Dougherty, J.; Kohavi, R.; Sahami, M. "Supervised and Unsupervised Discretization of Continuous Features", In: *Proceedings of the Twelfth International Conference on Machine Learning*, Pages 194 – 202, San Francisco, CA, 1995.

[Duda & Hart 1973] Duda, R. O.; Hart, P. E. "*Pattern Classification and Scene Analysis*", John Wiley & Sons, New York, 1973.

[Fahlman 1988] Fahlman, S. "Faster Learning Variations on Backpropagation: An Empirical Study", In: *Proceedings of the 1988 Connectionist Models Summer School*, Pages 38 – 51, Morgan Kaufmann, Los Altos, CA, 1988.

[Fahlman & Libiere 1990] Fahlman, S.; Libiere, C. "The Cascade Correlation Architeture", In: *Advances in Neural Information Processing Systems* 2, Pages 524 – 532, Morgam Kaufman, San Maateo, CA, 1990.

[Figueira *et al.* 2006] Figueira, L.; Palma Neto, L.; Bertini Jr, J. R.; Nicoletti, M. C. "Using Constructive Neural Networks for Detecting Central Vestibular System Lesion", *Applied Artificial Intelligence*, 2006.

[Fine 1999] Fine, T. L. "*Feedforward Neural Network Methodology*", Springer-Verlag, New York, 1999.

[Frean 1990a] Frean, M "The Upstart Algorithm: A Method for Constructing and Training Feedforward Neural Networks", *Neural Computation*, Vol. 2, Pages 198 – 209, 1990.

[Frean 1990b] Frean, M. "Small Nets and Short Paths: Optimising Neural Computation", Ph.D. Thesis – Center of Cognitive Science, University of Edinburh, Scotland, 147 pages, 1990.

[Frean 1992] Frean, M. "A Thermal Perceptron Learning Rule", *Neural Computation*, Vol. 4, Pages 946 – 957, 1992.

[Gallant 1986] Gallant, S. I. "Three Constructive Algorithms for Network Learning", In: *Proceedings Eight Annual Conference of the Cognitive Society*, Pages 652 – 660, Amherst, MA, August 15 – 17, 1986.

[Gallant 1990] Gallant, S. I. "Perceptron-Based Learning Algorithm", *IEEE Transactions on Neural Networks*, Vol. 1, N. 2, Pages 179 – 192, June, 1990.

[Gallant 1994] Gallant, S. I. "*Neural Network Learning & Expert Systems*", The MIT Press London, England, 1994.

[Ghiassi & Saidane 2005] Ghiassi, M.; Saidane, H. "A Dynamic Architecture for Artificial Neural Networks", *Neuralcomputing*, Vol. 63, Pages 397 – 413, 2005.

[Hajnal *et al.* 1987] Hajnal, A.; Maass, W.; Pudlak, P.; Szegekdy, M.; Turan, G. "Threshold Circuits of Bounded Deph", In: *Proceedings of 28th Annual IEEE Symposium Foundations of Computer Science*, Pages 99 – 110, IEEE Computer Society Press, 1987.

[Haykin 1994] Haykin, S. "*Neural Networks, a Comprehensive Foundation*", Prentice Hall International Edition, 1994.

[Hebb 1949] Hebb, D. "*The Organization of Behavior*", John Wiley, New York, 1949.

[Hrycej 1992] Hrycej, T. "*Modular Learning in Neural Networks*", John Wiley & Sons, New York, 1992.

[Kohonen 1989] Kohonen, T. "*Self-Organization and Associative Memory*", Springer-Verlag, New York, 1989.

[Kovács 1996] Kovács, Z. L. "*Redes Neurais Artificiais: Fundamentos e Aplicações*", 2ª Edição, Collegium Cognitio, São Paulo, 1996.

[Krauth & Mezard 1987] Krauth, W.; Mezard, M. "Learning Algorithms with Optimal Stability in Neural Networks", *Journal of Physics A*, Vol. 20, Pages 745 – 752, 1987.

