Universidade Federal de São Carlos
Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

REDES NEURAIS CONSTRUTIVAS PARA A CLASSIFICAÇÃO DE PADRÕES

Luiz Garcia Palma Neto

São Carlos
Março de 2004

Universidade Federal de São Carlos
Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

REDES NEURAIS CONSTRUTIVAS PARA A CLASSIFICAÇÃO DE PADRÕES

Luiz Garcia Palma Neto

Dissertação de Mestrado apresentada no Programa de Pós Graduação em Ciência da Computação do Centro de Ciências Exatas e de Tecnologia da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Inteligência Artificial.

São Carlos
Março de 2004

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

P171rn

Palma Neto, Luiz Garcia.
Redes neurais construtivas para a classificação de padrões / Luiz Garcia Palma Neto. -- São Carlos : UFSCar, 2004.
134 p.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2004.

1. Perceptrons. 2. Redes neurais (computação). 3. Sistemas de reconhecimento de padrões. I. Título.

CDD: 006.42 (20ª)

“É preciso ter dúvidas. Só os estúpidos têm uma absoluta confiança em si mesmos”.

(Orsos Welles).

AGRADECIMENTOS

- À Profa. Dra. Maria do Carmo Nicoletti pela orientação, amizade, paciência e por todo conhecimento compartilhado durante o desenvolvimento do trabalho.
- Ao CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) pelo suporte financeiro prestado.
- Às Profas. Dras. Sandra Abib e Heloísa de Arruda Camargo pela leitura, análise e avaliação de alguns relatórios técnicos que ajudaram na elaboração desse trabalho.
- À Leonie C. Pearson pela ajuda quando da preparação da minha apresentação junto à ICMLA – International Conference on Machine Learning and Applications.
- À minha família pelo incentivo e carinho.
- Aos meus amigos Bruno Zanetti, Bruno Amaral, Evgueni Dodonov, Luís Gustavo Castanheira e Alessandro Baldão Quadrado pela amizade e apoio.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Aprendizado Neural Construtivo é um modelo de aprendizado neural que não pressupõe a definição de uma topologia de rede fixa 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 de pesquisa investiga seis algoritmos neurais construtivos, a saber, tower, pyramid, tiling, upstart, Distal e cascade-correlation, buscando avaliar cada um deles com relação a vantagens e desvantagens, facilidade no treinamento, tamanho e topologia de rede criada, restrições de uso e desempenho.

O trabalho apresenta um ambiente computacional (CONEB) que disponibiliza a implementação de cada um dos algoritmos. São apresentados e analisados os resultados obtidos utilizando os diferentes algoritmos em vários domínios de conhecimento.

ABSTRACT

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

This research work investigates six constructive neural algorithms namely, tower, pyramid, tiling, upstart, Distal and cascade-correlation, evaluating each of them with relation to advantages and disadvantages, ease of training, size and topology of the network, restrictions and performance.

The work presents a computational system (CONEB) which implements each algorithm. Results obtained by using the different algorithms in several knowledge domains are presented and analysed.

CAPÍTULO 1. INTRODUÇÃO

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

A maioria dos algoritmos neurais construtivos é baseada no algoritmo perceptron [Rosenblatt 1962] e/ou numa evolução do perceptron chamada algoritmo pocket com modificação ratchet [Gallant 1986]. De acordo com [Yang et al 1999] aprendizado neural construtivo tem as seguintes vantagens com relação às técnicas convencionais de treinamento da rede:

- Não há necessidade de se determinar a topologia da rede inicialmente. Algoritmos neurais construtivos determinam a topologia da rede dinamicamente, durante a fase de treinamento.
- Algoritmos construtivos têm o potencial de gerar redes pequenas, quase mínimas.
- Não geram erros de classificação em qualquer conjunto de treinamento que seja não contraditório e finito.
- Usam neurônios básicos que são treinados pelo algoritmo perceptron.

Apesar dessas inúmeras vantagens, entretanto, como apontado em [Campbell 1997], "o maior defeito dos algoritmos construtivos é o potencial que eles têm em incorporar no conceito, o ruído existente nos dados. O objetivo do treinamento dos algoritmos construtivos não é obter uma representação exata do conjunto de treinamento mas construir um modelo do processo que gerou aqueles dados. Infelizmente, a habilidade dos algoritmos construtivos em garantir convergência pode fazer com que eles consigam adequar perfeitamente o conjunto de treinamento, adequando, então, perfeitamente o ruído."

Esse trabalho focaliza os algoritmos construtivos tower e pyramid [Gallant 1986], algoritmo tiling [Mézard & Nadal 1989], upstart [Frean 1990a], DistAl [Yang et al 1997] e cascade-correlation [Fahlman & Lebiere 1990]. Os algoritmos tower, pyramid, tiling e upstart são limitados ao aprendizado a partir de instâncias que representam duas classes apenas. O algoritmo Distal não é limitado, podendo aprender em domínios com várias classes. O mesmo acontece com o cascade-correlation. Existem, entretanto, várias versões do algoritmo cascade-correlation. A versão implementada nessa dissertação é para o aprendizado em domínios com duas classes.

A escolha do tower e pyramid foi baseada no fato desse dois algoritmos serem versões construtivas do algoritmo perceptron. Os algoritmos tiling e upstart são algoritmos clássicos do aprendizado construtivo. A escolha do upstart foi motivada pela maneira diferenciada com que esse algoritmo constrói a rede neural. O interesse na investigação do Distal foi motivado pelo fato desse algoritmo ter sido proposto como um melhoramento dos algoritmos construtivos e, também, pelo fato desse algoritmo não ser iterativo, como os demais. O cascade-correlation é conhecido pelo seu bom desempenho e um dos algoritmos construtivos mais usados. O objetivo desse trabalho é investigar esse conjunto de algoritmos construtivos, buscando avaliar cada um deles com relação a desempenho, vantagens e desvantagens, facilidade no treinamento e 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. O trabalho está organizado como segue.

O Capítulo 2 apresenta o conceito de TLU, discute o algoritmo perceptron bem como dois algoritmos derivados do perceptron chamados de pocket e pocket com modificação ratchet [Gallant 1986]. Duas são as justificativas para a apresentação e discussão desses três algoritmos: padronização/formalização da notação e apresentação dos algoritmos básicos que subsidiam vários algoritmos neurais construtivos tratados nessa dissertação.

O Capítulo 3 discute os algoritmos tower e pyramid propostos por Gallant em [Gallant 1986] [Gallant 1990]. Ambos algoritmos podem ser abordados como versões construtivas do algoritmo perceptron. Esse trabalho propõe uma pequena modificação na proposta original desses dois algoritmos de maneira a adequá-los a uma específica situação de treinamento evidenciada empiricamente

O Capítulo 4 apresenta, discute e descreve o algoritmo tiling [Mézard & Nadal 1989]. Esse algoritmo introduz e usa as noções de neurônio mestre e neurônio auxiliar, na construção de cada camada da rede neural. A mesma modificação sugerida para o tower e pyramid pode ser também adotada para o tiling.

O Capítulo 5 investiga o algoritmo upstart [Frean 1990a], cuja estratégia de construção da rede é diferenciada das estratégias adotadas pelos outros métodos. O fato da descrição original do algoritmo não ser específica em determinados pontos, deixou margem a diferentes interpretações quando do treinamento da rede. Duas possíveis abordagens são propostas, discutidas e implementadas nesse trabalho.

O Capítulo 6 trata do Distal [Yang et al 1997], um algoritmo neural construtivo relativamente recente que difere dos demais por ser não iterativo o que, teoricamente, torna o aprendizado mais rápido que aquele realizado por um método iterativo.

O Capítulo 7 focaliza o algoritmo cascade-correlation [Fahlman & Lebiere 1990] que, diferentemente dos demais, não usa um treinamento baseado no perceptron e se caracteriza por ser um algoritmo bastante popular e eficiente.

O Capítulo 8 descreve experimentos de aprendizado realizados com os algoritmos em diferentes domínios de conhecimento e analisa e discute os resultados obtidos. Ao final do capítulo são apresentadas as conclusões e sugeridas atividades para a continuidade do trabalho descrito nessa dissertação.

O Anexo A traz algumas definições básicas com o objetivo de tornar a dissertação auto-contida. O Anexo B traz a prova de convergência do algoritmo perceptron. O Anexo C trata da implementação do Distal. O código original, que foi fornecido pelo autor da proposta, foi modificado em algumas partes de maneira a torná-lo mais amigável. O Anexo D descreve o sistema CONEB (Constructive Neural Builder), desenvolvido ao longo deste trabalho de pesquisa com o objetivo de criar uma ambiente computacional voltado ao aprendizado neural construtivo para subsidiar a experimentação com os diferentes algoritmos.

CAPÍTULO 2. PERCEPTRON E SEUS DERIVADOS

De acordo com [Gallant 1994, pg. 63], "modelos baseados em perceptron são atrativos porque são rápidos e poderosos para a modelagem de dados". As chamadas *threshold logic unit* (TLU), também conhecidas como perceptrons são usadas para encontrar vetores de peso $W = <w_0,w_1,...,w_p>$, de dimensão (p+1) que são usados para classificar um conjunto de treinamento descrito por instâncias, em classes.

Um conjunto de treinamento E consiste de n exemplos de treinamento notados por E^i, $1 \leq i \leq n$, cada um associado a uma classe C^i, ou seja:

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

onde cada E^i, $1 \leq i \leq n$ é um vetor p-dimensional de números reais correspondentes às entradas do problema e C^i é um número (ou vetor de números) correspondente à saída do problema, referenciado como 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 *bipola*r; se $C^i \in \{0,1\}$, a TLU é chamada *binária*. No que segue serão usadas TLUs bipolares.

Como comentado em [Parekh 1998, pg. 89] “TLUs bipolares implementam a construção de um hiperplano de dimensão (p−1) definido pelas coordenadas x_1, x_2,..., x_p que passa por W.X = 0 e divide o espaço das instâncias de treinamento em 2 regiões (correspondentes às duas classes das instâncias)”. A primeira região satisfaz a equação W.X > 0 e a segunda satisfaz W.X < 0 onde W.X é definido como:

$$W.X = w_0 + w_1x_1 + w_2x_2 +....+ w_px_p$$

onde o termo w_0 é chamado de *bias.*

Dada uma instância p-dimensional $X = (x_1,x_2,\ldots,x_p)$, a saída $o(x_1,x_2,\ldots,x_p)$ calculada pelo perceptron é:

$$o(x_1,x_2,\ldots,x_p) = \begin{cases} 1 \text{ se } w_0 + w_1x_1 + w_2x_2 + ... + w_px_p > 0 \\ -1 \text{ caso contrário} \end{cases}$$

onde cada w_i é uma constante real (peso), que determina a contribuição da entrada x_i na saída do perceptron. O valor $(-w_0)$ é um limite que a combinação linear de entradas $w_1x_1 + w_2x_2 +....+ w_px_p$ deve superar para que a saída do perceptron seja 1 [Mitchell 1997]. Para simplificar a notação, pode-se imaginar uma constante adicional de entrada, $x_0 = 1$, o que permite reescrever a fórmula anterior como:

$$o(x_1,x_2,\ldots,x_p) = \begin{cases} 1 \text{ se } \sum_{i=0}^{p} w_ix_i > 0 \\ -1 \text{ caso contrário} \end{cases}$$

Aprendizado usando perceptron consiste na determinação de valores para os pesos $w_0, w_1,\ldots,w_p$, como ilustra a Figura 1.

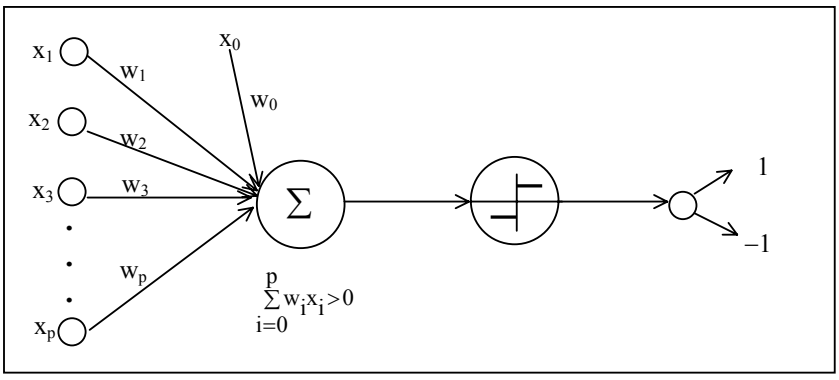


Figura 1. Diagrama do funcionamento do algoritmo perceptron

Como comentado anteriormente, o perceptron pode ser visto como a representação da superfície de um hiperplano em um espaço p-dimensional de instâncias. O perceptron vai ter saída 1 para instâncias que estão em um lado do hiperplano e saída −1 para instâncias que estão do outro lado do hiperplano. Isto pode ser visto na Figura 2(a). Entretanto alguns conjuntos com exemplos positivos e negativos não podem ser separados por um hiperplano. A Figura 2(b) mostra uma situação dessas − é impossível traçar uma reta que separe as instâncias positivas das negativas. Aqueles conjuntos cujas instâncias podem ser separadas são chamados de *conjuntos de treinamento linearmente separáveis* [Mitchell 1997].

Como comentado em [Mitchell 1997, pg. 87], "um único perceptron pode ser usado para representar muitas funções booleanas. Por exemplo, se associarmos o valor 1 a *true* e o valor −1 a *false*, uma maneira de usar um perceptron com duas entradas para implementar a função booleana AND é estabelecer os pesos $w_0 = -0.8$[1] e $w_1 = w_2 = 0.5$. Este perceptron pode também ser usado para representar a função booleana OR, bastando para isso estabelecer $w_0 = 0.3$. De fato, as funções AND e OR podem ser vistas como casos especiais de funções *m-de-n* ie, funções onde pelo menos *m* das *n* entradas para o perceptron devem ser *true*. A função OR corresponde a $m=1$ e a função AND a $m=n$. Qualquer função *m-de-n* é facilmente representada usando um perceptron; isso pode ser feito estabelecendo todos os pesos de entrada iguais (eg. 0.5) e, então, estabelecendo o valor w_0 de acordo. Perceptrons podem representar todas as funções booleanas primitivas ie, AND, OR, NAND (¬AND) e NOR (¬OR). Infelizmente algumas funções não podem ser representadas por um único perceptron, como é o caso da função XOR. A habilidade de perceptrons para representar AND, OR, NAND e NOR é importante porque toda função booleana pode ser representada por alguma rede de unidades interconectadas baseada nessas funções primitivas. De fato, toda função booleana pode ser representada por alguma rede de perceptrons com dois níveis de profundidade, na qual as entradas alimentam múltiplas unidades cujas saídas são entrada para a camada final. Devido ao fato que redes de perceptrons podem representar uma grande variedade de funções e também porque unidades sozinhas não podem, há grande interesse no aprendizado de redes multi-camadas de perceptrons".

[1] Nessa dissertação, o . (ponto) está sendo usado para separar a parte inteira da parte decimal na representação de números reais.

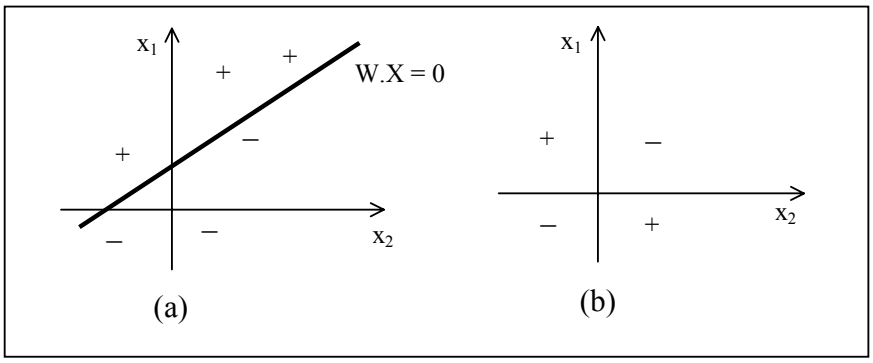


Figura 2. Superfície de decisão representada por um perceptron de duas entradas
(a) O conjunto de treinamento e a superfície de decisão do perceptron que os classifica corretamente.
(b) conjunto de treinamento que é não linearmente separável.
x_1 e x_2 são as entradas do perceptron. Exemplos positivos são indicados por '+' e negativos por '–'.

2.1 O TERMO *BIAS*

Como dito anteriormente, aprendizado baseado em perceptron faz uso de uma entrada adicional x_0 denominada *bias* que irá ajudar na determinação dos valores do vetor de pesos W. O uso dessa entrada adicional, que possui valor constante, melhora o desempenho do algoritmo perceptron e seus derivados. Isso ocorre porque com o uso de um *bias* diferente de zero, o poder de separação do algoritmo (ie, a capacidade para encontrar um hiperplano separador) é aumentado. O valor do *bias* igual a zero corresponde à busca de um hiperplano que necessariamente passa pela origem. Muitas vezes, com um hiperplano passando pela origem, não é possível obter a separação entre as duas classes de instâncias de treinamento. Na Figura 2(a), por exemplo, qualquer reta que passe pela origem não separa as duas classes de instâncias. A introdução do *bias* é feita via introdução de um componente $x_0=1$ a cada um dos exemplos de treinamento E_i, $1 \leq i \leq n$. O *bias* é tratado exatamente como qualquer outro peso (ver Figura 1). Considere, por exemplo, uma situação em que o conjunto de treinamento é dado por $E=\{E^1, E^2, E^3\}$ como mostra a Tabela 1.

Tabela 1. Conjunto de treinamento
$E=\{(E^1,C^1),(E^2,C^2),(E^3,C^3)\}$

Exemplo	x_1	x_2	x_3	Classe
E^1	–1	–1	–1	+1
E^2	+1	–1	–1	–1
E^3	+1	+1	+1	+1

Para a introdução do *bias*, o conjunto da Tabela 1 é transformado no conjunto mostrado na Tabela 2.

Tabela 2. Conjunto de treinamento
$E=\{(E^1,C^1),(E^2,C^2),(E^3,C^3)\}$ com o *bias*

Exemplo	x_0	x_1	x_2	x_3	Classe
E^1	+1	–1	–1	–1	+1
E^2	+1	+1	–1	–1	–1
E^3	+1	+1	+1	+1	+1

Com o objetivo de padronizar a notação utilizada bem como definir exatamente a simbologia empregada, o Anexo A introduz a fundamentação matemática que subsidia o estabelecimento dos resultados deste capítulo.

2.2 O ALGORITMO PERCEPTRON

O pseudocódigo do algoritmo perceptron está descrito na Figura 3 e é uma reescrita estruturada do algoritmo apresentado em [Gallant 1986]. A entrada para o algoritmo é um conjunto de treinamento cujas instâncias pertencem a duas classes distintas, presumivelmente linearmente separáveis.

```
procedure perceptron(E,W)
{Entradas: E conjunto de treinamento com n elementos da forma:
E^k = (x^k_0, x^k_1,x^k_2,…, x^k_p,C^k) representando n instâncias
pertencentes às classes C^k = {1,-1}                          }
begin
 W = (0,0,0,…,0){W é inicializado como um vetor nulo com p+1
                 posições}
 while not(tudo_certo(E,W)) do
 begin
  select (E,E^k)
  if (W.E^k > 0 and C^k == +1) or (W.E^k < 0 and C^k == –1)
   then {não é necessário fazer nada}
  else W = W + C^k E^k
 end
end

function tudo_certo(E,W)
begin
 todos_certos = true
 i = 1
 while todos_certos and i ≤ n do
  begin
    select(E,E^i)
    if (W.E^i > 0 and C^i == +1) or (W.E^i < 0 and C^i == –1)
     then i = i+1
    else todos_certos = false
  end
 return todos_certos
end
```

Figura 3. Pseudocódigo do algoritmo perceptron

É importante mencionar que a seleção do exemplo de treinamento E^k (feita pelo procedimento *select*(E,E^i) na Figura 3) pode ser feita de várias maneiras. As opções mais comuns são: escolha aleatória, cíclica ou numa ordem pré-determinada. Para tornar o pseudocódigo da function *tudo_certo*(E,W) facilmente legível, entretanto, optamos por fazer uma seleção sistemática seqüencial, dado que este procedimento necessita varrer todo o conjunto de treinamento, para garantir a condição de término do procedimento *perceptron*(E,W). Nos exemplos que seguem será usada a escolha aleatória. A modificação do vetor de pesos, como descrita na Figura 3, não é única. Detalhes sobre possíveis outras opções podem ser encontrados em [Parekh 1998, pg. 90].

2.2.1 Exemplo de Execução do Perceptron

O exemplo tratado nesta seção foi extraído de [Gallant 1994, pg. 65]. Dado o conjunto de treinamento descrito na Seção 2.1, Tabela 2, nesta seção é descrito em detalhes os passos do algoritmo perceptron na determinação do vetor de pesos W que define o hiperplano que separa as instâncias das duas classes.

Pode ser verificado na Tabela 3 que, após a execução de 10 passos, o vetor de pesos W não precisa mais ser modificado, pois para todo $E^k \in E$, $W.E^k$ e C^k têm o mesmo valor.

Tabela 3. Execução passo a passo do perceptron para dados da Tabela 2

Passo	W atual	Exemplo escolhido	Classificação correta do exemplo?	Ação
1	<0 0 0 0>	E^1	Não	$W \leftarrow W + E^1$
2	<1 –1 –1 –1>	E^2	Não	$W \leftarrow W – E^2$
3	<0 –2 0 0>	E^1	Sim	Nenhuma
4	<0 –2 0 0>	E^3	Não	$W \leftarrow W + E^3$
5	<1 –1 1 1>	E^1	Não	$W \leftarrow W + E^1$
6	<2 –2 0 0>	E^3	Não	$W \leftarrow W + E^3$
7	<3 –1 1 1>	E^1	Sim	Nenhuma
8	<3 –1 1 1>	E^2	Não	$W \leftarrow W – E^2$
9	<2 –2 2 2>	E^1	Não	$W \leftarrow W + E^1$
10	<3 –3 1 1>			

O algoritmo perceptron sempre encontra um hiperplano separador, se este existir. No entanto, como uma única TLU só consegue implementar um único hiperplano no espaço de entrada, o algoritmo é incapaz de classificar eficientemente conjuntos de entrada

que não sejam linearmente separáveis. Um conjunto de entrada definido pela função booleana XOR é um exemplo de conjunto não linearmente separável (como mostrado na Figura 2 (b)).

2.2.2 Resultados Teóricos que Governam o Perceptron

O algoritmo perceptron consegue encontrar, depois de um certo número de passos, um vetor de pesos W que classifica corretamente todas as instâncias de entrada, se estas forem linearmente separáveis. Assim sendo, o perceptron encontrará o vetor de pesos perfeito em tempo finito se as instâncias de entrada forem finitas. O conteúdo das próximas duas subseções foi adaptado de [Gallant 1994, pg. 70-72].

2.2.2.1 TEOREMA DA CONVERGÊNCIA DO PERCEPTRON

Para provar os teoremas a seguir são usados exemplos de treinamento com componentes E_j^k que são números reais arbitrários e então derivar limites para o caso especial onde $E_j^k = \{+1, -1, 0\}$.

Primeiramente, se o *bias* for 0 então poderemos converter o problema de se achar pesos para um conjunto de exemplos de treinamento com saídas podendo ser +1 ou −1 para um problema equivalente onde todas as saídas corretas serão +1. Para tal, basta substituir cada exemplo de treinamento E_j^k com $C^k = -1$ por um novo exemplo de treinamento $(-E_j^k)$ com $C^k = +1$.

Essa simplificação não melhora muito o aprendizado do algoritmo perceptron e por isso não é muito usada na prática. No entanto, ela simplifica as provas que seguem. Assim sendo, assumiremos que cada problema foi convertido em um outro problema equivalente com todos C = +1. É importante lembrar que W significa o vetor de pesos atual = $<W_0, W_1,...,W_p>$.

Teorema da Convergência do Perceptron. *Seja E um (possivelmente infinito) conjunto de exemplos de treinamento, com cada exemplo tendo tamanho máximo K. Se existir um vetor*

de pesos W^* e um número $\delta > 0$ tal que $W^*.E^k \geq \delta$ para todo $E^k \in E$ então o algoritmo perceptron irá executar o passo de mudança do vetor de pesos ($W = W + C^kE^k$) no máximo

$$\left(\frac{K\left\|W^*\right\|}{\delta}\right)^2 \textit{ vezes}$$

A prova desse teorema pode ser encontrada no Anexo B. Os dois próximos corolários são conseqüência direta do teorema de convergência do perceptron.

Corolário 1. *Para um conjunto de exemplos de treinamento separáveis, E, o algoritmo perceptron irá produzir um vetor de pesos W que satisfaz $W.E^k > 0$ para todo $E^k \in E$ depois de um número finito de passos de mudança no vetor de pesos.*

Corolário 2. *Seja E um conjunto de exemplos de treinamento finito e separável, onde cada exemplo tem p + 1 entradas $E_j^k \in \{+1, -1, 0\}$. O algoritmo perceptron irá produzir um vetor de pesos inteiro W^* depois de executar no máximo*

$$(p+1)\left\|W^*\right\|^2$$

passos de mudança. Se os exemplos forem escolhidos consecutivamente seguindo uma ordem fixa, então

$$(p+1)\,|E|\,\left\|W^*\right\|^2$$

iterações são necessárias para produzir tal solução.

2.2.2.2 TEOREMA CÍCLICO DO PERCEPTRON

Teorema Cíclico do Perceptron. *Dado um conjunto finito de exemplos de treinamento E, existe um número M tal que se o algoritmo perceptron for executado, a partir de qualquer vetor de pesos inicial W^0, então qualquer vetor de pesos W^t gerado pelo algoritmo irá satisfazer:*

$$\left\|W^t\right\| \leq \left\|W^0\right\| + M.$$

A prova é extensa e não será mostrada aqui. Para maiores detalhes, consultar [Block & Levin 1970].

Corolário 3. *Se um conjunto finito de exemplos de treinamento tiver componentes* E_j^k *inteiros ou racionais, o algoritmo perceptron irá visitar uma coleção finita de diferentes vetores* W^t.

Corolário 4. *Para um conjunto finito de exemplos de treinamento E, com componentes* E_j^k *inteiros ou racionais, o algoritmo perceptron irá, em tempo finito:*

1. Produzir um vetor de pesos que satisfaz todos os exemplos de treinamento (se e somente se E for linearmente separável), ou,

2. Sair e visitar novamente um vetor de pesos (se e somente se E for não linearmente separável).

Em conjuntos com instâncias não linearmente separáveis, o perceptron tem baixo rendimento. Na literatura podem ser encontradas as propostas de várias modificações deste algoritmo com o objetivo de achar um vetor de pesos eficiente que classifica corretamente uma grande porção do conjunto de treinamento S, quando S é não linearmente separável e, também, que não produza erro, quando S é linearmente separável. Algumas dessas modificações são novos algoritmos conhecidos como: algoritmo pocket e algoritmo pocket com modificação ratchet [Gallant 1986], algoritmo thermal perceptron [Frean 1990a] [Frean 1992], algoritmo loss minimization [Hrycej 1992] e barycentric correction procedure [Poulard 1995]. As Seções 2.3 e 2.4 detalham o pocket e o pocket com modificação ratchet respectivamente, uma vez que tais algoritmos são usados por alguns dos algoritmos construtivos investigados nesse trabalho.

2.3 O ALGORITMO POCKET

Como comentado anteriormente, embora o perceptron seja simples, de fácil implementação, rápido e poderoso, ele não classifica eficientemente conjuntos de entrada

não linearmente separáveis. Em conjuntos não linearmente separáveis, não existe um vetor de pesos capaz de classificar corretamente todas as instâncias de treinamento, pois não existe um hiperplano separador que divida o espaço de entrada em dois conjuntos (ou duas classes de saída).

A melhor maneira de classificar conjuntos de entrada não linearmente separável é encontrar um vetor de pesos que classifique corretamente o maior número possível de instâncias de entrada. Uma possível alternativa é usar o algoritmo pocket proposto em [Gallant 1986]. Esse algoritmo tem alta probabilidade de encontrar soluções ótimas se for executado por tempo suficiente. No entanto, dependendo da entrada em questão, os passos podem ser muitos e, desse modo, o algoritmo encontrará apenas soluções boas e/ou subótimas.

Como comentado em [Gallant 1994, pg. 76], "o problema do método de treinamento perceptron é que este algoritmo usa apenas reforço negativo ie, o perceptron apenas modifica o vetor de pesos W se a saída referente à entrada em questão 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 algoritmo pocket proposto por Gallant é um perceptron modificado, que leva em consideração reforço positivo. Este algoritmo faz uso de duas variáveis extras, uma que guarda um vetor de pesos W^{pocket} e outra que guarda o número de iterações consecutivas que W^{pocket} classificou corretamente os exemplos de entrada escolhidos. Se um outro vetor W conseguir classificar corretamente por mais passos que W^{pocket}, então W^{pocket} é atualizado como W. A Figura 4 descreve o pseudocódigo do algoritmo, que é uma reescrita estruturada do algoritmo proposto em [Gallant 1986]. No pseudocódigo a variável Run^{w} representa número de vezes que W classificou corretamente e $Run^{wpocket}$ representa o número de vezes que W^{pocket} classificou corretamente os exemplos.

A função *estime*, da Figura 4, determina o número de iterações que o algoritmo deve executar. Esse número deve ser suficientemente grande para garantir que o vetor de pesos W encontrado pelo algoritmo seja suficientemente bom. Infelizmente, não se conhece nenhum método para se achar esse número. Ver Seção 2.3.2 para maiores detalhes.

```
procedure pocket (E, W)
{Entradas: E conjunto de treinamento com n elementos da forma:
E^k = (x^k_0, x^k_1,x^k_2,…, x^k_p,C^k) representando n instâncias
pertencentes às classes C^k = {1,-1}                          }
begin
 W = (0,0,0,…,0)   { W é inicializado como um vetor nulo com
                                        p+1 posições}
 W^pocket = W
 Run^w = 0
 Run^wpocket = 0
 passos_dados = 0
 estime(passos_suficientes)
 while passos_dados < passos_suficientes do
 begin
  select (E,E^k)
  if (W.E^k > 0 and C^k == +1) or (W.E^k < 0 and C^k == −1) then
  begin
   Run^w = Run^w + 1
   if(Run^w > Run^wpocket) then
   begin
    W^pocket = W
    Run^wpocket = Run^w
   end
  end
  else
   begin
     W = W + C^k E^k
     Run^w = 0
   end
   passos_dados = passos_dados + 1
 end
end
```

Figura 4. Pseudocódigo do algoritmo pocket

2.3.1 Exemplo de Execução do Pocket

Para exemplificar o funcionamento do pocket será usado um conjunto de treinamento cujas instâncias definem a função booleana XOR que, como mostrado anteriormente (Figura 2(b)), são não linearmente separáveis. O conjunto estendido com a introdução do *bias* é mostrado na Tabela 4.

Tabela 4. Conjunto de treinamento XOR (*bias* incluído)

Exemplo	x_0	x_1	x_2	Classe
E^1	1	−1	−1	−1
E^2	1	−1	1	1
E^3	1	1	−1	1
E^4	1	1	1	−1

A Tabela 5 exibe o funcionamento do pocket, passo a passo, para o conjunto de entrada descrito na Tabela 4. Note que no passo 9 o vetor de pesos W é igual ao vetor nulo

(que é usado para começar o algoritmo). Esse vetor nulo classifica erradamente todas as instâncias de entradas E^k, mas o W^{pocket} do passo 9 classifica corretamente 3 das 4 instâncias de entrada.

Tabela 5. Execução do algoritmo pocket com os dados da Tabela 4

Passo	W	Run^w	W^{pocket}	$Run^{wpocket}$	Exemplo escolhido	Classifica corretamente?	Ação
1	<0 0 0>	0	<0 0 0>	0	E^4	Não	$W \leftarrow W - E^4$ $Run^w \leftarrow 0$
2	<–1 –1 –1>	0	<0 0 0>	0	E^4	Sim	$Run^w \leftarrow Run^w + 1$ $W^{pocket} \leftarrow W$ $Run^{wpocket} \leftarrow Run_w$
3	<–1 –1 –1>	1	<–1 –1 –1>	1	E^2	Não	$W \leftarrow W + E^2$ $Run^w \leftarrow 0$
4	<0 –2 0>	0	<–1 –1 –1>	1	E^3	Não	$W \leftarrow W + E^3$ $Run^w \leftarrow 0$
5	<1 –1 –1>	0	<–1 –1 –1>	1	E^4	Sim	$Run^w \leftarrow Run^w + 1$
6	<1 –1 –1>	1	<–1 –1 –1>	1	E^2	Sim	$Run^w \leftarrow Run^w + 1$ $W^{pocket} \leftarrow W$ $Run^{wpocket} \leftarrow Run^w$
7	<1 –1 –1>	2	<1 –1 –1>	2	E^3	Sim	$Run^w \leftarrow Run^w + 1$ $Run^{wpocket} \leftarrow Run^w$
8	<1 –1 –1>	3	<1 –1 –1>	3	E^1	Não	$W \leftarrow W - E^1$ $Run^w \leftarrow 0$
9	<0 0 0>	0	<1 –1 –1>	3	.....	.....	.....

2.3.2 Resultados Teóricos que Governam o Pocket

O pocket consegue encontrar a solução ótima do problema depois de um certo número de passos ou iterações. O teorema enunciado a seguir, baseado em [Gallant 1994], e sua prova garantem isso. No entanto, o Teorema de Convergência do Pocket não estabelece o número necessário de iterações que garantem pesos ótimos. Com isso, para problemas de médio ou grande porte, são necessários muitos passos para que o pocket encontre a solução ótima. Isso não se configura, entretanto, como um problema muito grave uma vez que são conhecidos apenas métodos exponenciais para se achar uma solução ótima de problemas não linearmente separáveis.

Teorema da Convergência do Pocket: *Dado um conjunto finito de instâncias de entrada E^k e suas correspondentes saídas esperadas C^k e uma probabilidade $p < 1$, existe um N tal que depois de $n \geq N$ iterações do algoritmo pocket, a probabilidade de que o vetor de pesos gerado seja ótimo supera a probabilidade p.*

A prova pode ser encontrada em [Gallant 1994]. É importante saber que, mesmo em problemas de médio porte, o número de iterações necessárias para se encontrar o vetor de pesos ótimo é, geralmente, demasiadamente grande. Assim, em aplicações práticas, é muito provável que o algoritmo encontre soluções apenas subótimas. No entanto, de acordo com [Gallant 1994] “as soluções subótimas geradas têm alta probabilidade de serem boas soluções e classificarem corretamente grande parte das instâncias que o vetor de pesos ótimo classificaria.”.

2.4 ALGORITMO POCKET COM MODIFICAÇÃO RATCHET

Apesar do pocket implementar mecanismos de reforço positivo e negativo, é incapaz de lidar com uma eventual situação de "azar". Um vetor de pesos ruim pode substituir um vetor de pesos bom caso o primeiro "tenha sorte", ie, caso as instâncias de entrada escolhidas consecutivamente sejam classificadas corretamente por esse vetor. Se as instâncias escolhidas e classificadas corretamente com o vetor W forem muitas, então esse vetor pode dominar o W^{pocket} impossibilitando que melhores vetores possam fornecer seus valores a este vetor de pesos. O contrário também pode acontecer, ie, se o melhor vetor tiver azar, ele pode não fornecer seus valores ao W^{pocket}. Uma modificação chamada modificação ratchet corrige esse problema. O pocket alterado com a introdução dessa modificação é conhecido como pocket com modificação ratchet e foi proposto em [Gallant 1986]. Neste algoritmo modificado, o vetor de pesos W^{pocket} sempre melhora, não existe a possibilidade de acontecer uma atualização que faça com que ele piore.

A lógica da modificação é bastante simples: antes do vetor W^{pocket} ser atualizado com os valores do vetor de pesos W, é verificado se W classifica corretamente mais exemplos de entrada do que W^{pocket}. Caso a resposta seja negativa, W^{pocket} não é atualizado com os valores de W. O custo computacional deste procedimento pode ser razoavelmente alto, caso o número de instâncias do conjunto de treinamento seja grande. Entretanto, à medida que o número de instâncias consecutivas que W^{pocket} classifica corretamente aumenta, o tempo antes das verificações que definem se W pode ou não fornecer valores a W^{pocket} aumenta exponencialmente e, portanto o tempo de tais verificações não influencia muito no tempo total. Por essa razão a modificação é útil e deve ser sempre usada na

prática. A Figura 5 descreve o pseudocódigo do algoritmo. É uma reescrita estruturada do algoritmo descrito em [Gallant 1986].

```
procedure pocket_ratchet (E,W)
{Entradas: E conjunto de treinamento com n elementos da forma:
  E^k = (x^k_0, x^k_1,x^k_2,…, x^k_p,C^k) representando n instâncias
  pertencentes às classes C^k = {1,-1}                          }
begin
 W = (0,0,0,…,0)   { W é inicializado como um vetor nulo com
                              p+1 posições}
 W^pocket = W
 Run^w = 0
 Run^wpocket = 0
 passos_dados = 0
 estime(passos_suficientes)
 while passos_dados < passos_suficientes do
  begin
   select (E,E^k)
   if (W.E^k > 0 and C^k == +1) or (W.E^k < 0 and C^k == −1) then
    begin
     Run^w = Run^w + 1
     if(Run^w > Run^wpocket and melhor(W, W^pocket)) then
      begin
       W^pocket = W
       Run^wpocket = Run^w
      end
    end
   else
    begin
     W = W + C^k E^k
    Run^w = 0
    end
   passos_dados = passos_dados + 1
  end
end

function melhor(W, W^pocket)
begin
 numeroW = 0
 numeroPocket = 0
 for_all E^i = (x^i_0, x^i_1,x^i_2,…, x^i_p,C^i) ∈ E do
 begin
  if (W.E^i > 0 and C^i == +1) or (W.E^i < 0 and C^i == −1)
   then numeroW = numeroW + 1
  if (W^pocket.E^i > 0 and C^i == +1) or (W^pocket.E^i < 0 and C^i == −1)
   then numeroPocket = numeroPocket + 1
 end
 if (numeroW > numeroPocket)
  then return true
  else return false
end
```

Figura 5. Pseudocódigo do algoritmo pocket com modificação ratchet

A referência [Palma Neto & Nicoletti 2003a] apresenta um estudo detalhado sobre o perceptron e seus derivados.

CAPÍTULO 3. TORNANDO O PERCEPTRON CONSTRUTIVO – ALGORITMOS TOWER E PYRAMID

Esse capítulo discute dois algoritmos, o tower e o pyramid, ambos propostos por Gallant em [Gallant 1986], que podem ser abordados como uma versão construtiva do algoritmo perceptron, na sua versão ratchet.

3.1 O ALGORITMO TOWER

O algoritmo tower foi proposto como um modelo de rede neural construtiva. Cada camada intermediária dinamicamente construída pelo algoritmo durante a fase de treinamento é composta por apenas um neurônio. O número de camadas intermediárias construídas é dependente da precisão desejada.

A idéia na qual o algoritmo se baseia é bastante simples. Inicialmente é usado o algoritmo pocket com modificação ratchet para treinar o primeiro neurônio lógico da rede. Se p é o número de atributos que descrevem os exemplos de treinamento, esse neurônio recebe p + 1 entradas, ou seja, o número de atributos mais o termo constante *bias*.

Como visto no Capítulo 2, um único neurônio consegue aprender com 100% de precisão apenas a partir de conjuntos de treinamento cujas instâncias são linearmente separáveis. Se não for esse o caso, o algoritmo tower continua adicionando neurônios à rede (um por camada) até que uma determinada precisão seja alcançada ou, então, um critério de parada seja satisfeito. O fato do único neurônio que define cada uma das camadas intermediárias ter como entradas valores dos neurônios de entrada e o valor do neurônio da

camada intermediária imediatamente anterior à sua, faz com que a visualização da rede se assemelhe a uma torre, daí seu nome.

Considerando que o passo 1 é o passo onde o primeiro neurônio é adicionado à rede, pode-se dizer que no k-ésimo passo o k-ésimo neurônio adicionado à rede recebe as p + 1 entradas da rede, mais uma entrada adicional que é a saída do (k–1)-ésimo neurônio. Com a adição dinâmica de neurônios, o algoritmo tende a classificar corretamente mais exemplos de treinamento.

O algoritmo tower termina quando o processo de construção da rede pela adição de neurônios intermediários não colabora mais na precisão de classificação no conjunto de treinamento ou, então, quando um critério de parada é satisfeito (por exemplo, quando um limite de número de camadas intermediárias é estabelecido). A Figura 6 mostra uma reescrita padronizada do pseudocódigo do algoritmo apresentado em [Palma Neto & Nicoletti 2003c] e a Figura 7 ilustra a arquitetura de uma "rede tower"[2], para um conjunto de treinamento com três atributos.

```
Inicialização de variáveis globais:
camada = 0 {representa o número de camadas intermediárias}
MAX = número máximo de camadas intermediárias da rede
Woutput[MAX] {um vetor de vetores de peso que é a saída do sistema}

procedure tower(E)
{Entradas: E conjunto de treinamento com n elementos da forma:
 E^k = (x^k_0, x^k_1, x^k_2, ..., x^k_p, C^k) representando n instâncias
 pertencentes às classes C^k = {1,-1}                    }
begin
 pocket_ratchet(E,W) {usa o algoritmo pocket com modificação ratchet
                      para gerar o vetor de pesos W}
 camada = camada + 1
 Woutput[camada] = W  {guarda o vetor de pesos da camada atual}
 modifica(E,E1,W)     {expande o conjunto de treinamento}
 pocket_ratchet(E1,W1)
 if (pior_igual(W,W1,E,E1) and (camada < MAX)) then
  tower(E1)
end

procedure modifica(E,E1,W)
{procedimento que modifica o valor da posição p+1 do vetor de pesos}
begin
 E1 = E
 for all E1^k ∈ E1 do
   if(E^k.W > 0)
     then E1^k_p+1 = 1
     else E1^k_p+1 = -1
end
```

Figura 6. Pseudocódigo do algoritmo tower

[2] Por "rede tower", estamos nos referenciando à uma rede neural criada pelo algoritmo tower.