[Langley 1995] Langley, P. "*Elements of Machine Learning*", Morgan Kaufmann, Palo Alto, CA, 1995.

[Liu *et al.* 2001] Liu, D.; Chang, T.-S.; Zhang, Y. "A New Learning Algorithm for Feedforward Neural Networks", In: *Proceedings of the IEEE International Symposium on Intelligent Control*, Pages 39 – 44, Mexico City, Mexico, September, 2001.

[Marchand & Golea 1993] Marchand, M.; Golea, M. "On Learning Simple Neural Concepts: From Halfspace Intersections to Neural Decision Lists", *Network*, Vol. 4, Pages 67 – 85, 1993.

[Marchand *et al.* 1990] Marchand, M.; Golea, M.; Rujan, P. "A Convergence Theorem for Sequencial Learning in Two-Layer Perceptrons", *Europhysics Letters*, Vol. 11, N. 6, Pages 487 – 492, 1990.

[Martinez & Estève 1992] Martinez, D.; Estève, D. "The Offset Algorithm: Building and Learning Method for Multilayer Neural Networks", *Europhysics Letters*, Vol. 18, N. 2, Pages 95 – 100, 1992.

[Mascioli & Martinelli 1995] Mascioli, F. M. F.; Martinelli, G. "A Constructive Algorithm for Binary Neural Networks: The Oil-Spot Algorithm", *IEEE Transaction on Neural Networks*, Vol 6, No 3, Pages 794 – 797, 1995.

[Mayoraz & Aviolat 1996] Mayoraz, E.; Aviolat, F. "Constructive Training Methods for Feedforward Neural Networks with Binary Weights", *International Journal of Neural Networks*, Vol. 7, N. 2, Pages 149 – 166, 1996.

[McCulloch & Pitts 1943] McCulloch, W. S.; Pitts, W. "A Logical Calculus of the Ideas Immanent in Nervous Activity", *Bulletin of Mathematical Biophysics*, Vol. 5, Pages 115 – 133, 1943.

[Mézard & Nadal 1989] Mézard M.; Nadal J. P. "Learning in Feedforward Layered Networks: The Tiling Algorithm", *Journal of Physics A*, Vol. 22, N. 12, Pages 2191-2204, 1989.

[Minsky & Papert 1969] Minsky, M.; Papert, S. "*Perceptrons: An Introduction to Computational Geometry*", MIT Press, Cambridge, M.A., 1969

[Mitchell 1997] Mitchell, T. M. "*Machine Learning*", McGranw-Hill Series in Computer Science, McGraw-Hill, 1997.

[Mitchison & Durbin 1989] Mitchison, G. J.; Durbin, R. M. "Bounds on the Learning Capacity of some Multilayer Networks", *Biological Cybernetics*, Vol. 60, Pages 345 – 356, 1989.

[Moller 1993] Moller, M. F. "A Scale Conjugated Gradient Algorithm for Fast Supervised Learning", *Neural Networks*, Vol. 6, Pages 525 – 534, 1993.

[Nakai & Kanehisa 1991] Nakai, K.; Kanehisa, M. "Expert System for Predicting Protein Localization Sites in Gram-Negative Bacteria", *PROTEINS: Structure, Function, and Genetics*, Vol. 11, Pages 95-110, 1991.

[Nilsson 1965] Nilsson, N. "*The Mathematical Foundations of Learning Machines*", McGraw-Hill, New York, 1965.

[Oja 1989] Oja, E. "Neural Networks, Principal Components, and Subspaces", *International Journal of Neural Systems*, Vol.1, Pages 61 – 68, 1989.

[Okabe *et al.* 1992] Okabe, A.; Boots, B.; Sugihara, K. "*Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*", John Wiley & Sons, Chichester, England, 1992.

[Palma Neto 2004] Palma Neto, L. G. "Redes Neurais Construtivas para a Classificação de Padrões", Dissertação de Mestrado, Universidade Federal de São Carlos, São Carlos, 134 pags., 2004.