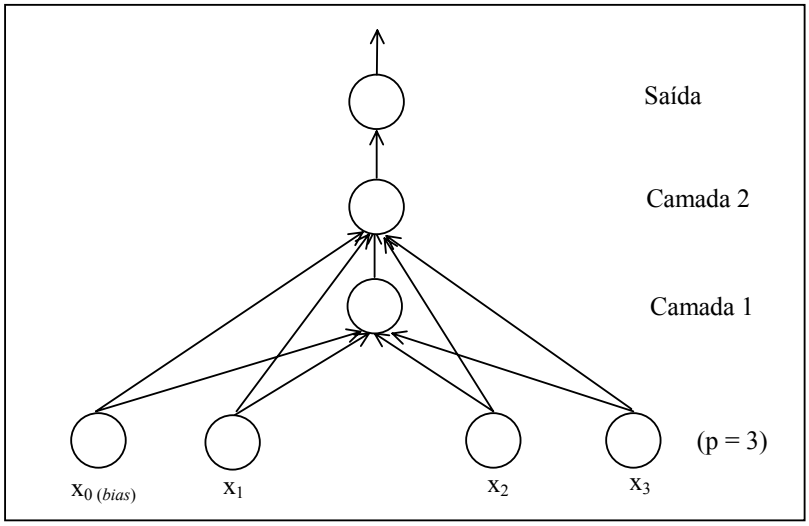


Figura 7. Arquitetura de uma rede construída pelo algoritmo tower

No pseudocódigo descrito na Figura 6, o procedimento *modifica*(E,E1,W) expande (no primeiro passo) ou modifica sem expandir (nos demais passos) o conjunto de treinamento E. No primeiro passo o procedimento transforma E num novo conjunto E1 por meio da introdução de uma nova dimensão; nos demais passos apenas atualiza a última dimensão, que representa o valor de saída do último neurônio adicionado à rede. E1 é usado como conjunto de entrada para a criação da próxima camada (de um neurônio) da rede.

Cada exemplo de treinamento do conjunto E1 é exatamente o mesmo de E apenas com uma dimensão a mais. Esta dimensão a mais adicionada no passo 1, quando é criado o primeiro nó intermediário, vai representar o comportamento do nó intermediário recém introduzido na rede. A cada passo, portanto, os valores que ocupam a posição p + 1 nas instâncias serão atualizados de maneira a representar as saídas do último nó intermediário criado até então.

Note que no primeiro passo do algoritmo, o conjunto de treinamento é o conjunto E, com dimensão p + 1 (p valores de atributos + 1 valor de *bias*). O primeiro vetor de pesos, então, tem dimensão p + 1. A partir desse ponto, é criada a primeira versão de E1, que é uma expansão de E, com uma dimensão extra, representando o comportamento do nó recém criado. Como o conjunto de treinamento passa a ter dimensão p + 2, os vetores de peso obtidos em cada passo seguinte terão dimensão p + 2.

O procedimento *pior_igual*(W,W1,E,E1) verifica se a precisão de classificação do vetor de pesos W no conjunto de treinamento E é igual ou menor do que a precisão de classificação do vetor de pesos W1 no conjunto de treinamento E1. Caso seja, o procedimento retorna *true* (verdade) e caso contrário retorna *false* (falso). Vale notar que se a precisão de classificação do vetor de pesos W no conjunto E for 100%, o procedimento retornará *false*, pois, com todos os exemplos classificados corretamente, não é necessário continuar com a construção de novas camadas.

Na proposta original do algoritmo tower [Gallant 1986], é usado um procedimento *pior* e não *pior_igual* como descrito na Figura 6. Assim sendo, para continuar com a construção da "torre", era necessário que o neurônio da camada N + 1 tivesse desempenho absolutamente melhor que o neurônio da camada N. Foi notado experimentalmente, no entanto, que em alguns casos, é necessário permitir que o algoritmo continue mesmo quando o neurônio da camada N + 1 tem desempenho igual ao neurônio da camada N, pois o desempenho pode melhorar posteriormente.

3.1.1 Exemplo de Funcionamento do Tower

Nesta subseção é detalhado o aprendizado do conceito de paridade-2, também conhecido como XOR, pelo algoritmo tower. Como comentado anteriormente, a função XOR representa um problema não linearmente separável e, portanto, os algoritmos perceptron e suas variações (incluindo o pocket com modificação ratchet) não conseguem aprender o vetor de pesos que classifica corretamente todos os exemplos de treinamento. Sendo assim, o algoritmo tower terá que criar, no mínimo, uma camada de neurônios intermediários (será necessário uma ou mais camadas além da primeira) para conseguir classificar corretamente todos os exemplos.

De acordo com [Gallant 1994], para o aprendizado do conceito de paridade-n usando o tower são suficientes $\lceil (n-1)/2 \rceil$[3] camadas intermediárias e, portanto, para o exemplo XOR, será necessária apenas uma camada intermediária. O conjunto de treinamento para o aprendizado do XOR é dado na Tabela 6; note que o termo x_0 que representa o *bias* já está incluído.

[3] $\lceil x \rceil$: menor inteiro maior ou igual a x.

Tabela 6. Conjunto de treinamento para o conceito de paridade-2, *bias* incluído

Exemplo	x_0	x_1	x_2	Classe
E^1	1	–1	–1	–1
E^2	1	–1	1	1
E^3	1	1	–1	1
E^4	1	1	1	–1

Primeiramente, é executado o algoritmo pocket com modificação ratchet que irá gerar o vetor de peso da primeira camada. A execução passo a passo desse algoritmo está descrita na Tabela 7. Run^w e $Run^{wpocket}$ significam o número de vezes que o vetor de pesos W e W^{pocket}, respectivamente, classificaram corretamente e consecutivamente as instâncias de treinamento.

Tabela 7. Criação da primeira camada da rede pelo tower. PMR significa pocket com modificação ratchet

Passos do PMR	W	Run^w	W^{pocket}	$Run^{wpocket}$	Exemplo escolhido	Classifica corretamente?	Ação
1	<0 0 0>	0	<0 0 0>	0	E^4	Não	$W \leftarrow W - E^4$ $Run^w \leftarrow 0$
2	<–1 –1 –1>	0	<0 0 0>	0	E^4	Sim	$Run^w \leftarrow Run^w + 1$ $W^{pocket} \leftarrow W$ $Run^{wpocket} \leftarrow Run_w$
3	<–1 –1 –1>	1	<–1 –1 –1>	1	E^2	Não	$W \leftarrow W + E^2$ $Run^w \leftarrow 0$
4	<0 –2 0>	0	<–1 –1 –1>	1	E^3	Não	$W \leftarrow W + E^3$ $Run^w \leftarrow 0$
5	<1 –1 –1>	0	<–1 –1 –1>	1	E^4	Sim	$Run^w \leftarrow Run^w + 1$
6	<1 –1 –1>	1	<–1 –1 –1>	1	E^2	Sim	$Run^w \leftarrow Run^w + 1$ $W^{pocket} \leftarrow W$ $Run^{wpocket} \leftarrow Run^w$
7	<1 –1 –1>	2	<1 –1 –1>	2	E^3	Sim	$Run^w \leftarrow Run^w + 1$ $Run^{wpocket} \leftarrow Run^w$
8	<1 –1 –1>	3	<1 –1 –1>	3	E^1	Não	$W \leftarrow W - E^1$ $Run^w \leftarrow 0$
9	<0 0 0>	0	<1 –1 –1>	3	.....	.....	.....

O vetor W^{pocket} encontrado na primeira camada (passo 1 do algoritmo tower) é <1 –1 –1>. Esse vetor classifica corretamente três das quatro instâncias de treinamento; apenas a instância E^1 não é classificada corretamente. Note que esse vetor de pesos é ótimo (ie, o algoritmo pocket com modificação ratchet não encontra nenhum outro vetor de pesos que classifica corretamente mais do que três instâncias de treinamento). Note também que esse vetor de pesos tem três dimensões.

A saída da primeira camada é usada como entrada para a segunda camada. A Tabela 8 define a saída da primeira camada, para cada uma das quatro instâncias de treinamento.

Tabela 8. Saída da primeira camada para cada instância de treinamento

Exemplo	$E.W^{pocket}$	Saída
E^1	3	1
E^2	1	1
E^3	1	1
E^4	–1	–1

Encontrado o vetor de pesos da primeira camada e constatado que esse vetor de pesos não classifica corretamente todas as instâncias de treinamento, o algoritmo adiciona um novo neurônio intermediário com o intuito de melhorar a classificação.

O vetor de pesos desse neurônio é determinado pelo algoritmo pocket com modificação ratchet, sendo que esse algoritmo agora tem que treinar uma rede com p + 2 entradas, ou seja, o novo neurônio intermediário possuirá p + 2 entradas (*bias* + p valores correspondentes à camada de entrada + saída da camada anterior). As p + 2 entradas para a segunda camada, ie, o novo conjunto de treinamento, sem as classes, criado pelo procedimento *modifica* podem ser visualizadas na Tabela 9.

Tabela 9. Entradas para a segunda camada (conjunto de treinamento expandido–E1)

Exemplo de E1	x_0 (*bias*)	x_1	x_2	x_3 (saída da camada anterior)
E^1	1	–1	–1	1
E^2	1	–1	1	1
E^3	1	1	–1	1
E^4	1	1	1	–1

A Tabela 10 mostra a execução do algoritmo pocket com modificação ratchet passo a passo, na segunda camada. Note que o vetor de pesos W = <–2 2 4 4> gerado classifica corretamente todas as instâncias de treinamento e, portanto, não é necessário adicionar mais nenhum neurônio intermediário. Note também que o vetor de pesos tem dimensão quatro, pois existem quatro entradas para a segunda camada (dois atributos + *bias* + saída da camada anterior). A Figura 8 mostra a "torre" gerada pelo algoritmo tower para o exemplo XOR.

Tabela 10. Criação da segunda camada da rede pelo tower

W	W^{pocket}	Exemplo escolhido	Classifica corretamente?	Ação
<0 0 0 0>	<0 0 0 0 >	E^2	Não	$W \leftarrow W + E^2$ $Run^w \leftarrow 0$
<1 –1 1 1>	<0 0 0 0>	E^3	Não	$W \leftarrow W + E^3$ $Run^w \leftarrow 0$
<2 0 0 2>	<0 0 0 0>	E^1	Não	$W \leftarrow W - E^1$ $Run^w \leftarrow 0$
<1 1 1 1>	<0 0 0 0>	E^4	Não	$W \leftarrow W - E^4$ $Run^w \leftarrow 0$
<0 0 0 2>	<0 0 0 0>	E^1	Não	$W \leftarrow W - E^1$ $Run^w \leftarrow 0$
<–1 1 1 1>	<0 0 0 0>	E^2	Não	$W \leftarrow W + E^2$ $Run^w \leftarrow 0$
<0 0 2 2>	<0 0 0 0>	E^3	Não	$W \leftarrow W + E^3$ $Run^w \leftarrow 0$
<1 1 1 3>	<0 0 0 0>	E^4	Não	$W \leftarrow W - E^4$ $Run^w \leftarrow 0$
<0 0 0 4>	<0 0 0 0>	E^1	Não	$W \leftarrow W - E^1$ $Run^w \leftarrow 0$
<–1 1 1 3>	<0 0 0 0>	E^1	Não	$W \leftarrow W - E^1$ $Run^w \leftarrow 0$
<–2 2 2 2>	<0 0 0 0>	E^4	Não	$W \leftarrow W - E^4$ $Run^w \leftarrow 0$
<–3 1 1 3>	<0 0 0 0>	E^3	Não	$W \leftarrow W + E^3$ $Run^w \leftarrow 0$
<–2 2 0 4>	<0 0 0 0>	E^2	Não	$W \leftarrow W + E^2$ $Run^w \leftarrow 0$
<–1 1 1 5 >	<0 0 0 0>	E^1	Não	$W \leftarrow W - E^1$ $Run^w \leftarrow 0$
<–2 2 2 4>	<0 0 0 0>	E^4	Sim	$Run^w \leftarrow Run^w + 1$ $W^{pocket} \leftarrow W$ $Run^{wpocket} \leftarrow Run^w$
<–2 2 2 4>	<–2 2 2 4>	.....	Sim	$Run^w \leftarrow Run^w + 1$ $W^{pocket} \leftarrow W$ $Run^{wpocket} \leftarrow Run^w$

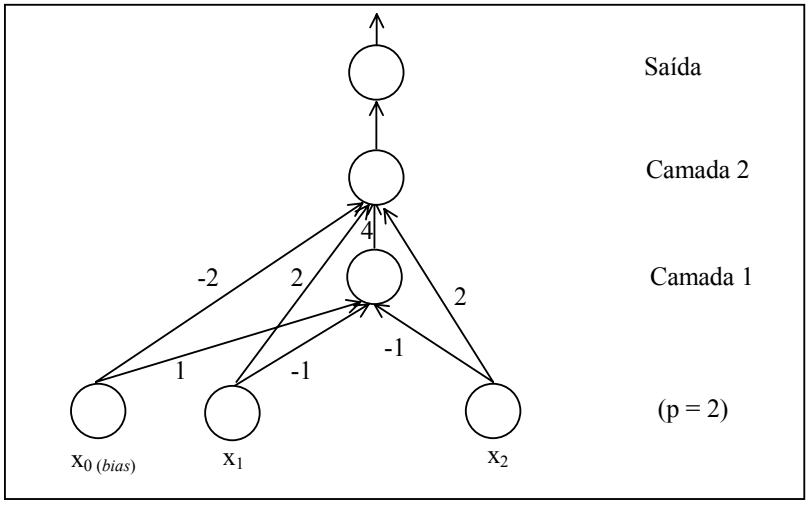


Figura 8. O conceito de paridade-2 expresso por uma rede tower

3.1.2 Prova de Convergência do Algoritmo Tower

O teorema enunciado a seguir garante que, verificadas determinadas condições, o algoritmo tower cria uma rede neural que representa o conceito, assumindo que o conjunto de treinamento que representa as instâncias do conceito seja discreto e com instâncias não contraditórias.

Teorema da convergência do Tower: *Com uma probabilidade arbitrariamente alta, o algoritmo tower irá representar, como uma rede neural tower, um conjunto de treinamento com instâncias não contraditórias, com valores de entrada restritos a {+1,–1}. Entretanto, é preciso que seja garantido que um número suficiente de neurônios tenha sido adicionado bem como um número suficiente de iterações tenha sido realizado. Além disso, cada neurônio adicionado irá classificar corretamente um número maior de exemplos de treinamento que qualquer neurônio anterior.*

Prova do teorema. O passo principal da prova consiste em estabelecer que uma rede com n camadas, ie, n neurônios, que classifica incorretamente um exemplo E^k pode ser transformado numa rede com n + 1 camadas que classificará corretamente E^k e manterá correta as outras classificações. Para maiores detalhes sobre a prova, consultar [Gallant 1994].

3.2 O ALGORITMO PYRAMID

O algoritmo pyramid, proposto em [Gallant 1986], é praticamente idêntico ao algoritmo tower discutido na Seção 3.1 exceto que, durante a criação da rede, cada novo neurônio intermediário introduzido na rede tem conexões com todos os outros neurônios intermediários criados até então e não apenas com o neurônio intermediário criado no passo anterior, como acontece com o tower.

Considerando que o passo um é o passo onde o primeiro neurônio é adicionado à rede, pode-se dizer que no k-ésimo passo o k-ésimo neurônio adicionado recebe, além das p + 1 entradas relativas aos p valores de atributos e o termo constante *bias*, k − 1 entradas, correspondentes às saídas de todos os neurônios intermediários já introduzidos na rede. Por exemplo, considerando que os exemplos de treinamento possuem três atributos, o neurônio intermediário introduzido no passo três terá seis entradas, correspondentes a três valores de

atributos de entrada, a um *bias,* à saída da camada 1 e à saída da camada 2. Assim como acontece no tower, a adição dinâmica de neurônios e a influência dos neurônios intermediários anteriores fazem com que o algoritmo tenda a classificar corretamente mais exemplos de treinamento. A Figura 9 descreve uma reescrita do pseudocódigo do algoritmo apresentado em [Palma Neto & Nicoletti 2003c]. A Figura 10 ilustra uma rede pyramid com três neurônios de entrada e o termo constante *bias*.

```
Inicialização de variáveis globais:
camada = 0 {representa o número de camadas intermediárias}
MAX = número máximo de camadas intermediárias da rede
Woutput[MAX] {um vetor de vetores de peso que é a saída do sistema}
procedure pyramid(E)
{Entradas: E conjunto de treinamento com n elementos da forma:
  E^k = (x^k_0, x^k_1,x^k_2,…, x^k_p,C^k) representando n instâncias
  pertencentes às classes C^k = {1,-1}                          }
begin
 pocket_ratchet(E,W) {use o algoritmo pocket com modificação ratchet
                      para gerar o vetor de pesos W}
 camada = camada + 1
 Woutput[camada] = W  {guarda o vetor de pesos da camada atual}
 modifica2(E,E1,W)    {expande o conjunto de treinamento}
 pocket_ratchet(E1,W1)
 if (pior_igual(W,W1,E,E1) and (camada < MAX)) then
  pyramid(E1)
end
procedure modifica2(E,E1,W)
begin
 i = camada - 1 + p + 1 {índice de entrada equivalente à saída do
                         último neurônio intermediário }
 E1 = E
 for all E1^k ∈ E1 do
   if(E^k.W > 0)
     then E1^k_i = 1
     else E1^k_i = -1
end
```

Figura 9. Pseudocódigo do algoritmo pyramid

O procedimento *modifica2*(E,E1,W) expande o conjunto E num novo conjunto E1 por meio da introdução de uma nova dimensão, que representa o valor do último neurônio adicionado à rede. E1 é então usado como conjunto de entrada na próxima camada. A cada passo, os exemplos de treinamento do conjunto E1 são expandidos em uma dimensão a mais.

O procedimento *modifica* do algoritmo tower difere ligeiramente do procedimento *modifica2* do algoritmo pyramid. O procedimento *modifica* adiciona uma dimensão a mais ao conjunto de treinamento apenas no primeiro passo (ie, quando é adicionado o primeiro neurônio intermediário à rede). A partir de então, a cada adição de um novo neurônio

intermediário, o *modifica* apenas atualiza o valor dessa última dimensão introduzida, de maneira que reflita o comportamento do neurônio em questão, dado que será entrada para o próximo neurônio a ser introduzido.

Já o *modifica2* adiciona uma dimensão a mais em todos os passos (ie, sempre que for adicionado um neurônio intermediário à rede). No algoritmo pyramid, um neurônio intermediário recebe conexões de todos os outros neurônios intermediários anteriores e, portanto, é necessário adicionar uma nova dimensão em cada passo, para que cada dimensão represente um neurônio intermediário anterior.

O procedimento *pior_igual*(W,W1,E,E1) usado no pseudocódigo da Figura 9 funciona como o procedimento de mesmo nome que é parte do algoritmo tower (ver Seção 3.1). Novamente, é usado o procedimento *pior_igual* ao invés do procedimento *pior* pelo mesmo motivo descrito anteriormente.

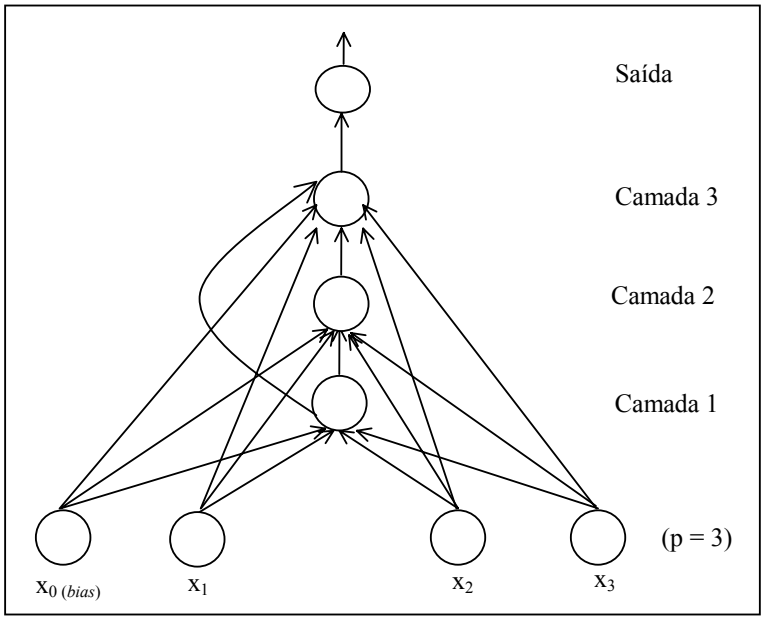


Figura 10. Arquitetura de uma rede construída pelo algoritmo pyramid

De acordo com [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 a tower e pyramid”.

3.2.1 Exemplo de Funcionamento do Pyramid

Nesta subseção é mostrado como o conceito de paridade-5 par é aprendido por uma rede pyramid. O conjunto de treinamento é descrito na Tabela 11; note que o termo constante que representa o *bias* (que não entra no cálculo da paridade), x_0, já está incluído no conjunto. Note também que devido ao fato de estarmos trabalhando com redes bipolares (saídas +1 ou –1), as saídas cujos valores deveriam ser zero (para contemplar o conceito de paridade par), são substituídas por –1.

Tabela 11. Conjunto de treinamento para o conceito de paridade-5, *bias* incluído

Exemplo	x_0	x_1	x_2	x_3	x_4	x_5	Saída
E^1	1	0	1	0	1	1	1
E^2	1	1	1	1	1	1	1
E^3	1	1	0	0	0	0	1
E^4	1	0	0	0	1	0	1
E^5	1	0	1	1	1	0	1
E^6	1	1	1	1	0	0	1
E^7	1	1	0	1	0	1	1
E^8	1	0	1	1	0	1	1
E^9	1	0	0	1	0	0	1
E^{10}	1	1	0	0	1	1	1
E^{11}	1	0	1	0	0	0	1
E^{12}	1	1	0	1	1	0	1
E^{13}	1	0	0	1	1	1	1
E^{14}	1	1	1	0	0	1	1
E^{15}	1	0	0	0	0	1	1
E^{16}	1	1	1	0	1	0	1
E^{17}	1	0	1	1	0	0	–1
E^{18}	1	1	1	1	1	0	–1
E^{19}	1	1	0	1	1	1	–1
E^{20}	1	0	0	1	0	1	–1
E^{21}	1	1	0	0	0	1	–1
E^{22}	1	1	0	0	1	0	–1
E^{23}	1	0	0	0	0	0	–1
E^{24}	1	1	1	0	0	0	–1
E^{25}	1	0	0	1	1	0	–1
E^{26}	1	0	1	0	0	1	–1
E^{27}	1	1	1	0	1	1	–1
E^{28}	1	0	0	0	1	1	–1
E^{29}	1	0	1	0	1	0	–1
E^{30}	1	1	1	1	0	1	–1

E^{31}	1	0	1	1	1	1	–1
E^{32}	1	1	0	1	0	0	–1

O primeiro passo do algoritmo pyramid é executar o algoritmo pocket com modificação ratchet parar gerar o neurônio da primeira camada que, na prática, é um vetor de pesos. Esse vetor de pesos terá seis posições (uma relativa ao *bias* e cinco relativas aos valores dos cinco atributos de entrada). Usando o sistema CONEB[4] (ver Anexo D) com máximo de mil iterações por camada, o vetor de pesos encontrado é $W_1 = <1\ 1\ 1\ 1\ 1\ 1>$.

Esse vetor de pesos classifica corretamente apenas 50% dos exemplos de entrada e, portanto, é necessário treinar um segundo neurônio com o intuito de melhorar a classificação. Para tal, é preciso expandir o exemplo de entrada E, transformando-o em E1 por meio da adição de uma nova dimensão a cada exemplo de E, que representa o neurônio já criado. Como o valor $E^k.W_1$ ($1 \leq k \leq 32$) é sempre positivo, a nova dimensão terá valor 1 para todos os exemplos de entrada.

O processo de treinamento de vetor de pesos se repete, ou seja, o algoritmo pocket com modificação ratchet é executado novamente usando agora o conjunto expandido E1. Note que esse conjunto é o conjunto descrito Tabela 11 ao qual foi acrescentado mais um atributo (x_6) que tem valor 1 para todos os exemplos. O vetor de pesos encontrado na execução do sistema CONEB foi $W_2 = <–1\ –2\ –2\ –1\ –1\ 1\ –1>$. Esse vetor de pesos classifica corretamente 50% dos exemplos de entrada.

Continuando a execução do sistema CONEB foi encontrado $W_3 = <–1\ 3\ 3\ –2\ 3\ –2\ –1\ 1>$ com 59.3% de precisão em E2. $W_4 = <1\ –4\ –5\ 4\ –5\ 5\ 1\ –1\ 5>$ com precisão de 84.3% em E3 e finalmente, $W_5 = <0\ 4\ 3\ –5\ 5\ –6\ 0\ 0\ –4\ 6>$ com 100% de precisão em E4. A Tabela 12 mostra esses valores para cada camada e a Tabela 13 mostra o conjunto de entrada E4. Note que na Tabela 13 é possível visualizar facilmente todos os conjuntos de entrada. Por exemplo, para visualizar E3 basta ignorar a coluna x_9, para visualizar E2 basta ignorar as colunas x_8 e x_9 e assim por diante.

[4] O sistema CONEB (Constructive Neural Builder) foi desenvolvido para viabilizar a experimentação com os diversos algoritmos neurais construtivos.

Tabela 12. Conjunto de entrada, vetor de pesos e precisão para cada camada do pyramid

Camada	Conjunto de entrada usado para treinar W	Vetor de pesos W	Precisão do vetor de pesos W
1	E	<1 1 1 1 1 1>	50%
2	E1	<–1 –2 –2 –1 –1 1 –1>	50%
3	E2	<–1 3 3 –2 3 –2 –1 1>	59.3%
4	E3	<1 –4 –5 4 –5 5 1 –1 5>	84.3%
5	E4	<0 4 3 –5 5 –6 0 0 –4 6>	100%

Tabela 13. Conjunto de entrada E4

Exemplo	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	Saída
E^1	1	0	1	0	1	1	1	–1	1	1	1
E^2	1	1	1	1	1	1	1	–1	1	1	1
E^3	1	1	0	0	0	0	1	–1	–1	–1	1
E^4	1	0	0	0	1	0	1	–1	–1	–1	1
E^5	1	0	1	1	1	0	1	–1	1	1	1
E^6	1	1	1	1	0	0	1	–1	1	1	1
E^7	1	1	0	1	0	1	1	–1	–1	1	1
E^8	1	0	1	1	0	1	1	–1	–1	1	1
E^9	1	0	0	1	0	0	1	–1	–1	1	1
E^{10}	1	1	0	0	1	1	1	–1	1	1	1
E^{11}	1	0	1	0	0	0	1	–1	–1	–1	1
E^{12}	1	1	0	1	1	0	1	–1	1	1	1
E^{13}	1	0	0	1	1	1	1	–1	–1	1	1
E^{14}	1	1	1	0	0	1	1	–1	1	1	1
E^{15}	1	0	0	0	0	1	1	–1	–1	1	1
E^{16}	1	1	1	0	1	0	1	–1	1	–1	1
E^{17}	1	0	1	1	0	0	1	–1	–1	–1	–1
E^{18}	1	1	1	1	1	0	1	–1	1	–1	–1
E^{19}	1	1	0	1	1	1	1	–1	–1	–1	–1
E^{20}	1	0	0	1	0	1	1	–1	–1	1	–1
E^{21}	1	1	0	0	0	1	1	–1	–1	–1	–1
E^{22}	1	1	0	0	1	0	1	–1	1	–1	–1
E^{23}	1	0	0	0	0	0	1	–1	–1	–1	–1
E^{24}	1	1	1	0	0	0	1	–1	1	–1	–1
E^{25}	1	0	0	1	1	0	1	–1	–1	–1	–1
E^{26}	1	0	1	0	0	1	1	–1	–1	–1	–1
E^{27}	1	1	1	0	1	1	1	–1	1	–1	–1
E^{28}	1	0	0	0	1	1	1	–1	–1	–1	–1
E^{29}	1	0	1	0	1	0	1	–1	1	–1	–1
E^{30}	1	1	1	1	0	1	1	–1	–1	–1	–1
E^{31}	1	0	1	1	1	1	1	–1	–1	–1	–1
E^{32}	1	1	0	1	0	0	1	–1	–1	–1	–1

3.2.2 Prova de Convergência do Algoritmo Pyramid

A prova de convergência do algoritmo pyramid é bastante parecida com a prova do algoritmo tower (ver Seção 3.1.2). Para maiores detalhes sobre a prova, consultar [Gallant 1994].

CAPÍTULO 4. O ALGORITMO TILING

O algoritmo tiling proposto por Marc Mézard e Jean Pierre Nadal em [Mézard & Nadal 1989] constrói uma rede neural com várias camadas e usa as noções de neurônio mestre e neurônios auxiliares, tratadas mais adiante. Cada camada, exceto a última, contém um único *neurônio mestre* e alguns *neurônios auxiliares*. A última camada contém apenas o neurônio mestre. Numa "rede tiling"[5], a camada k recebe conexões de todos os neurônios da camada k – 1, ie, do neurônio mestre e dos neurônios auxiliares que compõem a camada.

O algoritmo tiling garante que o neurônio mestre na camada k + 1 classifica com mais precisão os exemplos de treinamento, quando comparado com o neurônio mestre da camada k. Assumindo um conjunto de treinamento com instâncias não contraditórias, o neurônio mestre da última camada de uma rede tiling classifica corretamente todos os exemplos de treinamento. A Figura 11 mostra a arquitetura de uma rede tiling.

Como comentado em [Gallant 1994], "os neurônios auxiliares têm como objetivo aumentar o número de neurônios em cada camada L (menos a última) de modo que nenhum par de exemplos de treinamento com classes diferentes tenha a mesma saída em cada camada L." Assim sendo, a representação das entradas será particular a cada uma das camadas da rede e nenhum par de exemplos de treinamento, pertencentes a classes diferentes, terá as mesmas saídas, em qualquer uma das camadas. Camadas com essa propriedade são ditas *confiáveis* e uma condição necessária para que a rede classifique corretamente todos os exemplos de treinamento é que as camadas sejam confiáveis.

Assim como acontecia no tower e pyramid, o primeiro passo do tiling é gerar um vetor de pesos usando o pocket com modificação ratchet. Esse vetor será o *neurônio mestre* da primeira camada intermediária. Nessa camada, as entradas para o neurônio mestre são os p atributos de entrada e o termo constante *bias*, exatamente como acontece com o primeiro neurônio intermediário dos algoritmos tower e pyramid. Se esse vetor não classificar

corretamente todos os exemplos de treinamento, então será necessário adicionar *neurônios auxiliares* à primeira camada com o intuito de tornar essa camada confiável, ie, garantir que nenhum par de exemplos de treinamento com classes diferentes multiplicados pelos vetores de pesos gerados dos neurônios mestres e auxiliares tenham a mesma saída.

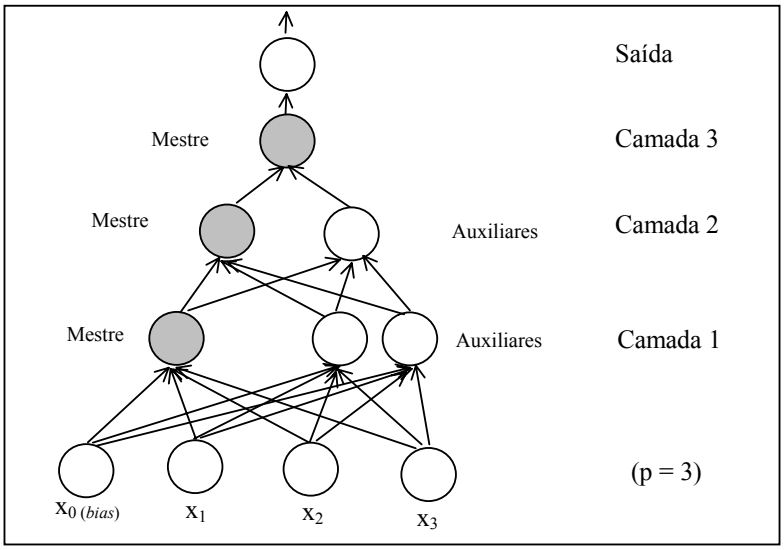


Figura 11. Arquitetura da rede construída pelo algoritmo tiling

Uma possível maneira de adicionar neurônios auxiliares com o objetivo de tornar a camada confiável é dividir o conjunto de entrada em dois subconjuntos e gerar vetores de pesos, ie, neurônios auxiliares, para cada um dos subconjuntos. Se esses neurônios não forem suficientes para tornar a camada confiável, então cada subconjunto é dividido em dois e o processo continua até que a camada se torne confiável. O critério para dividir os conjuntos em dois é simples: em um dos conjuntos ficam todos os exemplos de entrada com saída positiva (a multiplicação do vetor de pesos do conjunto original pelo exemplo de entrada dá um valor positivo) e no outro conjunto ficam todos os exemplos de entrada com saída negativa ou nula (a multiplicação do vetor de pesos do conjunto original pelo exemplo de entrada dá um valor negativo ou nulo). A partir da segunda camada, os neurônios mestre e auxiliares recebem conexões de todos os neurônios da camada anterior, como dito anteriormente. Os vetores de pesos desses neurônios terão, então, dimensão p + 2 + n_aux, onde p é o número de atributos, n_aux é o número de neurônios auxiliares gerados na camada anterior e o "2" representa o termo constante *bias* somado à entrada relativa ao

[5] Por "rede tiling", estamos nos referenciando à uma rede criada pelo algoritmo tiling.

neurônio mestre da camada anterior. A Figura 12 fornece o pseudocódigo do algoritmo, numa versão adaptada à padronização desse trabalho [ver Palma Neto & Nicoletti 2003c].

```
Inicialização de variáveis globais:
camada = 0 {representa o número de camadas intermediárias}
MAX_CAM = número máximo de camadas intermediárias da rede
MAX_AUX = número máximo de vetores auxiliares por camada
Woutput[MAX_CAM] {um vetor de vetores de peso que é a saída do sistema}
Waux[MAX_CAM][MAX_AUX] {vetor de vetores de peso que guarda os
                        valores dos neurônios auxiliares em
                        cada camada. Também é parte da saída do sistema}
procedure tiling(E)
{Entradas: E conjunto de treinamento com n elementos da forma:
  E^k = (x^k_0, x^k_1,x^k_2,…, x^k_p,C^k) representando n instâncias  pertencentes
  às classes C^k = {1,-1}                                                  }
begin
 pocket_ratchet(E,W) { gera o vetor de pesos W }
 camada = camada + 1
 Woutput[camada] = W       {guarda o vetor de pesos da camada atual}
 modifica3(E,E1,W1)        {expande o conjunto de treinamento}
 pocket_ratchet(E1,W1)
 if (pior_igual(W,W1,E,E1) and (camada < MAX_CAM)) then tiling(E1)
end
procedure modifica3(E,E1,W)
begin
 C = camada
 E1 = E
 for all E1^k ∈ E1 do
   if(E^k.W > 0) then E1^k_p+1 = 1
                 else E1^k_p+1 = -1
{Coloca no vetor Waux todos os neurônios auxiliares da camada C}
 Waux[C][MAX_AUX] = acha_todos_auxiliares(E,W,0)
for all Waux^i ∈ Waux[C]
   for all E1^k ∈ E1 do
      if(E^k.Waux^i > 0) then E1^k_p+1+i = 1
                         else E1^k_p+1+i = -1
end
procedure acha_todos_auxiliares(E,W,posicao)
begin
 C = camada
 if((precisao(E,W) ≠ 100%) and (posicao < MAX_CAM/2)) then
  begin
   acha_positivos(E,W,E_pos) ;  acha_negativos(E,W,E_neg)
   pocket_ratchet(E_pos,Wpos);  pocket_ratchet(E_neg,Wneg)
   if(precisao(E_pos,W) ≠ 100%) then
    begin
     Waux[C][] = Wpos {adiciona um neurônio auxiliar a Waux}
     acha_todos_auxiliares(E_pos,Wpos,posicao+1)
    end
   if(precisao(E_neg,W) ≠ 100%) then
    begin
     Waux[C][] = Wneg {adiciona um neurônio auxiliar a Waux}
     acha_todos_auxiliares(E_neg,Wneg,posicao+1)
    end
  end
end
```

Figura 12. Pseudocódigo do algoritmo tiling

O procedimento *modifica3*(E,E1,W) expande o conjunto E num novo conjunto E1 por meio da introdução de uma ou mais dimensões, que representam o valor do último

neurônio mestre adicionado à rede e os valores dos neurônios auxiliares da camada em questão. O número de dimensões de E1 depende do número de neurônios auxiliares gerados; quanto maior for o número de neurônios auxiliares necessários para tornar a camada em questão confiável, maior o número de dimensões que descrevem as instâncias do conjunto E1. Assim como no tower e pyramid, E1 será usado como conjunto de entrada na próxima camada. Note que as p + 1 (p atributos + o termo constante *bias*) primeiras dimensões permanecerão constantes durante o processo de construção da rede tiling.

O procedimento *modifica3* tem duas partes. Na primeira, adiciona (no primeiro passo) ou atualiza (nos passos seguintes) uma única dimensão relativa ao neurônio mestre da camada em questão, exatamente como acontece no algoritmo tower (considerando que o neurônio mestre do tiling é equivalente ao único neurônio intermediário por camada, criado pelo tower). Na segunda parte, outras dimensões são adicionadas. O número de dimensões adicionadas depende do número de neurônios auxiliares necessários para tornar a camada em questão confiável.

Por exemplo, supondo que os exemplos de treinamento possuem dois atributos, se foi necessário adicionar três neurônios auxiliares para que uma camada se tornasse confiável, então o *modifica3* daquela camada irá criar E1 com sete dimensões (2 atributos + *bias* + dimensão relativa ao neurônio mestre + 3 dimensões relativas aos neurônios auxiliares).

O procedimento *acha_todos_auxiliares*(E,W,posicao) cria todos os neurônios auxiliares necessários, relativos à camada em questão, de maneira a torná-la confiável. Para tal, o procedimento divide o conjunto de entrada em dois subconjuntos e gera vetores de pesos, ie, neurônios auxiliares, para cada um desses subconjuntos. Se esses neurônios não forem suficientes para tornar a camada confiável, ie, se o vetor de pesos desses neurônios não têm 100% de precisão de classificação nos seus subconjuntos (existe algum par de exemplos com classes diferentes, mas com a mesma saída), então cada subconjunto é dividido em dois e o processo continua recursivamente até que a camada se torne confiável. O critério para a divisão do conjunto em dois é bastante simples: em um conjunto ficam todos os exemplos de entrada com saída positiva (a multiplicação do vetor de pesos W pelo exemplo de entrada dá um valor positivo) e no outro conjunto ficam todos os exemplos de

entrada com saída negativa ou nula (a multiplicação do vetor de pesos W pelo exemplo de entrada dá um valor negativo ou nulo).

O procedimento *acha_positivos*(E,W,E_pos) coloca todos os exemplos de entrada E^k que possuem saída positiva ($E^k.W > 0$) em E_pos e *acha_negativos*(E,W,E_pos) coloca todos os exemplos de entrada E^k que possuem saída negativa ou nula ($E^k.W \leq 0$) em E_neg.

O procedimento *pior_igual*(W,W1,E,E1) funciona como o procedimento de mesmo nome do algoritmo tower (ver Capítulo 3, Seção 3.1). Na proposta original do algoritmo é usado um procedimento *pior* ao invés de *pior_igual*. Optamos, no entanto, por usar o procedimento *pior_igual* pela mesma razão discutida anteriormente.

4.1 EXEMPLO DE FUNCIONAMENTO DO TILING

Para exemplificar o funcionamento do algoritmo tiling, é usado o conjunto de entrada que define a função XOR, como descrito na Tabela 6 (*bias* já incluído). Note que esse conjunto é não linearmente separável e, portanto, será necessário adicionar neurônios auxiliares para fazer com que a primeira camada seja confiável.

O primeiro passo do algoritmo tiling é gerar o vetor de pesos principal, ie, o neurônio mestre, da primeira camada. Executando o algoritmo pocket com modificação ratchet que é usado pelo algoritmo tiling, tendo como entrada os dados da Tabela 6, uma possível solução seria: $W^{mestre} = <-1\ 1\ -1>$. Esse vetor de pesos classifica corretamente três dos quatro exemplos de entrada; apenas o exemplo E^2 é classificado incorretamente.

O neurônio mestre da primeira camada não tem 100% de precisão de classificação no conjunto de entrada e, portanto, a primeira camada não satisfaz o critério de confiabilidade – existe algum par de exemplos de treinamento que possuem classes distintas, mas com mesma saída. É fácil observar que existem dois de tais pares: $E^1 - E^2$ e $E^4 - E^2$. A Tabela 14 ilustra a configuração das classes e saídas para a primeira camada, antes de serem introduzidos os neurônios auxiliares.

Tabela 14. Configuração da primeira camada antes da introdução dos neurônios auxiliares

Exemplo	x_0	x_1	x_2	Classe	$W^{mestre}.E^k$	Saída[1]
E^1	1	–1	–1	–1	–1	–1
E^2	1	–1	1	1	–3	–1
E^3	1	1	–1	1	1	1
E^4	1	1	1	–1	–1	–1

Como dito anteriormente, com essa configuração, a primeira camada não satisfaz o critério de confiabilidade. Para torná-la confiável, é preciso adicionar um ou mais neurônios auxiliares. Para tal, é necessário dividir o conjunto de entrada em dois novos subconjuntos e gerar vetores de pesos para esses dois novos subconjuntos. O primeiro subconjunto (referente às saídas positivas) terá apenas E^3 e o segundo subconjunto (referente às saídas nulas ou negativas) terá E^1, E^2 e E^4. Como o primeiro subconjunto tem 100% de precisão de classificação com W^{mestre}, não é necessário gerar um vetor de pesos auxiliar para esse subconjunto, uma vez que ele não fere a confiabilidade da camada. O segundo subconjunto não tem 100% de precisão de classificação e, portanto, é necessário gerar um vetor de pesos para este subconjunto. Executando o algoritmo pocket com modificação ratchet do sistema CONEB (ver Anexo D), uma possível solução seria W^{aux} = <–1 –1 1>. Esse vetor de pesos auxiliar classifica corretamente todos os exemplos do segundo subconjunto e, portanto, restaura a confiabilidade na primeira camada e o processo de adição de neurônios auxiliares pára. A Tabela 15 ilustra a configuração da primeira camada depois de serem introduzidos os neurônios auxiliares.

Tabela 15. Configuração da primeira camada depois da introdução dos neurônios auxiliares

Exemplo	Classe	$W^{mestre}.E^k$	Saída^1	$W^{aux}.E^k$	Saída^2
E^1	–1	–1	–1	–1	–1
E^2	1	–3	–1	1	1
E^3	1	1	1	–3	–1
E^4	–1	–1	–1	–1	–1

Com a configuração mostrada na Tabela 15, a primeira camada passa a ter confiabilidade, pois não existe qualquer par de exemplos com classes distintas que produza a mesma saída.