[Palma Neto & Nicoletti 2004] Palma Neto, L. G.; Nicoletti M. C. "Aprendizado Neural Construtivo usando os Algoritmos Upstart e Cascade Correlation", RT-DC 002/2004, 24 pags., 2004.

[Palma Neto & Nicoletti 2005] Palma Neto, L. G.; Nicoletti M. C. "*Introdução às Redes Neurais Construtivas*", EdUFSCar – EdUFSCar, 192 pags., 2005.

[Parekh *et al.* 1995] Parekh, R.; Yang, J.; Honavar, V. "Constructive Neural Network Learning Algorithm for Multi-Category Classification", Technical Report ISU-CS-TR95-15a, Iowa State University, Ames, IA, 1995.

[Parekh *et al.* 1997a] Parekh, R.; Yang, J.; Honavar, V. "MUpstart – A Constructive Neural Network Learning Algorithm for Multi-Category Pattern Classification", In: *Proceedings of the IEEE/INNS International Conference on Neural Networks (ICNN'97)*, Vol. 3, Pages 1924 – 1929, Houston, TX, June 9 – 12, 1997.

[Parekh *et al.* 1997b] Parekh, R.; Yang, J.; Honavar, V. "Pruning Strategies for the MTiling Constructive Learning Algorithm", In: *Proceedings of the IEEE/INNS International Conference on Neural Networks (ICNN'97)*, Vol. 3, Pages 1960 – 1965, Houston, TX, June 9 – 12, 1997.

[Parekh 1998] Parekh, R. G. "Constructive Learning: Inducting Grammars and Neural Networks", Ph.D. Dissertation, Iowa State University, Ames, Iowa, 297 pages, 1998.

[Parekh *et al.* 1998] Parekh, R.; Yang, J.; Honavar, V. "Constructive Theory Refinement in Knowledge Based Neural Networks", In: *Proceedings of the IEEE/INNS International Joint Conference on Neural Networks (IJCNN'98)*, Pages 2318 – 2323, Anchorage, AK, May 4 – 8, 1998.

[Parekh *et al.* 2000] Parekh, R.; Yang, J.; Honavar, V. "Comparison of Performance of Variants of Single-Layer Perceptron Algorithms on Non-Separable Datasets", *Neural, Parallel, and Scientific Computation*, Vol. 8, Pages 415 – 438, 2000.

[Polikar *et al.* 2001] Polikar, R.; Udpa, L.; Udpa, S.; Honavar, V. "Learn++: An Incremental Learning Algorithm for Supervised Neural Networks", *IEEE Transactions on Systems, Man, and Cybernetics – Part C: Applications and Reviews*, Vol. 31, N. 4, Pages 497 – 508, November, 2001.

[Poulard 1995] Poulard, H. "Barycentric Correction Procedure: A Fast Method of Learning Threshold Units", In: *Proceedings of WCNN'95*, Vol. 1, Pages 710-713, Washington D.C., 1995.

[Poulard & Estèves 1995] Poulard, H.; Estèves, D. "A Convergence Theorem for Barycentric Correction Procedure", Technical Report 95180, LAAS-CNRS, Toulouse, France, May, 1995.

[Poulard & Labreche 1995] Poulard, H.; Labreche, S. "A New Threshold Unit Learning Algorithm", Technical Report 95504, LAAS, December, 1995.

[Reed 1993] Reed, R. "Pruning Algorithms – A Survey", *IEEE Transactions on Neural Networks*, Vol. 4, N. 5, Pages 740 – 747, September, 1993.

[Ripley 1996] Ripley, B. D. "*Pattern Recognition and Neural Networks*", Cambridge University Press, Cambridge, 1996.

[Rojas 1996] Rojas, R. "*Neural Networks a Systematic Introduction*", Springer-Verlag, Berlin, Heidelberg, 1996.

[Rosenblatt 1958] Rosenblatt, F. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain", *Psychological Review*, Vol. 65, Pages 386 – 408, 1958.

[Rosenbaltt 1962] Rosenblatt, F. "*Principles of Neurodynamics: Perceptron and the Theory of Brain Mecanism*", Spartan, Washington D.C., 1962.