Resolvido o problema de confiabilidade na primeira camada, o próximo passo do algoritmo é criar uma nova entrada para a segunda camada. Para tal, usa o procedimento *modifica3* descrito no pseudocódigo da Figura 12. A nova entrada criada pelo *modifica3* nada mais é do que a entrada original (descrita na Tabela 6) mais as colunas Saída^1 e Saída^2 da Tabela 15 ie, os dois atributos de entrada, o termo *bias*, a saída relativa ao neurônio mestre da camada anterior e a saída relativa ao neurônio auxiliar da camada anterior. A

Tabela 16 mostra a nova entrada criada para a segunda camada. Note que as colunas x_3 e x_4 são equivalente às colunas Saída[1] e Saída[2], respectivamente.

Tabela 16. Conjunto de entrada para a segunda camada

Exemplo	x_0(*bias*)	x_1	x_2	x_3	x_4	Classe
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

O próximo passo do tiling é gerar o neurônio mestre para a segunda camada. Para tal, executa o algoritmo pocket com modificação ratchet no conjunto de entrada descrito pela Tabela 16. Como esse conjunto é linearmente separável, o algoritmo pocket com modificação ratchet encontra facilmente um vetor de pesos que classifica corretamente todos os exemplos de entrada. Como esse vetor de pesos é perfeito, não é mais necessário adicionar neurônios auxiliares ou mais camadas e o algoritmo termina. O vetor de pesos encontrado na execução do sistema CONEB na segunda camada foi: W^{mestre} = <2 0 0 2 2>.

4.2 PROVA DE CONVERGÊNCIA DO ALGORITMO TILING

A prova de convergência do algoritmo tiling é bastante parecida com a prova de convergência do algoritmo tower (ver Seção 3.1.2). Para maiores detalhes sobre a prova, consultar [Mézard & Nadal 1989].

4.3 O ALGORITMO POINTING – UMA VARIAÇÃO DO ALGORITMO TILING

O autor Jean Pierre Nadal, um dos criadores do algoritmo tiling, propõe em [Nadal 1989] um algoritmo neural construtivo que chama de Pointing. Na mesma referência, comenta que o algoritmo Pointing é bastante similar ao tiling, podendo ser considerado uma sua variação. O Pointing, entretanto, é idêntico ao algoritmo tower proposto primeiramente em [Gallant 1986] e abordado no Capítulo 3. De acordo com [Gallant 1994], os autores Nadal

em [Nadal 1989], Gallant e Frean em [Frean 1990b] criaram o algoritmo tower independentemente.

O autor Śmieja descreve e analisa vários algoritmos neurais construtivos em [Śmieja 1991] e, entre eles, o algoritmo tiling e a variação Pointing proposto por Nadal. Em momento algum, entretanto, evidencia o fato dos algoritmos Pointing e tower serem o mesmo algoritmo, propostos independentemente com diferentes nomes.

CAPÍTULO 5. O ALGORITMO UPSTART

O upstart [Frean 1990a] é um algoritmo construtivo que dinamicamente cria uma rede neural de neurônios do tipo perceptron, cuja estrutura é a de uma árvore binária.

Como os demais, o upstart constrói a rede à medida que o aprendizado progride; entretanto, a maneira de construir a rede é diferenciada das demais investigadas até agora. Ao invés de construir camadas, a partir da camada de entrada, até atingir a convergência, o algoritmo vai adicionando camadas intermediárias entre a camada de entrada e de saída, no sentido saída-entrada.

5.1 CONSIDERAÇÕES SOBRE O ALGORITMO

No que segue o algoritmo apresentado em [Frean 1990a] para saídas +1 e 0 foi adaptado para saídas +1 e −1, com o objetivo de manter a padronização adotada quando da apresentação/discussão dos outros métodos construtivos tratados nesse trabalho.

A idéia básica do algoritmo upstart é a de criar neurônios ou nós filhos/auxiliares com o objetivo de corrigir os erros de classificação feitos pelo neurônio pai. Se um neurônio qualquer não consegue classificar corretamente os exemplos de treinamento, um ou dois neurônios filhos são criados com o objetivo de corrigir a classificação incorreta feita pelo neurônio pai.

Se os filhos, por sua vez, também não conseguirem classificar corretamente os exemplos de treinamento, o processo se repete, e são criados um ou dois neurônios filhos com o objetivo de corrigir os seus erros de classificação. O processo continua se repetindo recursivamente até que algum neurônio criado não tenha mais erros de classificação. Esse

último neurônio criado corrige os erros de seu pai que, por sua vez, corrige os erros de seu pai e assim por diante, até chegar ao primeiro neurônio.

Dois tipos de erros podem ocorrer com uma rede neural criada para resolver um problema de classificação em duas categorias, –1 e 1.

Os 'erros de positivos' ocorrem quando um neurônio u_n (peso W_n) qualquer classifica um exemplo de treinamento E^k como de classe negativa ou nula, mas a classe C^k desse exemplo é positiva (ver Figura 13).

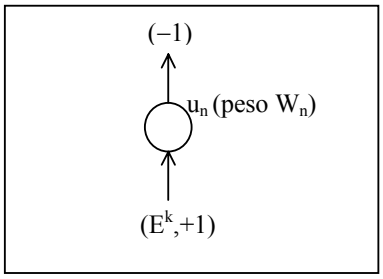


Figura 13. Situação em que ocorre ‘erro de positivo’ no neurônio u_n

Os 'erros de negativos' ocorrem quando um neurônio u_n (peso W_n) qualquer classifica um exemplo de treinamento E^k como de classe positiva ou nula, mas a classe C^k desse exemplo é negativa (ver Figura 14).

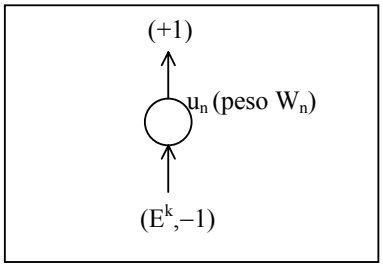


Figura 14. Situação em que ocorre ‘erro de negativo’ no neurônio u_n

Suponha um neurônio u_n que classifica exemplos de treinamento gerando 'erros de positivos'. O upstart trata essa situação criando um neurônio filho u_{n+} que irá tentar corrigir os 'erros de positivos' do neurônio pai. O objetivo do neurônio u_{n+} é, pois, corrigir a classificação daqueles exemplos do conjunto de treinamento que foram erroneamente classificados por u_n como de classe nula ou negativa, quando eram de classe positiva,

procurando, ao mesmo tempo, manter inalteradas as classificações corretas feitas pelo neurônio pai.

Similarmente, para tratar os 'erros de negativos' de um neurônio u_n, o upstart cria um neurônio filho u_{n-} cujo objetivo é corrigir a classificação dos exemplos classificados por u_n como nulo ou positivos, quando efetivamente são de classe negativa, procurando manter inalteradas as classificações corretas feitas por u_n (ver Figuras 15 e 16).

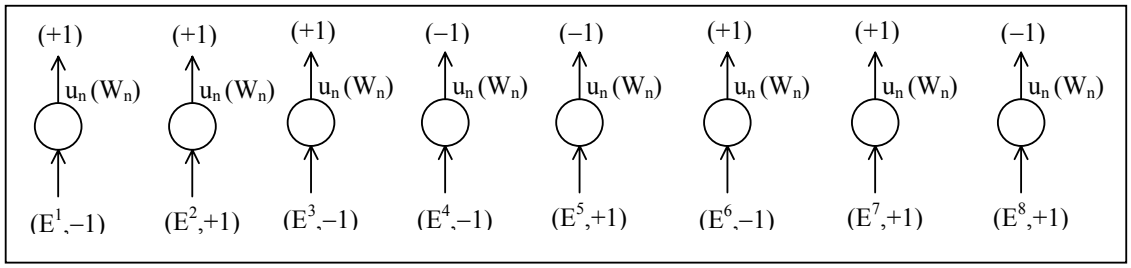


Figura 15. O neurônio u_n classifica corretamente três exemplos de treinamento (E^2, E^4, e E^7) provoca dois 'erros de positivos' (E^5 e E^8) e três 'erros de negativos' (E^1,E^3 e E^6)

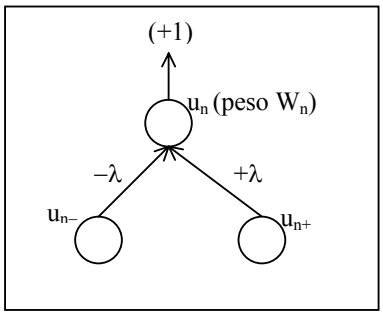


Figura 16. Os nós filhos u_{n-} e u_{n+} são criados para tentar corrigir os'erros de negativos' e 'erros de positivos'que ocorrem u_n

É importante notar que um neurônio u_n pode possuir um (quando ocorre uma das duas situações de erros), dois (quando ocorrem as duas situações de erro) ou nenhum (quando não há erro) filho(s). Por isso, de acordo com [Bishop 1995], "a rede upstart final terá a forma de uma árvore binária, embora alguns galhos possam não existir se eles não forem necessários".

Para que os neurônios filhos corrijam os erros feitos por seu pai, é necessário que haja uma conexão/reforço de filho para pai. Essa conexão para o pai (que é implementada pela conexão do neurônio filho, que é entrada para o neurônio pai) irá ser multiplicada por um

número grande e positivo λ, se o filho for um neurônio u_{n+} , gerado para se corrigir a situação 'erros de positivos'. A conexão será multiplicada por −λ se o filho for um neurônio u_{n-}, criado para corrigir a situação 'erros de negativos'. De acordo com [Campbell 1997], "o valor absoluto de −λ deve exceder a soma dos pesos positivos dos exemplos de u_n". A Figura 17 mostra a arquitetura de uma rede upstart, com dois atributos de entrada e dois neurônios filhos/auxiliares.

Considere uma situação na qual um neurônio u_n gera 'erros de negativos'. A classificação desses exemplos pode ser corrigida por meio do estabelecimento de uma conexão com peso negativo suficientemente grande (em valor absoluto), proveniente de u_{n-}. Com a adição dessa conexão, os exemplos 'erros de negativos' passam a ter saída negativa e não positiva ou nula como ocorria originalmente quando eram classificados pelo u_n e, conseqüentemente, esses exemplos são corrigidos. Se u_n também gerar 'erros de positivos', entretanto, esse reforço negativo vai dificultar a classificação desses exemplos, pois eles necessitam de reforço positivo e não negativo.

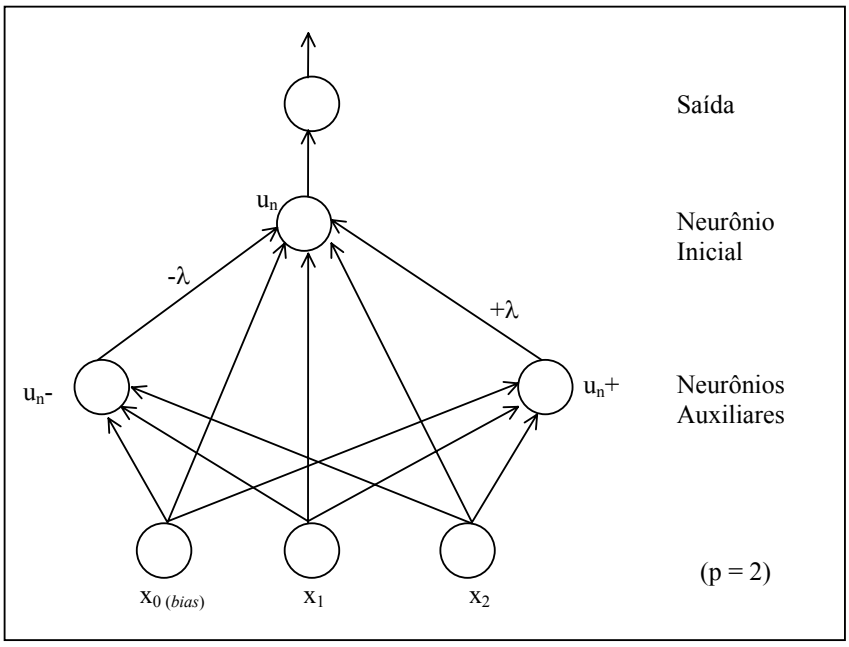


Figura 17. Arquitetura de rede construída pelo algoritmo upstart
A constante λ é um número grande e positivo

Assim sendo, é necessário que u_{n-} reforce negativamente apenas os 'erros de negativos' e não os 'erros de positivos'. Um exemplo com classe positiva classificado

corretamente por u_n não deve ser reforçado negativamente pois, com esse reforço, ele pode passar a ter saída negativa, ficando, desse modo, incorreto. Um exemplo classificado corretamente com classe negativa, entretanto, pode ou não receber essa conexão (a conexão negativa não irá mudar a saída já negativa).

Como a conexão de u_{n-} será multiplicada por um valor suficientemente grande (em valor absoluto) e negativo, os valores das conexões terão que ser 1 para os exemplos que necessitam de reforço negativo e zero para os demais exemplos.

Como já comentado, o nó u_{n-} é o nó encarregado de corrigir a classificação errada feita por u_n, dos exemplos que geraram 'erros de negativos'. Uma maneira de implementar esse processo é treinar u_{n-} com esses exemplos; antes do treinamento, entretanto, a classe negativa de cada um dos exemplos que provocou um 'erro de positivo' é trocada para classe positiva. Com esse artifício o vetor de pesos W_{n-} aprendido durante o treinamento de u_{n-} vai classificar os exemplos que geraram 'erros de negativos', como exemplos positivos. Como essa classificação será multiplicada por um número negativo, será dado um reforço/conexão suficientemente grande e negativo para esses exemplos.

Para o treinamento efetivo de u_{n-} é necessário também que exemplos originalmente com classe positiva façam parte do conjunto de treinamento; antes, entretanto, tais exemplos têm suas classes trocadas para negativo. O vetor de pesos obtido com o treinamento de u_{n-} deve ser tal que dê conexão com valor 1 para os exemplos que geraram os 'erros de negativo' em u_n e conexão zero para os demais, dado que o papel de u_{n-} é o de corrigir os 'erros de negativo' e não interferir nas demais classificações (razão pela qual a conexão para os demais exemplos deve ser 0).

Um procedimento análogo é adotado para o tratamento dos exemplos que provocaram 'erros de positivos' em u_n. A classificação de tais exemplos deve ser corrigida por meio do treinamento do nó u_{n+}. O conjunto de treinamento do nó u_{n+} é formado pelos exemplos que provocaram 'erros de positivos' mais os exemplos com classe negativa usados para treinar u_n. Após o treinamento de u_{n+}, o vetor de pesos aprendido deve ser tal que dê conexão 1 para os exemplos que provocaram 'erros de positivos' e 0 para os demais. A conexão será multiplicada por um inteiro positivo bem grande.

O algoritmo upstart funciona, então, da seguinte maneira: começa com uma rede sem qualquer neurônio intermediário. Primeiramente, é treinado um neurônio u_n, usando o

algoritmo pocket com modificação ratchet. Assumindo que esse neurônio u_n possa ter algum erro de classificação, então são treinados os neurônios u_{n+}, caso ocorra a situação de 'erros de positivos' e u_{n-}, caso ocorra a situação de 'erros de negativos'. Se os u_{n+} e u_{n-} classificarem erradamente, serão criados e treinados seus próprios filhos e assim por diante até que um neurônio que não tenha erros seja criado ou até um critério de parada (como número máximo de neurônios auxiliares) seja satisfeito.

Considere um exemplo de treinamento e sua respectiva classe dados por (E^k,C^k) e seja u_n^k a saída de u_n para o exemplo E^k.

1) exemplos de treinamento para u_{n+} são:

1.1) positivos – aqueles que

$$C^k = 1 \text{ e } u_n^k = -1$$

1.2) negativos – aqueles que:

$$C^k = -1$$

1.3) outros exemplos são ignorados

2) exemplos de treinamento para u_{n-} são:

2.1) positivos – aqueles que

$$C^k = -1 \text{ e } u_n^k = 1$$

2.2) negativos – aqueles que:

$$C^k = 1$$

2.3) outros exemplos são ignorados

Note que o procedimento de geração dos filhos é recursivo; os valores são definidos do último para o primeiro. Quando um neurônio u_n recebe os reforços dos seus filhos, então as saídas dos exemplos de treinamento de u_n terão que ser novamente calculadas, pois agora essas saídas levam em consideração os reforços positivos e/ou negativos. Devido à proposta original não ser específica com relação ao cálculo das novas saídas, existem diferentes possibilidades para esse cálculo. Esse trabalho propõe duas que são descritas nas próximas subseções.

5.2 A ABORDAGEM UPSTART-SOMA

Na abordagem upstart-soma, a nova saída para cada exemplo de treinamento será encontrada usando a saída original somada com os reforços provenientes do(s) neurônios filho(s). Seja W_n o vetor de pesos do neurônio u_n, E o conjunto de treinamento em u_n e u_n^k a saída original de u_n para o exemplo E^k de E. A saída original (sem os reforços dos filhos) é:

$$u_n^k = \begin{cases} 1 \text{ se } W_n.E^k > 0 \\ -1 \text{ caso contrário} \end{cases}$$

A nova saída (calculada após os reforços) é dada então por:

$$\text{novo\_}u_n^k = \begin{cases} 1 \text{ se } W_n.E^k + (\lambda \,\text{conexão}_{u_{n+}}^k) + (-\lambda \,\text{conexão}_{u_{n-}}^k) > 0 \\ -1 \text{ caso contrário} \end{cases}$$

onde $\text{conexão}_{u_{n+}}^k$ é o valor da conexão do neurônio filho u_{n+} e $\text{conexão}_{u_{n-}}^k$ é o valor da conexão do neurônio filho u_{n-}, para o exemplo de treinamento E^k. O valor λ é um número grande. Note que os valores de $\text{conexão}_{u_{n+}}^k$ e $\text{conexão}_{u_{n-}}^k$ nunca podem ser diferentes de zero simultaneamente. Isso ocorre pois um exemplo de treinamento E^k não pode gerar a situação de 'erros de positivos' e 'erros de negativos' simultaneamente. Note também que $\text{conexão}_{u_{n+}}^k$ e $\text{conexão}_{u_{n-}}^k$ podem não existir; nesse caso, os seus valores são assumidos como zero.

Finalmente, o valor da $\text{conexão}_{u_n}^k$ é encontrado da seguinte maneira:

$$\text{conexão}_{u_n}^k = \begin{cases} 1 \text{ se novo\_}u_n^k = 1 \\ 0 \text{ caso contrário} \end{cases}$$

5.3 A ABORDAGEM UPSTART-TREINAMENTO

Na abordagem upstart-treinamento o vetor de pesos W_n do neurônio u_n é novamente treinado usando o algoritmo pocket com modificação ratchet, mas agora levando em conta uma ou duas dimensões adicionais, que são os reforços provenientes dos neurônios filhos.

Os exemplos de treinamento (com uma ou duas dimensões) multiplicados pelo novo W_n fornecerão, então, as novas saídas. Sendo E o conjunto de treinamento em u_n e sendo $novo\_u_n^k$ a nova saída de u_n para o exemplo E^k de E, a nova saída é, então, encontrada da seguinte maneira:

$$novo\_u_n^k = \begin{cases} 1 \text{ se } W_n.E1^k > 0 \\ -1 \text{ caso contrário} \end{cases}$$

onde $E1^k$ é exemplo E^k acrescido de uma ou duas dimensões: (λ $conexão_{u_{n+}}^k$) e ($-\lambda$ $conexão_{u_{n-}}^k$). W_n é o vetor de pesos encontrado através da execução do algoritmo pocket com modificação ratchet em E1 e as variáveis $conexão_{u_{n+}}^k$ e $conexão_{u_{n-}}^k$ tem o mesmo significado da abordagem upstart-soma. A $conexão_{u_n}^k$ é definida da mesma maneira que na abordagem upstart-soma (ver Seção 5.2).

5.4 CONSIDERAÇÕES SOBRE AS DUAS ABORDAGENS

O valor λ responsável pelos reforços que os neurônios recebem pode não ser suficientemente grande. Se for esse o caso, somá-lo não irá corrigir o problema; entretanto, treinar novamente o neurônio irá corrigir. Essa é uma vantagem importante da abordagem upstart-treinamento, dado que o problema da classificação incorreta é corrigido, mesmo que o número λ não seja suficientemente grande.

O vetor de pesos novamente treinado usando a abordagem upstart-treinamento pode ser melhor, pior ou igual ao vetor de pesos original com os reforços (ie, pode classificar corretamente mais, menos ou a mesma quantidade de exemplos de treinamento). Caso ele seja igual, a abordagem upstart-treinamento funcionará exatamente como a upstart-soma (considerando que o número grande é suficientemente grande). Se ele for pior, o desempenho do algoritmo upstart usando upstart-treinamento tenderá a ser pior do que se tivesse sido usado upstart-soma. Se ele for melhor, entretanto, o desempenho do algoritmo upstart tenderá a ser melhor usando upstart-treinamento. É importante ressaltar que a versão

upstart-treinamento é computacionalmente mais lenta, pois há o custo extra de se treinar novamente os vetores de peso.

O sistema CONEB disponibiliza as duas maneiras, ie, o usuário pode escolher qual delas prefere usar. O pseudocódigo do algoritmo upstart, usando a abordagem upstart-treinamento, é mostrado nas Figuras 18 e 19.

```
Inicialização de variáveis globais:
indice_pos = 0 {representa o número do neurônio auxiliar positivo u_n+}
indice_neg = 0 {representa o número do neurônio auxiliar negativo u_n-}
MAX = número máximo de neurônios auxiliares da rede
Woutput_pos[MAX/2] {um vetor de vetores de peso dos neurônio auxiliares positivos}
Woutput_neg[MAX/2] {um vetor de vetores de peso dos neurônio auxiliares negativos}
nó_positivo = 0 {variável booleana que identifica se um neurônio é positivo}
nó_negativo = 0 {variável booleana que identifica se um neurônio é negativo}
Winicial {Vetor de pesos para o neurônio inicial}
{Winicial, Woutput_pos e Woutput_neg são as saídas do sistema}
NUMERO_GRANDE {número grande que irá multiplicar as conexões}

procedure upstart(E)
{Entradas: E conjunto de treinamento com n elementos da forma:
 E^k = (x^k_0, x^k_1,x^k_2,…, x^k_p,C^k) representando n instâncias  pertencentes
 às classes C^k = {1,-1}                                                    }
begin
 pocket_ratchet(E,W) {use o algoritmo pocket com modificação ratchet para gerar o
                      primeiro neurônio da rede, expresso pelo vetor de pesos W}
 temp_indice_pos = indice_pos ;  temp_indice_neg = indice_neg
 temp_nó_positivo = nó_positivo ;  temp_nó_negativo = nó_negativo
 conexao_pos = conexao_neg = 0
 if(erros_de_positivos(E,W) and indice_pos < MAX/2) then
    begin
     cria_treinamento_pos(E,W,Epos)
     conexao_pos = 1
     indice_pos = indice_pos + 1
     nó_positivo = 1
     nó_negativo = 0
     upstart(Epos)
    end
 if(erros_de_negativos(E,W) and indice_neg < MAX/2) then
    begin
     cria_treinamento_neg(E,W,Eneg)
     conexao_neg = 1
     indice_neg = indice_neg + 1
     nó_positivo = 0
     nó_negativo = 1
     upstart(Eneg)
    end

 mod_entrada_upstart(E,E1,conexao_neg,conexao_pos,temp_indice_neg,temp_indice_pos)
  pocket_ratchet(E1,W)
  if(temp_nó_positivo) {nó corrente é nó positivo} then
   Woutput_pos[temp_indice_pos] = W  {guarda o vetor de pesos auxiliar positivo}
  else if(temp_nó_negativo) {nó corrente é nó negativo} then
          Woutput_neg[temp_indice_neg] = W {guarda vetor de pesos auxiliar
                                                                  negativo}
          else {nó corrente é o primeiro nó da rede}
              Winicial = W
end
```

Figura 18. Pseudocódigo do algoritmo upstart usando abordagem upstart-treinamento

```
procedure mod_entrada_upstart(E,E1,conexao_neg,conexao_pos,indice_neg,indice_pos)
begin
 E1 = E
 bol = 0
 if(conexao_pos)
 begin
  bol = bol + 1
  for all E1^k ∈ E1 do
   if(E^k.Woutput_pos[indice_pos + 1] > 0) then E1^k_p+bol =  NÚMERO_GRANDE
                                           else E1^k_p+bol = 0
 end
 if(conexao_neg)
 begin
  bol = bol + 1
   for all E1^k ∈ E1 do
    if(E^k.Woutput_neg[indice_neg + 1] > 0) then E1^k_p+bol = -NÚMERO_GRANDE
                                            else E1^k_p+bol = 0
 end
end
```

Figura 19. Pseudocódigo do algoritmo upstart usando abordagem upstart-treinamento (cont.)

O procedimento *erros_de_negativos*(E,W) verifica se existem 'erros de negativos' no conjunto E, ie, se a multiplicação de algum exemplo E^k por W tem como resultado um valor positivo ou nulo, mas a classe C^k do exemplo é negativa. Se existir algum, o procedimento retorna *true* caso contrário retorna *false.* Similarmente, o procedimento *erros_de_positivos*(E,W) verifica se existe a situação 'erros de positivos' em E, ie, algum exemplo E^k tem saída negativa ou nula, mas classe C^k positiva. Caso haja, o procedimento retorna *true* e caso contrário retorna *false*.

O procedimento *cria_treinamento_pos*(E,W,Epos) cria um conjunto de treinamento Epos a partir da entrada E e do vetor de pesos W. O conjunto Epos terá todos os exemplos de treinamento de E que possuem saída negativa ou nula ($E^k.W \leq 0$), mas classe C^k positiva, bem como todos os exemplos de E que possuem classe C^k negativa. O conjunto Epos é o conjunto de treinamento do neurônio filho u_{n+}. Similarmente, o procedimento *cria_treinamento_neg*(E,W,Eneg) cria um conjunto de treinamento Eneg com todos os exemplos de E que possuem saída positiva ou nula, mas classe negativa, bem como os exemplos que possuem classe positiva. O conjunto Eneg é o conjunto de treinamento do neurônio filho u_{n-}.

O procedimento *mod_entrada_upstart*(E, E1, conexao_neg, conexao_pos, temp_indice_neg, temp_indice_pos) cria uma nova entrada E1 usando a entrada E e seus neurônios filhos. As variáveis conexao_neg e conexao_pos são usadas para indicar se o

neurônio atual tem conexão positiva e/ou negativa respectivamente, ie, se ele tem neurônios filhos u_{n-} e/ou u_{n+}. Caso haja conexão, então uma ou duas dimensões serão criadas. Essas dimensões terão os valores das conexões dos neurônios u_{n+} ou u_{n-} multiplicadas por um número grande λ (caso de u_{n+}) ou $-\lambda$ (caso de u_{n-}). As variáveis temp_indice_neg e temp_indice_pos acrescidas de um são os índices para os vetores de peso dos neurônios u_{n+} e u_{n-}. A referência [Palma Neto & Nicoletti 2004] apresenta um estudo detalhado sobre o algoritmo upstart.

5.5 EXEMPLO DE FUNCIONAMENTO DO UPSTART

Nesta seção é mostrado o funcionamento do algoritmo upstart num exemplo simples, usando a abordagem upstart-treinamento (ver Seção 5.3). Na Seção 5.5.1 é mostrado como uma rede upstart é construída e na Seção 5.5.2 é mostrado como usar uma rede upstart já construída para classificar um exemplo.

5.5.1 Treinando uma Rede Upstart

Para exemplificar o funcionamento do algoritmo upstart, nesta seção ele é usado para o aprendizado do conceito de paridade-2, conhecido como XOR. Os exemplos de treinamento estão definidos na Tabela 17. Note que o termo constante *bias* já está incluído na Tabela.

Tabela 17. Conjunto de treinamento para o aprendizado do conceito XOR (*bias* incluído)

Exemplo	x_0 (*bias)*	x_1	x_2	Classe
E^1	1	0	0	–1
E^2	1	0	1	1
E^3	1	1	0	1
E^4	1	1	1	–1

Inicialmente, o algoritmo pocket com modificação ratchet é executado para a criação de um nó u_0, tendo como entrada o conjunto de treinamento descrito na Tabela 17. Como esse conjunto de treinamento é não linearmente separável, o vetor de pesos W_0, associado ao neurônio u_0, não consegue classificar todos os exemplos corretamente. Sendo

assim, o algoritmo upstart terá que treinar, no mínimo, um ou dois neurônios auxiliares (u_{1-} e/ou u_{1+}) para corrigir os erros de classificação de u_0.

Ao se executar o algoritmo pocket com modificação ratchet, seja <–1 0 –2> o valor para W_0. Desse modo, W_0 classifica corretamente os exemplos de treinamento E^1 e E^4 e classifica incorretamente os exemplos E^2 e E^3.

Ambos os exemplos de treinamento classificados incorretamente possuem classe positiva e, portanto, provocam a situação de 'erros de positivos'. É necessário, então, criar um neurônio auxiliar u_{1+} com o intuito de corrigir a classificação de E^2 e E^3.

O conjunto de treinamento que é usado para treinar u_{1+} é construído pelo procedimento *cria_treinamento_pos*. Nele estão os exemplos que causaram os 'erros de positivos' isto é, os exemplos E^1 e E^2 mais os exemplos de treinamento que possuem classe negativa, ou seja, E^2 e E^4. Portanto u_{1+} será treinado com o mesmo conjunto de treinamento mostrado na Tabela 17.

O neurônio u_{1+} estará conectado ao neurônio u_0, ie, as conexões de u_{1+} serão usadas como entradas para u_0. Essas conexões serão multiplicadas por um número grande e positivo. Até o final desse exemplo, usaremos o número 10 (ou –10 para situações de 'erros de negativos'). O próximo passo do algoritmo é gerar um vetor de pesos W_{1+} para u_{1+}, utilizando os dados da Tabela 17. Suponha que o vetor de pesos seja <–1 2 2>.

Esse vetor de pesos classifica corretamente E^1, E^2 e E^3 e classifica incorretamente E^4. Como o exemplo de treinamento E^4 tem classe negativa ele provoca uma situação 'erros de negativos'. Para se corrigir E^4, é necessário criar um neurônio u_{2-} e treiná-lo para corrigir a classificação incorreta de seu pai. O conjunto de treinamento para u_{2-} é definido pelo procedimento *cria_treinamento_neg*. Nele vão estar os exemplos que causaram os 'erros de negativos' ie, o exemplo E^4 e os exemplos positivos utilizados no treinamento de u_1. O conjunto de treinamento de u_{2-} está descrito na Tabela 18. O neurônio u_{2-} estará conectado ao neurônio u_{1+}, ie, as conexões de u_{2-}, multiplicadas por –10, serão usadas como entradas para u_{1+}.

Tabela 18. Conjunto de treinamento de u_{2-}

Exemplo	x_0 (*bias)*	x_1	x_2	Classe
E^2	1	0	1	–1
E^3	1	1	0	–1
E^4	1	1	1	1

Os exemplos de treinamento da Tabela 18 são linearmente separáveis e, portanto, o algoritmo pocket com ratchet consegue gerar com facilidade um vetor que classifica corretamente todos os três exemplos. Seja o vetor de pesos gerado $W_{2-} = <-3\ 2\ 2>$.

Como esse vetor de pesos classifica corretamente todos os seus exemplos de treinamento, não é mais necessário gerar neurônios para corrigir algum exemplo de treinamento. Sendo assim, a recursividade acaba e as conexões dos neurônios auxiliares passam a ser usadas como entradas para os seus neurônios pais e os vetores de pesos desses neurônios serão novamente treinados (para levar em consideração os reforços). As conexões de u_{2-}, multiplicadas por −10, serão usadas como entrada para u_{1+}.

A Tabela 19 mostra as conexões de u_{2-} e entradas de u_{1+} para cada exemplo de treinamento e a Tabela 20 mostra o novo conjunto de treinamento para u_{1+}. Note que apesar de E^1 não ter sido usado em u_{2-}, é necessário multiplicá-lo pelo vetor de pesos W_{2-} gerado em u_{2-} para achar a conexão para E^1 em u_{1+}.

Tabela 19. Conexões de u_{2-} e entradas para u_{1+}

Exemplo	$E.W_{2-}$	Conexão de u_{2-}	Entrada para u_{1+}
E^1	–3	0	0
E^2	–1	0	0
E^3	–1	0	0
E^4	1	1	–10

Tabela 20. Conjunto de treinamento para u_1+ com conexão de u_{2-}

Exemplo	x_0 (*bias)*	x_1	x_2	Conexão de u_{2-} multiplicadas por –10	Classe
E^1	1	0	0	0	–1
E^2	1	0	1	0	1
E^3	1	1	0	0	1
E^4	1	1	1	–10	–1

Tendo modificado o conjunto de treinamento do neurônio u_{1+}, é necessário agora gerar um novo vetor de pesos W_{1+}, usando o conjunto de dados da Tabela 20. Esse conjunto de dados é linearmente separável, e, portanto o algoritmo pocket com modificação ratchet consegue gerar com facilidade um vetor de pesos que classifica corretamente todos os exemplos de treinamento. Esse novo vetor de pesos W_{1+} pode ser $<-1\ 2\ 2\ 10>$, note que esse vetor de pesos classifica corretamente todos os exemplos de treinamento e, portanto, a conexão/reforço de u_{2-} realmente conseguiu corrigir o exemplo E^4 que estava sendo

classificado erradamente. As conexões de u_{1+} multiplicadas por 10 são usadas como entrada para u_0. A Tabela 21 mostra o conjunto de treinamento para u_0 com conexão de u_{1+}.

Tabela 21. Conjunto de treinamento para u_0 com conexões de u_{1+}

Exemplo	x_0 (*bias)*	x_1	x_2	Conexão de u_{1+} multiplicadas por 10	Classe
E^1	1	0	0	0	–1
E^2	1	0	1	10	1
E^3	1	1	0	10	1
E^4	1	1	1	0	–1

Tendo modificado o conjunto de treinamento do neurônio u_0, é necessário agora gerar um novo vetor de pesos W_0, usando o conjunto de dados da Tabela 21. Esse conjunto de dados é linearmente separável, e, portanto o algoritmo pocket com modificação ratchet consegue gerar com facilidade um vetor de pesos que classifica corretamente todos os exemplos de treinamento. Seja esse novo vetor de pesos $W_0 = <–1\ 0\ –1\ 10>$; note que esse vetor de pesos classifica, de fato, corretamente todos os exemplos de treinamento e, portanto, a conexão/reforço de u_{1+} realmente conseguiu corrigir os exemplos E^2 e E^3 que estavam errados originalmente. Assim sendo, o algoritmo termina com todos os exemplos de treinamento classificados corretamente. A Figura 20 mostra o conceito gerado.

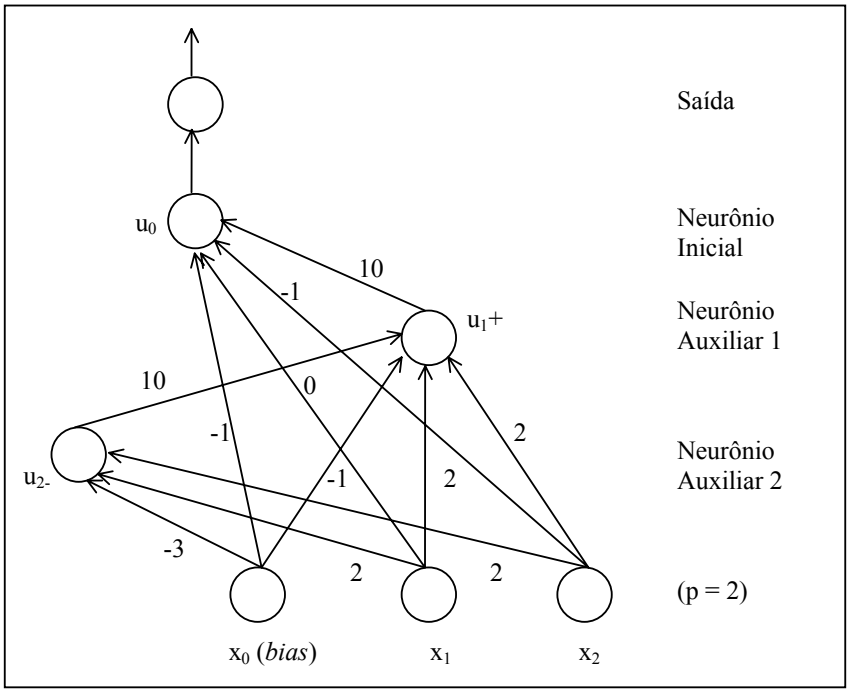


Figura 20. Conceito de paridade-2 expresso por uma rede upstart

É importante ressaltar que, por questões didáticas, foi mostrado um exemplo que necessitou de dois neurônios auxiliares (u_{1+} e u_{2-}) para classificar corretamente todos os exemplos de treinamento. Apenas um neurônio auxiliar (u_{1+} ou u_{1-}), no entanto, é suficiente para classificar todos os exemplos corretamente. Se o vetor de pesos W_0 inicial tiver 75% de precisão ie, classificar corretamente três das quatro instâncias de treinamento, então será necessário apenas um neurônio auxiliar para corrigir o exemplo de treinamento erroneamente classificado.

5.5.2 Testando uma Rede Upstart

A seguir é mostrado o uso da rede upstart aprendida na Seção anterior, na classificação de novos exemplos, como os descritos na Tabela 22 (que já têm o termo *bias* incluído).

Tabela 22. Conjunto de teste qualquer para teste (*bias* incluído)

Exemplo	x_0 (*bias)*	x_1	x_2	Classe
T^1	1	2	2	–1
T^2	1	2	0	1

Primeiramente tem-se que multiplicar os exemplos de teste aos vetores de peso dos neurônios mais externos, ie, os neurônios que foram gerados no final. As conexões encontradas nessas multiplicações para cada exemplo de teste, multiplicadas por o número grande (em valor absoluto) λ ou $-\lambda$, serão usadas, então, como entradas para o neurônio pai. O processo então continua até o primeiro neurônio da rede. Se a saída do exemplo de teste no primeiro neurônio for igual à sua classe, então a rede upstart gerada classifica o exemplo corretamente, senão, o exemplo está classificado incorretamente.

O neurônio mais externo da rede descrita na Seção 5.5.1 é o u_{2-} e o seu vetor de pesos é $W_{2-} = \langle -3\ 2\ 2 \rangle$. As conexões dos exemplos de teste descritos na Tabela 22, multiplicadas por W_{2-} e depois multiplicadas por –10 serão usadas como entrada para o neurônio pai, ie, u_{1+}. A Tabela 23 ilustra a configuração das conexões de u_{2-} e entradas para u_{1+} para cada exemplo de teste e a Tabela 24 ilustra o conjunto de teste para u_{1+}.

Tabela 23. Conexões de u_{2-} e entradas para u_{1+}

Exemplo	$T.W_{2-}$	Conexão de u_{2-}	Entrada para u_1+
T^1	5	1	–10
T^2	1	1	–10

Tabela 24. Conjunto de teste para u_{1+} com conexão de u_{2-}

Exemplo	x_0 (*bias)*	x_1	x_2	Conexão de u_{2-} multiplicadas por –10	Classe
T^1	1	2	2	–10	–1
T^2	1	2	0	–10	1

O próximo passo é multiplicar os exemplos da Tabela 24 pelo vetor de pesos de u_{1+}, ie, W_{1+} = <–1 2 2 10>. As conexões de T^1 e T^2 em u_1+ multiplicadas por 10 serão usadas, então, como entrada para u_0. A Tabela 25 ilustra o conjunto de teste para u_0.

Tabela 25. Conjunto de teste para u_0 com conexões de u_{1+}

Exemplo	x_0 (*bias)*	x_1	x_2	Conexão de u_{1+} multiplicadas por 10	Classe
T^1	1	2	2	0	–1
T^2	1	2	0	0	1

O passo final é multiplicar os exemplos de teste da Tabela 25 por W_0 = <–1 0 –1 0> e então comparar a saída encontrada com a classe do exemplo de teste. O exemplo T^1 multiplicado por W_0 é igual a –3 que configura uma saída negativa e, portanto, a rede classifica corretamente esse exemplo. O exemplo T^2 multiplicado por W_0 é igual a –1 que configura uma saída negativa e, portanto, a rede classifica incorretamente esse exemplo.

5.6 PROVA DE CONVERGÊNCIA DO ALGORITMO UPSTART

Dada uma entrada finita e não contraditória o algoritmo upstart consegue gerar uma rede que classifica corretamente todos os exemplos de entrada. Para detalhes sobre a prova, consultar [Frean 1990a].

CAPÍTULO 6. O ALGORITMO DISTAL

Este capítulo apresenta e discute as principais características do algoritmo DistAl. Parte das informações apresentadas neste capítulo foram extraídas das referências que propõem esse algoritmo e, em especial, de [Yang et al 1997] e [Yang et al 1999]. A referência [Palma Neto & Nicoletti 2003b] faz um estudo detalhado sobre o algoritmo DistAl.

DistAl é um algoritmo de aprendizado supervisionado de máquina que, de maneira construtiva, cria uma rede neural a partir de um conjunto de treinamento. Cada uma das instâncias de treinamento usada para a criação da rede é descrita por um vetor de valores de atributos e classe associada.

O DistAl usa uma métrica de distância (daí o seu nome) entre as instâncias ou t–uplas de treinamento para construir a única camada intermediária de neurônios que participa da rede; esses neurônios são do tipo *esférico*.

O peso e os dois *thresholds* associados a cada neurônio da camada intermediária são determinados comparando as distâncias entre as instâncias de treinamento. Desse modo, procedimentos iterativos, custosos e demorados que comumente são usados para o cálculo de pesos e *thresholds* em algoritmos neurais construtivos, são substituídos por um cálculo mais simples, rápido e com menos custo computacional.

6.1 CARACTERÍSTICAS DO DISTAL

O DistAl se diferencia de outros algoritmos neurais construtivos de aprendizado especialmente porque:

- usa unidades threshold *esféricas* – uma variação das TLUs – nos neurônios da camada intermediária e
- não usa um algoritmo iterativo para calcular os pesos e os *thresholds*

No DistAl um neurônio esférico i (graficamente mostrado na Figura 21) está associado a:

- um vetor de pesos W_i
- dois valores *threshold* identificados por $\theta_{i,inferior}$ e $\theta_{i,superior}$
- uma determinada métrica de distância d

Após o término da fase de aprendizado (descrita em detalhes na Seção 6.3.1), a rede treinada está pronta para o uso na classificação de novas instâncias. A classificação de uma nova instância representada por um vetor X^p é feita calculando a distância $d(W_i,X^p)$. O valor de saída do neurônio i é determinado da seguinte maneira:

$O_i^p = 1$ se $\theta_{i,inferior} \leq d(W_i, X^p) \leq \theta_{i,superior}$

$O_i^p = 0$ caso contrário

Assim sendo, cada neurônio esférico i da camada intermediária criado pelo DistAl representa um conjunto ou *cluster* de instâncias de treinamento que estão confinadas em um espaço limitado por duas regiões esféricas concêntricas, com raios $\theta_{i,inferior}$ e $\theta_{i,superior}$ respectivamente, como mostra a Figura 21. O peso do neurônio determina o centro comum das duas regiões e os dois *thresholds* associados ao neurônio determinam os dois ráios das hiper-esferas concêntricas.