[Rumelhart *et al*. 1986] Rumelhart, D.; Hinton, G.; Williams, R. J. "Learning Internal Representations by Error Propagation", *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1, MIT Press, Cambridge, M.A., 1986.

[Rumelhart & McClelland 1986] Rumelhart, D.; McClelland, J. "*Parallel Distributed Processing*", MIT Press, Cambrigde, M.A., 1986.

[Russel & Norvig 2004] Russel, N.; Norvig, P. "*Inteligência Artificial*", Elsevier Editora Ltda, 2004.

[Shynk 1990] Shynk, J. J. "Performance Surfaces of a Single-Perceptron", *IEEE Transactions on Neural Networks*, Vol. 1, Pages 268 – 274, 1990.

[Siegler 1976] Siegler, R. S. "Three Aspects of Cognitive Development", *Cognitive Psychology*, Vol. 8, Pages 481 – 520, 1976.

[Sietsma & Dow 1988] Sietsma, J.; Dow, R. J. F. "Neural Net Pruning – Why and How", In: *Proceedings of the IEEE International Conference on Neural Networks*, Vol. 1, Pages 325 – 333, San Diego, 1988.

[Sietsma & Dow 1991] Sietsma, J.; Dow, R. J. F. "Creating Artificial Neural Networks that Generalize", *Neural Networks*, Vol. 4, Pages 67 – 79, 1991.

[Sigillito *et al.* 1989] Sigillito, V. G.; Wing, S. P.; Hutton, L. V.; Baker, K. B. "Classification of Radar Returns from the Ionosphere using Neural Networks", Johns Hopkins APL Technical Digest, Vol. 10, Pages 262 – 266, 1989.

[Smieja 1993] Smieja, F. J. "Neural Networks Constructive Algorithms: Trading Generalization for Learning Efficiency?", *Cirtuits Systems and Sygnal Processing*, Vol. 12, N. 2, Pages 331 – 374, 1993.

[Squires & Shavlik 1991] Squires, C. S.; Shavlik, J. W. "Experimental Analysis of Aspects of the Cascade-Correlation Learning Architecture", Machine Learning Research Group Working Paper 91-1, Computer Science Department, University of Wisconsin-Maddison, 1991.

[Volpini *et al.* 2002] Volpini, P.; Figueira, L. B.; Colafemina, J. F.; Roque, A. C. "A Neural Network-Based System for Diagnosis of Central Vestibular Lesion", In: Valafar, F. (Ed.). *Proceedings. of the International Conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences-METMBS'02*, Pages 29 – 33, CSREA Press, 2002.

[Yang & Honavar 1996] Yang, J.; Honavar, V. "A Simple Randomized Quantization Algorithm for Neural Network Pattern Classifiers", In: *Proceedings of the World Congress on Neural Networks' 96*, Pages 223 – 228, San Diego, CA, 1996.

[Yang *et al.* 1996] Yang, J.; Parekh, R.; Honovar, V. "MTiling – A Constructive Neural Network Learning Algorithm for Multi-Category Pattern Classification", In: *Proceedings of the World Congress on Neural Networks*, Pages 182 – 187, 1996.

[Yang *et al.* 1998] Yang, J.; Parekh, R.; Honavar, V. "DistAl: An Inter-Pattern Distance-Based Constructive Learning Algorithm", In: *Proceedings of the IEEE/INNS International Joint Conference on Neural Networks (IJCNN'98)*, Pages 2208 – 2213, Anchorage, AK, May 4 – 8, 1998.

[Young & Downs 1998] Young, S.; Downs, T. "CARVE – A Constructive Algorithm for Real-Valued Examples", *IEEE Transactions on Neural Networks*, Vol. 9, N. 6, 1998.

[Zhang 1994] Zhang, B. T. "An Incremental Learning Algorithm That Optimizes Network Size and Sample Size in One Trial", In: *Proceedings of the IEEE International Conference on Neural Networks (ICNN'94)*, Vol. 1, Pages 215 – 220, Orlando, Florida, June, 1994.