A Figura 21 mostra a representação gráfica de um neurônio intermediário, num espaço euclidiano tridimensional. O vetor mais escuro representa o peso associado ao neurônio e os quatro outros vetores, os participantes do cluster representado pelo neurônio em questão. Note que os quatro vetores estão confinados no espaço determinado pelas duas esferas concêntricas.

O peso e *thresholds* associados a cada neurônio intermediário não são calculados iterativamente. O DistAl calcula apenas uma vez as distâncias entre as instâncias de treinamento e determina o peso dos neurônios da camada intermediária usando uma estratégia simples. Assim sendo, os pesos e os *thresholds* são determinados sem o custo computacional dos processos iterativos usados pelos algoritmos construtivos.

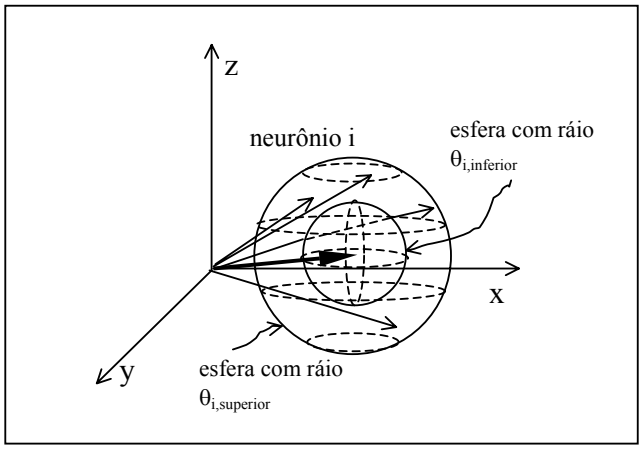


Figura 21. Representação gráfica de um neurônio da camada intermediária criado pelo DistAl

O uso de apenas um cálculo de distância entre as instâncias de entrada ao invés de procedimentos iterativos, custosos e demorados faz com que DistAl seja bastante rápido quando comparado a outros algoritmos construtivos neurais de aprendizado. Isto torna este algoritmo particularmente adequado para tarefas de grande porte.

6.2 MÉTRICAS DE DISTÂNCIA

A escolha de uma métrica de distância adequada é fundamental para se obter um bom desempenho do algoritmo DistAl. As diferentes métricas de distância representam noções distintas de distância, no espaço definido pelos atributos que descrevem as instâncias de treinamento e pela classe.

As distâncias a seguir são propostas na literatura (ver particularmente [Wilson & Martinez 1997]) e usadas em [Yang et al 1997] e [Yang et al 1999]. A distância entre atributos com valores nominais (e.g., atributo *a* com valores x e y) pode ser calculada usando as fórmulas:

$$\text{overlap: } d_{over}(x,y) \begin{cases} = 0 \text{ se } x = y \\ = 1 \text{ caso contrário} \end{cases}$$

$$\text{value difference: } d_{val}(x,y) = \sum_{c=1}^{C} \left| \frac{N_{a,x,c}}{N_{a,x}} - \frac{N_{a,y,c}}{N_{a,y}} \right|^{q}$$

onde:

$N_{a,x}(N_{a,y})$: número de instâncias de treinamento cujo atributo *a* têm o valor x(y).

$N_{a,x,c}$ ($N_{a,y,c}$): número de instâncias de treinamento de classe c, cujo atributo *a* têm o valor x(y).

C : número total de classes existentes no conjunto de treinamento.

q: constante (1 se Manhatan, 2 se euclideana, etc).

Se qualquer uma das instâncias de treinamento possui atributos com valores inexistentes, a distância relativa aos atributos em questão é assumida como 1. Para a descrição das métricas de distância a seguir, foram adotadas as convenções:

- um conjunto de treinamento com m instâncias.
- n: número de atributos que descrevem as instâncias
- instâncias de treinamento notadas por:

  $X^1 = [X_1^1, X_2^1, \ldots, X_n^1]$

  ……

  $X^p = [X_1^p, X_2^p, \ldots, X_n^p]$

  ……

  $X^q = [X_1^q, X_2^q, \ldots, X_n^q]$

  ……

  $X^m = [X_1^m, X_2^m, \ldots, X_n^m]$
- $max_i = \max\{X_i^1, \ldots, X_i^p, \ldots, X_i^q, \ldots, X_i^m\}$
- $min_i = \min\{X_i^1, \ldots, X_i^p, \ldots, X_i^q, \ldots, X_i^m\}$
- σ_i = desvio padrão$\{X_i^1, \ldots, X_i^p, \ldots, X_i^q, \ldots, X_i^m\}$
- med_i = média $\{X_i^1, \ldots, X_i^p, \ldots, X_i^q, \ldots, X_i^m\}$
- $soma_i = soma\{X_i^1, \ldots, X_i^p, \ldots, X_i^q, \ldots, X_i^m\}$

1) distância euclidiana: $d_1(X^p, X^q) = \sqrt{\sum_{i=1}^{n}\left[\left(X_i^p - X_i^q\right)^2\right]}$

2) distância euclidiana normalizada: $d_2(X^p, X^q) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left[\left(\frac{X_i^p - X_i^q}{\max_i - \min_i}\right)^2\right]}$

3) distância de Manhattan: $d_3(X^p, X^q) = \sum_{i=1}^{n}\left|X_i^p - X_i^q\right|$

4) distância de Manhattan normalizada: $d_4(X^p, X^q) = \frac{1}{n}\sum_{i=1}^{n}\frac{|X_i^p - X_i^q|}{\max_i - \min_i}$

5) valor máximo: $d_5(X^p, X^q) = \max_i \left|X_i^p - X_i^q\right|$

6) valor máximo normalizado: $d_6(X^p, X^q) = \max_i\left[\frac{\left|X_i^p - X_i^q\right|}{\max_i - \min_i}\right]$

7) Dice Coefficient: $d_7(X^p, X^q) = 1 - \frac{2\sum_{i=1}^{n} X_i^p X_i^q}{\sum_{i=1}^{n}\left(X_i^p\right)^2 + \sum_{i=1}^{n}\left(X_i^q\right)^2}$

8) Cosine coefficient: $d_8(X^p, X^q) = 1 - \frac{\sum_{i=1}^{n} X_i^p X_i^q}{\sqrt{\sum_{i=1}^{n}\left(X_i^p\right)^2 \sum_{i=1}^{n}\left(X_i^q\right)^2}}$

9) Jaccard coefficient: $d_9(X^p, X^q) = 1 - \frac{\sum_{i=1}^{n} X_i^p X_i^q}{\sum_{i=1}^{n}\left(X_i^p\right)^2 + \sum_{i=1}^{n}\left(X_i^q\right)^2 - \sum_{i=1}^{n} X_i^p X_i^q}$

10) Camberra : $d_{10}(X^p, X^q) = \sum_{i=1}^{n}\frac{\left|X_i^p - X_i^q\right|}{\left|X_i^p + X_i^q\right|}$

11) Correlation: $d_{11}(X^p, X^q) = \frac{\sum_{i=1}^{n}\left(X_i^p - \text{med}_i\right)\left(X_i^q - \text{med}_i\right)}{\sqrt{\sum_{i=1}^{n}\left(X_i^p - \text{med}_i\right)\left(X_i^p - \text{med}_i\right)\sum_{i=1}^{n}\left(X_i^q - \text{med}_i\right)\left(X_i^q - \text{med}_i\right)}}$

12) Chi – square: $d_{12}(X^p,X^q) = \sum_{i=1}^{n} \frac{1}{soma_i} \left(\frac{X_i^p}{\sum_{i=1}^{n} X_i^p} - \frac{X_i^q}{\sum_{i=1}^{n} X_i^q} \right)^2$

13) N. Chi – Square: $d_{13}(X^p,X^q) =$

$$\sum_{i=1}^{n} \frac{max_i - min_i}{soma_i} \left(\frac{\frac{X_i^p}{max_i - min_i}}{\sum_{i=1}^{n} \frac{X_i^p}{max_i - min_i}} - \frac{\frac{X_i^q}{max_i - min_i}}{\sum_{i=1}^{n} \frac{X_i^q}{max_i - min_i}} \right)^2$$

Note que essas distâncias levam em consideração apenas valores de atributos numéricos. Note também que a normalização existente em 2), 4) e 6) é feita por uma divisão de um valor por $max_i - min_i$. Chamamos esse tipo de normalização de *normalização baseada em limites*. Pode-se substituir $max_i - min_i$ por $4 * \sigma_i$ e, neste caso, a normalização é chamada de *normalização baseada em desvio padrão*. Para tratar valores nominais, são usadas as seguintes métricas:

14) distância euclidiana nominal : $d_{14}(X^p, X^q) = \sqrt{\frac{1}{n} \sum_{i=1}^{n} d_{nom}\left(X_i^p, X_i^q\right)^2}$

15) distância de Manhattan nominal : $d_{15}(X^p, X^q) = \frac{1}{n} \sum_{i=1}^{n} d_{nom}\left(X_i^p, X_i^q\right)$

16) valor máximo nominal : $d_{16}(X^p, X^q) = max_i\left(d_{nom}\left(X_i^p, X_i^q\right)\right)$

onde $d_{nom}(x,y) = d_{over}(x,y)$ ou $d_{val}(x,y)$, como definido no início dessa Seção.

Em certos domínios, os valores de um determinado atributo entre dois limites ou *thresholds* ($a_{inferior}$ e $a_{superior}$, por exemplo) podem identificar, sozinhos, instâncias pertencentes a uma mesma classe de saída. Nesses casos, a distância entre t-uplas ou instâncias de entrada não é muito relevante. Assim sendo, um neurônio intermediário que

armazene apenas o nome do atributo a e valores dos thresholds (a_{baixo} e a_{alto}) pode ser usado para formar um cluster de instâncias de entrada pertencentes a uma mesma classe. Portanto, podemos usar a *abordagem baseada em atributo* ao invés de métricas de distância na construção da rede e posterior classificação de novas instâncias. Maiores detalhes serão dados na apresentação da *abordagem baseada em atributo* na próxima seção.

6.3 O PROCESSO DE CONSTRUÇÃO E USO DA REDE NEURAL

Como todo algoritmo de aprendizado automático, o DistAl tem duas fases: a *fase de aprendizado*, quando a rede neural é construída a partir de um conjunto de treinamento e a *fase de classificação*, quando a rede construída é utilizada para a classificação de novas instâncias ou t-uplas de entrada. Segue a descrição de ambas as fases.

6.3.1 Fase de Aprendizado

O DistAl tem duas abordagens específicas: a baseada em distância e a baseada em atributo. A Seção 6.3.1.1 detalha a abordagem baseada em distância a Seção 6.3.1.2 detalha a abordagem baseada em atributo.

6.3.1.1 ABORDAGEM BASEADA EM DISTÂNCIA

Seja E = $\{X^1, X^2, X^3,...,X^N\}$ um conjunto de treinamento constituído de N instâncias descritas pelos vetores X^i, i=1,…,N que representam M classes distintas. Primeiramente o DistAl calcula a distância entre cada par de instâncias de treinamento (usando alguma das métricas mostrada na Seção 6.2) e armazena esses valores em uma matriz D. Dependendo da métrica utilizada, o DistAl enumera a matriz com o número correspondente associado à métrica. A implementação DistAl refinada (ver Anexo C) constrói 22 dessas matrizes, usando diferentes combinações de métricas, articuladas a diferentes tipos de normalização.

As linhas e colunas dessa matriz enumeram instâncias de entradas. Assim, o elemento $D_{me}[k,i]$ da matriz representa a distância (calculada com a métrica de distância me=1,…,22) entre as duas instâncias X^k e X^i respectivamente. Obviamente, a diagonal principal desta matriz é zero.

Cada linha da matriz D_{me} é então ordenada em ordem crescente. Após a ordenação, a linha k contém as distâncias da instância X^k a todas as outras instâncias do conjunto de treinamento. A posição $D_{me}[k,1]$ contém a menor distância e a posição $D_{me}[k,n]$ contém a maior.

A idéia básica do DistAl é criar uma única camada intermediária de neurônios que separa o conjunto de instâncias de treinamento, agrupando-as de acordo com um critério de distância entre elas. Esses neurônios, posteriormente, serão conectados a M neurônios da camada de saída, onde M é o número de classes existente no conjunto de treinamento. Diferente dos outros algoritmos construtivos vistos anteriormente, o DistAl não é limitado ao aprendizado em duas classes apenas.

A representação das instâncias na camada intermediária é linearmente separável. Assim, uma regra iterativa de aprendizado baseada em perceptron pode ser usada no treinamento de vetores de pesos entre os neurônios da camada intermediária e os de saída.

Entretanto tais vetores de pesos podem ser estabelecidos diretamente como segue: os pesos entre os neurônios da camada intermediária e os de saída são escolhidos de maneira tal que cada neurônio da camada intermediária deve 'dominar' o efeito dos neurônios (da camada intermediária) gerados posteriormente. Se existe um total de h neurônios intermediários (numerados 1,2,…,h da esquerda para a direita), então o peso entre o neurônio i da camada intermediária e o neurônio j da camada de saída é estabelecido como 2^{h-i} se o neurônio i exclui padrões pertencentes à classe j e zero caso contrário.

O pseudocódigo que descreve o processo de construção da rede neural, mostrado na Figura 22, usa as seguintes notações, adotadas no algoritmo original:

W_i^h: vetor de pesos entre os neurônios de entrada e o l-ésimo neurônio da camada intermediária

W_m^o: vetor de pesos entre o neurônio de saída correspondente à classe m e os neurônios intermediários

W_{ml}^o: peso entre o l-ésimo neurônio da camada intermediária e o neurônio de saída que representa a classe m

As letras *h* e *o* usadas como sobrescritos são constantes que indicam pesos associados a neurônios da camada intermediária (*h*idden) e a camada de saída (*o*utput) respectivamente.

```
Procedure DistAl(E, me)
{Entradas: E - Conjunto de treinamento com N elementos na forma:
 E^k = (x^k_1, x^k_2, …, x^k_p, C^k ) representando N instâncias pertencentes às
                          classes M = {1,2...m}
 me: identificação da métrica de distância a ser utilizada     }

begin
 leia(me)
 cria_matriz(E,me,D_me)
 ordena(D_me,D)
 i = 0 {inicialização do número de neurônios da camada intermediária}
 while E ≠ ∅ do
 begin
  for all m ∈ M  W^o_m = 2* W^o_m {torna as ligações entre os neurônios
                                   intermediários e os neurônios de saída
                                   duas vezes mais fortes}
  i = i + 1 {introduz mais um neurônio intermediário}

  {Selecionar um subconjunto c maximal de instâncias que pertencem à mesma
  classe. Isso equivale a encontrar uma linha k em D que exclua o maior
  subconjunto de instâncias de E que pertencem à mesma classe m}
 for r = 1,..., N
  begin
    sejam i_r e j_r os índices da coluna (da linha r) na matriz D tal que as
    instâncias correspondentes aos elementos D[r,i_r], D[r,i_r+1],...,D[r, j_r],
    pertencem  a uma mesma classe m e pertencem a E
   c_r ← j_r - i_r + 1 {calcular o número de instâncias deste subconjunto E'}
  end
  k = max c_r {selecionar o r que possui o maior valor correspondente c_r e os
                    correspondetes índices de coluna}
  d^k_inferior = D[k,i_k] {limite inferior para excluir instâncias de E'}
  d^k_superior = D[k,j_k] {limite superior para excluir instâncias de E'}
  W^h_i = X^k
  θ_i,inferior = d^k_inferior
  θ_i,superior = d^k_superior
  E = E - E'
  Atualizar(D) {As linhas que representam instâncias de E' são eliminadas}
  Ligue o novo neurônio intermediário aos de saída com os pesos:
  W^o_mi = 1, W^o_ni = 0 ∀n ≠ m
 end
end
```

Figura 22. Pseudocódigo da Fase de Aprendizado do DistAl baseada em distância

O procedimento *ordena*(D_{me},D) ordena cada uma das linhas de D_{me} em ordem crescente de valor. A matriz D é a matriz resultante deste processo de ordenação. O procedimento *cria_matriz*(E,me,D_{me}) cria uma matriz de distâncias entre as instâncias de E,

usando a métrica de distância identificada pelo número me. Se E = $\{X^1,X^2,....,X^n\}$, $D_{me} = [d_{ij}]_{nxn}$ onde $d_{ij} = d_{me}(X^i,X^j)$.

6.3.1.2 ABORDAGEM BASEADA EM ATRIBUTO

Nesta abordagem, uma matriz D' também é criada e os valores dos atributos das instâncias de treinamento são armazenados nessa matriz. Assim sendo, D'[k,i] representa o valor do i–ésimo atributo da instância de treinamento k. Cada coluna representa um atributo do conjunto de treinamento. O algoritmo de construção dessa matriz é relativamente parecido com aquele da abordagem baseada em distância. As diferenças são as seguintes:

- A matriz representa os próprios elementos, não as distâncias entre as instâncias.
- As colunas da matriz D' são ordenadas, não as linhas.
- O algoritmo de construção da rede tem que encontrar instâncias consecutivas de mesma classe nas colunas, não nas linhas.
- Os neurônios intermediários precisam guardar o nome de um atributo ao invés dos valores de um vetor de pesos.
- A classificação de uma nova instância de treinamento expressa por um vetor X^p é feita analisando os atributos de X^p. Sendo *name* o nome do atributo guardado por um neurônio intermediário H_i e $\theta_{i,inferior}$, $\theta_{i,superior}$ os limites desse neurônio, o valor de saída é determinado da seguinte maneira:

$$O_i^p = 1 \text{ se } \theta_{i,inferior} \leq atr(H_i, X^p) \leq \theta_{i,superior}.$$

$$O_i^p = 0 \text{ caso contrário.}$$

onde $atr(H_i, X^p)$ significa o valor de *name* em X^p.

O pseudocódigo que descreve o processo de construção da rede neural, mostrado na Figura 23, usa a mesma notação do algoritmo anterior exceto por: $name^i$ que é o nome do atributo guardado pelo i-ésimo neurônio intermediário.

```
Procedure DistAl_atr(E)
{Entradas: E - Conjunto de treinamento com N elementos na forma:
 E^k = (x^k_1, x^k_2, …, x^k_p, C^k ) representando N instâncias pertencentes às
                      classes M = {1,2...m}

begin
 cria_matriz(E, D')
 ordena(D', D'')
 i = 0 {inicialização do número de neurônios da camada intermediária}
 while E ≠ ∅ do
 begin
  for all m ∈ M do  W°_m ← 2* W°_m {torna as ligações entre os neurônios
                                    intermediários existentes e os neurônios de
                                    saída duas vezes mais fortes}
   i = i + 1 {introduzir mais um neurônio intermediário}
   {Selecionar um subconjunto c maximal de instâncias que pertencem a uma mesma
   classe. Isso esquivale a encontrar uma coluna a em D'' que exclua o maior
   subconjunto de instâncias de E que pertencem à mesma classe m}
   for c ← 1,..., N do
    begin
      sejam i_c e j_c os índices da linha (da coluna c) para na matriz D'' tal  que
      as instâncias correspondentes aos elementos D''[i_c,c], D''[i_c+1,c], ...,
      D''[j_c,c], pertencem a uma mesma classe m e pertencem a E
      c_c = j_c - i_c + 1 {calcular o número de instâncias deste subconjunto E'}
    end
  a = max c_c {selecionar a que possui o maior valor correspondente c_c e os
                                  correspondentes índices de linha}
  d^a_inferior = D''[i_a,a] {valor mínimo que o atributo pode assumir}
  d^a_superior = D''[j_a,a] {valor máximo que o atributo pode assumir}
  name^i  = nome^a {o neurônio deve guardar o nome do atributo correspondente à
                    coluna  escolhida}
  θ_i,inferior = d^a_inferior
  θ_i,superior = d^a_superior
  E = E - E'
  Atualizar(D'') {as colunas que representam instâncias de E' são eliminadas}
  Ligue o novo neurônio intermediário aos de saída com os pesos:
  W°_mi ← 1, W°_ni ← 0  ∀n ≠ m
 end
end
```

Figura 23. Pseudocódigo da Fase de Aprendizado do DistAl baseada em atributo

O procedimento *cria_matriz*(E,D') cria uma matriz D' com os valores de todos os atributos das instâncias de E. Assim sendo, D'[k,i] armazena o valor o i-ésimo atributo da k-ésima instância de treinamento.

O procedimento *ordena*(D',D'') ordena cada uma das colunas de D' em ordem crescente de valores. A matriz D'' é a matriz resultante dessa ordenação.

6.3.2 A Fase de Classificação

Como todo algoritmo de aprendizado automático, uma vez que o conceito é aprendido (resultado da fase de treinamento), ele é testado com relação à precisão, em um conjunto de teste, cujas instâncias estão já previamente classificadas. O conceito aprendido (neste caso,

a rede induzida) terá então um bom desempenho se a rede conseguir classificar as instâncias de teste na classe à qual elas pertencem.

A precisão do DistAl em um conjunto de teste é determinada da seguinte maneira: cada instância que faz parte do conjunto de teste é entrada para a rede e a saída é calculada pela estratégia *wta* (*winner takes all – vencedor leva todas*). Se existir um ou mais neurônios da camada intermediária que produza o valor 1 (ie. a instância de teste está dentro dos valores de *thresholds* de um ou mais neurônios), a saída é calculada de acordo com a estratégia *wta*. Se nenhum neurônio intermediário produzir o valor 1 para aquela instância, é calculada a distância entre a instância e os *thresholds* de cada neurônio intermediário. O neurônio intermediário que possuir a menor distância será o escolhido para produzir o valor 1. Finalmente, as saídas calculadas são comparadas com as classes associadas as instâncias e a precisão é calculada.

6.4 USANDO DISTAL NO APRENDIZADO E CLASSIFICAÇÃO – DOIS EXEMPLOS

Para exemplificar o uso do sistema DistAl, as duas subseções seguintes mostram em detalhes os passos do algoritmo, com as duas possíveis abordagens, usando como conjunto de treinamento o conjunto conhecido na literatura como 'robôs amigáveis/não amigáveis'. Este conjunto de treinamento está descrito na Tabela 26 e é constituído de seis instâncias que descrevem duas classes (A: amigo e I: inimigo). Este conjunto está disponível em várias referências bibliográficas (ver, por exemplo, [Lavrac & Dzeroski 1994]).

Tabela 26. Conjunto de treinamento 'robô amigo(A)/inimigo(I)'

Exemplo de treinamento	Atributos					
	sorrindo	segurando	gravata	cabeça	corpo	classe
X^1	sim	balão	sim	quadrado	quadrado	A
X^2	sim	bandeira	sim	octogonal	octogonal	A
X^3	sim	espada	sim	redondo	octogonal	I
X^4	sim	espada	não	quadrado	octogonal	I
X^5	não	espada	não	octogonal	redondo	I
X^6	não	bandeira	não	redondo	octogonal	I

6.4.1 Exemplo Usando Abordagem Baseada em Distância

Inicialmente são calculadas as distâncias entre as instâncias de treinamento, de acordo com uma métrica de distância escolhida e a matriz de distância correspondente é construída. Se a métrica de distância escolhida for a métrica (me=15) ie, distância de Manhattan nominal, usando *value difference*, a matriz D_{15} é:

$$D_{15} = \begin{pmatrix} 0 & 0.5 & 0.9 & 0.96 & 1.26 & 1.16 \\ 0.5 & 0 & 0.4 & 0.46 & 0.76 & 0.66 \\ 0.9 & 0.4 & 0 & 0.46 & 0.76 & 0.66 \\ 0.96 & 0.46 & 0.46 & 0 & 0.3 & 0.6 \\ 1.26 & 0.76 & 0.76 & 0.3 & 0 & 0.5 \\ 1.16 & 0.66 & 0.66 & 0.6 & 0.5 & 0 \end{pmatrix}$$

A matriz D, resultante da ordenação em ordem crescente das linhas da matriz D_{15} é mostrada a seguir. Cada elemento D[i,j] da matriz está na forma [A,B,C] onde A denota a distância entre instância i e a instância B; C denota a classe da instância B.

$$D = \begin{pmatrix} [0,1,A] & [0.5,2,A] & [0.9,3,I] & [0.96,4,I] & [1.16,6,I] & [1.26,5,I] \\ [0,2,A] & [0.5,1,A] & [0.4,3,I] & [0.46,4,I] & [0.66,6,I] & [0.76,5,I] \\ [0,3,I] & [0.4,2,A] & [0.46,4,I] & [0.66,6,I] & [0.76,5,I] & [0.9,1,A] \\ [0,4,I] & [0.3,5,I] & [0.46,2,A] & [0.46,3,I] & [0.6,6,I] & [0.96,1,A] \\ [0,5,I] & [0.3,4,I] & [0.5,6,I] & [0.76,2,A] & [0.76,3,I] & [1.26,1,A] \\ [0,6,I] & [0.5,5,I] & [0.6,4,I] & [0.66,2,A] & [0.66,3,I] & [1.16,1,A] \end{pmatrix}$$

A instância X^1, que pertence à classe A, exclui o número máximo de instâncias da mesma classe (E' = $\{X^3,X^4,X^6,X^5\}$). Um neurônio intermediário é criado, com peso W_1^h = X^1 e com os limites $\theta_{inferior}$ = 0.9 e $\theta_{superior}$ = 1.26. Esse neurônio da camada intermediária vai se conectar com o neurônio de saída associado à classe I com valor 1 (W_{I1}^o=1) e vai se conectar com os demais neurônios de saída (no caso, apenas o neurônio associado com a classe A) com valor 0 (W_{A1}^o= 0). O subconjunto do conjunto de treinamento, E' = $\{X^3,X^4,X^6,X^5\}$é eliminado; o novo conjunto E é atualizado para $\{X^1,X^2\}$. A matriz D é atualizada após a remoção de todos os elementos relacionados com as instâncias 3, 4, 6 e 5 existentes na matriz. Uma vez que $E \neq \varnothing$ o processo continua. Os pesos entre os neurônios intermediários e de saída são duplicados. Novamente, X^1 exclui o número máximo de

instâncias de mesma classe (E' = $\{X^1, X^2\}$). Um novo neurônio intermediário é adicionado com peso $W_2^h = X^1$ e com os limites $\theta_{inferior} = 0$ e $\theta_{superior} = 0.5$. Esse neurônio da camada intermediária vai se conectar com o neurônio de saída da classe A com valor 1 ($W_{A2}^o = 1$) e vai se conectar com o outro neurônio de saída com valor 0 ($W_{I2}^o = 0$). O processo pára, pois não existe mais nenhuma instância de entrada para ser classificada, uma vez que E – E' = ∅. A Figura 24 mostra o processo de construção da rede.

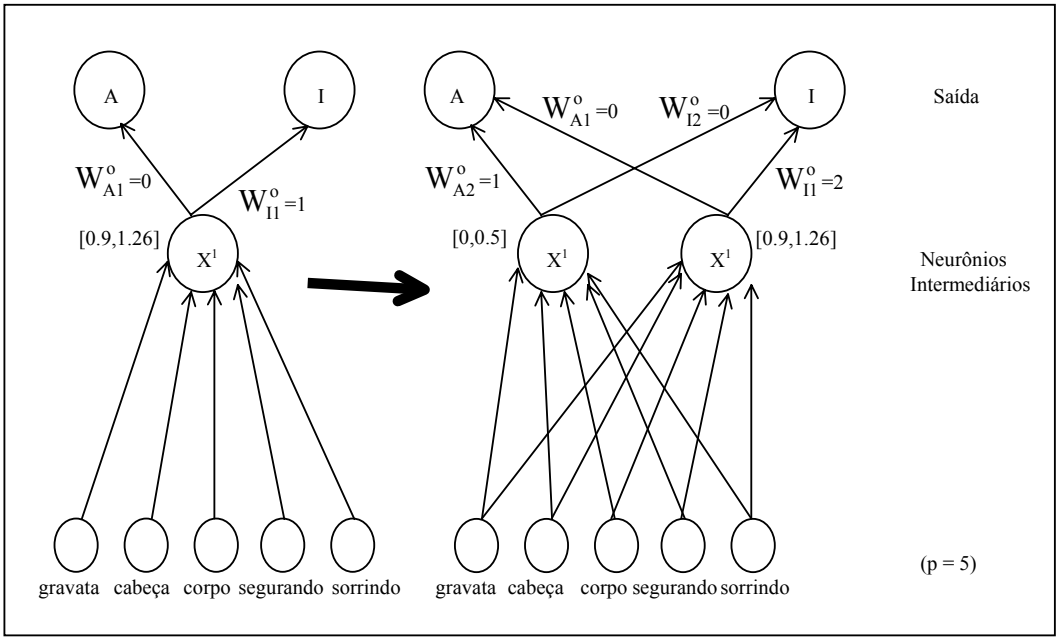


Figura 24. Construção da rede pelo DistAl para o conjunto de treinamento da Tabela 26

6.4.2 Exemplo Usando Abordagem Baseada em Atributo

O conjunto de treinamento 'robôs amigáveis/não amigáveis' não pode ser treinado pela abordagem baseada em atributo do algoritmo DistAl uma vez que não possui valores numéricos. Optamos, entretanto, por construir um novo conjunto de treinamento chamado 'robôs amigáveis/não amigáveis numérico' tendo como base o conjunto original. Neste novo conjunto, os valores dos atributos 'sim' e 'não' foram transformados nos valores numéricos 0 e 1 respectivamente. Os valores 'balão', 'bandeira' e 'espada' foram transformados em 0, 1 e 2 respectivamente. Os valores 'quadrado', 'octogonal' e 'redondo' foram transformados em 0, 1 e 2 respectivamente. A Tabela 27 mostra o novo conjunto de treinamento.

Tabela 27. Conjunto de treinamento 'robô amigo(A)/inimigo(I) numérico'

Exemplo de treinamento	Atributos					
	sorrindo	segurando	gravata	cabeça	corpo	classe
X^1	0	0	0	0	0	A
X^2	0	1	0	1	1	A
X^3	0	2	0	2	1	I
X^4	0	2	1	0	1	I
X^5	1	2	1	1	2	I
X^6	1	1	1	2	1	I

A matriz D' construída a partir desse conjunto de treinamento é:

$$D' = \begin{pmatrix} 0\,0\,0\,0\,0 \\ 0\,1\,0\,1\,1 \\ 0\,2\,0\,2\,1 \\ 0\,2\,1\,0\,1 \\ 1\,2\,1\,1\,2 \\ 1\,1\,1\,2\,1 \end{pmatrix}$$

A matriz D'', resultante da ordenação em ordem crescente das colunas da matriz D' é mostrada a seguir. Cada elemento D'[i,j] da matriz está na forma [A,B,C] onde A denota o valor do atributo pertencente à instância B; C denota a classe da instância B.

$$D''= \begin{pmatrix} [0,1,A] & [0,1,A] & [0,1,A] & [0,1,A] & [0,1,A] \\ [0,2,A] & [1,2,A] & [0,2,A] & [0,4,I] & [1,2,A] \\ [0,3,I] & [1,6,I] & [0,3,I] & [1,2,A] & [1,3,I] \\ [0,4,I] & [2,3,I] & [1,4,I] & [1,5,I] & [1,4,I] \\ [1,5,I] & [2,4,I] & [1,5,I] & [2,3,I] & [1,6,I] \\ [1,6,I] & [2,5,I] & [1,6,I] & [2,6,I] & [2,5,I] \end{pmatrix}$$

O atributo 'sorrindo' (relativo à primeira coluna) é um dos que exclui o número máximo de instâncias da mesma classe (E' = $\{X^3,X^4,X^5,X^6\}$). Um neurônio intermediário é criado, com valor $name^1$ = 'sorrindo' e com os limites $\theta_{inferior} = 0$ e $\theta_{superior} = 1$. Esse neurônio da camada intermediária vai se conectar com o neurônio de saída associado à classe I com valor 1 ($W^o_{I1}=1$) e vai se conectar com os demais neurônios de saída (no caso, apenas o neurônio associado com a classe A) com valor 0 ($W^o_{A1}= 0$). O subconjunto do conjunto de treinamento, E' = $\{X^3,X^4,X^5,X^6\}$ é eliminado; o novo conjunto E é atualizado

para $\{X^1,X^2\}$. A matriz D'' é atualizada após a remoção de todos os elementos relacionados com as instâncias 3, 4, 5 e 6 existentes na matriz.

Uma vez que $E \neq \varnothing$ o processo continua. Os pesos entre os neurônios intermediários e de saída são duplicados. Novamente, o atributo 'sorrindo' exclui o número máximo de instâncias de mesma classe ($E' = \{X^1,X^2\}$). Um novo neurônio intermediário é adicionado com valor $name^2$ = 'sorrindo' e com os limites $\theta_{inferior}$ e $\theta_{superior} = 0$. Esse neurônio da camada intermediária vai se conectar com o neurônio de saída da classe A com valor 1 ($W^o_{A2} = 1$) e vai se conectar com o outro neurônio de saída com valor 0 ($W^o_{I2} = 0$). O processo pára, pois não existe mais nenhuma instância de entrada para ser classificada, uma vez que $E - E' = \varnothing$.

6.5 CONSIDERAÇÕES FINAIS SOBRE O ALGORITMO DISTAL

Como discutido anteriormente, o algoritmo DistAl é diferente dos outros algoritmos neurais construtivos particularmente em dois aspectos. Primeiro, o algoritmo não usa processos iterativos para gerar neurônios ou vetores de pesos (como é o caso do perceptron e seus derivados). Ao invés disso, o algoritmo calcula a distância entre cada par de instâncias de treinamento e, a partir dessas distâncias, constrói a rede. Note que as distâncias são calculadas apenas uma vez para cada par de instâncias e, com as distâncias calculadas, o processo de construção da rede é bastante rápido se comparado a outros algoritmos neurais construtivos. Isso torna o algoritmo especialmente atrativo para tarefas de grande porte, onde velocidade é um fator importante. Segundo, o DistAl usa neurônios esféricos e não neurônios lógicos. Assim sendo, a rede neural gerada pelo DistAl identifica conjuntos ou clusters de instâncias de entrada confinadas em uma determinada região do espaço.

Como o algoritmo se baseia em distâncias para construir a rede, a eficiência do algoritmo depende muito da métrica de distância utilizada. Por esse mesmo motivo, o algoritmo é bastante sensível à presença de dados inexistentes, irrelevantes ou incorretos. Assim sendo, é necessário escolher a métrica de distância mais adequada para um determinado problema antes da construção da rede. Uma maneira simples de se escolher a métrica é executar o sistema e analisar as redes geradas pelo sistema para todas as métricas implementadas. É evidente que esse processo pode consumir muito tempo computacional,

principalmente em problemas de grande porte, onde, teoricamente, o algoritmo DistAl é mais eficiente justamente devido à sua alta velocidade.

Uma outra desvantagem do DistAl é a quantidade de memória necessária para se manter a matriz. A quantidade de memória necessária cresce quadraticamente com o aumento de número de instâncias de treinamento. Assim sendo, o algoritmo DistAl necessita de muito mais memória do que outros algoritmos neurais construtivos, que apenas necessitam de memória para guardar alguns neurônios e vetores de pesos.

CAPÍTULO 7. O ALGORITMO CASCADE-CORRELATION

O cascade-correlation [Fahlman & Lebiere 1990] é um algoritmo construtivo que, durante o aprendizado, adiciona neurônios intermediários à rede, quando necessário. Assim sendo, a topologia de rede é criada dinamicamente durante a fase de treinamento. As principais motivações que levaram à sua criação foram problemas e limitações existentes com o backpropagation [Rumelhart et al 1986], principalmente os relacionados ao tempo de treinamento.

Uma diferença importante entre o algoritmo cascade-correlation e os demais algoritmos estudados nesse documento, exceto o DistAl, é que o cascade-correlation não necessariamente usa o algoritmo perceptron (ou uma variação como o pocket com modificação ratchet) como base para treinar vetores de peso. Ao invés disso, ele pode treinar os seus vetores de peso por meio de 'gradiente ascendente' e 'gradiente descendente' usando um algoritmo de aprendizado chamado quickprop [Fahlman 1988] que, sob certos aspectos, é parecido com o método convencional backpropagation. De acordo com [Parekh 1998], "apesar do algoritmo cascade-correlation ser bem mais rápido do que o algoritmo backpropagation, ele ainda usa um método muito caro computacionalmente para se treinar vetores de peso se comparado a outros algoritmos construtivos." A Seção 7.1 detalha o algoritmo quickprop e a Seção 7.2 detalha o funcionamento do cascade-correlation.

7.1 A REGRA DELTA E O ALGORITMO QUICKPROP

Supondo que se queira minimizar uma função de erro expressa por:

$$\text{Erro} = \frac{1}{2}\sum_{k}(C^{k} - O^{k})^{2} \quad \text{e} \quad O^{k} = f(E^{k}.W)$$

onde C^k é a classe do exemplo de treinamento E^k e O^k é a saída da rede para esse exemplo. O valor de O^k é encontrado por meio de uma função *f* de E^k e W. Usando a regra Delta, o vetor do peso W é modificado, a cada passo, com o objetivo de diminuir o erro, da seguinte maneira:

$$\Delta W'_j = \Delta W_j + W_j \quad \text{e} \quad \Delta W_j = -n \frac{\partial \text{Erro}}{\partial W_j}$$

Essa derivada pode ser expandida na seguinte expressão:

$$\Delta W_j = n \sum_k (C^k - O^k).f'(E^k.W).E_j^k$$

onde n é uma constante de aprendizado, *f'* é a função derivada de *f*. E_j^k é o valor do atributo j no exemplo de entrada E^k.

Assim sendo, pode-se treinar o vetor de pesos W por meio de um 'gradiente descendente' minimizando o erro, usando a regra Delta. Esse modo de treinar o vetor de pesos é usado pelo algoritmo backpropagation.

É interessante ressaltar que quanto maior for o valor da constante n, maior será a modificação no vetor de pesos W. Assim sendo, o aprendizado tende a ser mais rápido conforme se usa maiores valores para n. Um valor muito grande, entretanto, pode fazer com que a mudança no vetor de pesos W seja demasiadamente grande, piorando, desse modo, o aprendizado.

Uma maneira de acelerar o processo de treinamento do vetor de pesos W é usar o algoritmo quickprop. Sendo

$$S_j = \Delta W_j = n \sum_k (C^k - O^k).f'(E^k.W).E_j^k$$

então o algoritmo quickprop modifica o vetor de pesos W da seguinte maneira:

$$\Delta' W_j(t) = \begin{cases} \dfrac{S_j(t)}{S_j(t-1) - S_j(t)} \Delta' W_j(t\text{-}1) & \text{se } t > 0 \\ S_j(t) & \text{caso contrário} \end{cases}$$

onde t é uma variável que indica o tempo.

7.2 FUNCIONAMENTO DO CASCADE-CORRELATION

As informações sobre o algoritmo, descritas a seguir, foram extraídas, principalmente, das referências [Fahlman & Lebiere 1990] e [Campbell 1997]. A referência [Palma Neto & Nicoletti 2004] apresenta uma descrição detalhada do cascade-correlation.

Durante o aprendizado, o algoritmo cascade-correlation adiciona um neurônio intermediário à rede de cada vez e não muda esse neurônio depois dele ter sido adicionado. Cada neurônio intermediário está conectado a todos os neurônios intermediários anteriores, bem como aos neurônios da camada de entrada. O objetivo dos neurônios intermediários é corrigir os erros residuais que ocorreram na saída. Assim sendo, o foco durante o treinamento é no valor absoluto da correlação entre os valores do neurônio intermediário e o erro residual encontrado na saída. Se o neurônio intermediário correlacionar positivamente com o erro residual pode-se usar um peso negativo entre este neurônio e a saída; se a correlação for negativa, o peso usado é positivo.

O algoritmo começa o treinamento com uma rede sem neurônios intermediários, com uma camada de entrada formada por tantos neurônios quanto os atributos que descrevem as instâncias de treinamento, mais um neurônio associado ao termo constante *bias*, um neurônio inicial e um vetor de pesos W qualquer associado a esse neurônio. Esse vetor de pesos W pode ser escolhido aleatoriamente ou, então, iniciado com um valor constante.

O primeiro passo do algoritmo é treinar o vetor de pesos W (que representa o peso da conexão entre entrada e saída), usando algum algoritmo de treinamento. As entradas para esse vetor de pesos são os atributos que descrevem as instâncias de treinamento mais o termo constante *bias*. Pode-se usar o algoritmo perceptron (ou umas das suas variações), a regra Delta (Seção 7.1) ou algum outro algoritmo para treinamento de redes para uma única

camada. Na proposta original do algoritmo cascade-correlation [Fahlman & Lebiere 1990], no entanto, é usado o algoritmo quickprop (Seção 7.1).

Usando o algoritmo quickprop ou a regra Delta, o 'gradiente descendente' termina quando o vetor de pesos W não melhora após um certo número de passos (ie, erro não diminui), controlado por um 'nível de paciência'.

Ao término do treinamento, verifica-se a precisão do vetor de pesos W gerado durante o processo, na entrada E. Se a precisão for maior do que a precisão mínima tolerável, então, o algoritmo termina sem neurônios intermediários. Caso contrário, é necessário adicionar à rede um neurônio intermediário com o objetivo de corrigir os erros ocorridos. Antes de se adicionar esse neurônio, no entanto, é necessário calcular o erro residual de cada exemplo de entrada E^k, o que é feito da seguinte maneira:

$$\text{Erro\_res}^k = (C^k - O^k) \text{ e } O^k = f(E^k.W)$$

onde C^k é a classe do exemplo de treinamento E^k e O^k é a saída da rede para esse exemplo. O^k é encontrada através de uma função de E^k e W. Também é necessário guardar a média aritmética desses valores numa variável aqui chamada de erro_médio.

O neurônio intermediário que irá ser adicionado terá como entrada os valores dos atributos e o termo constante *bias*. Novamente, os valores para o vetor de pesos desse neurônio podem ser encontrados aleatoriamente ou podem ser iniciados com um valor constante. O vetor de pesos Wh desse neurônio é treinado para maximizar a correlação entre a saída desse neurônio intermediário e o erro residual encontrado anteriormente. Assim sendo, faz-se um 'gradiente ascendente' para maximizar a correlação S, que é definida da seguinte maneira:

$$S = \left| \sum_k (ON^k - \text{ON\_médio})(\text{Erro\_res}^k - \text{erro\_médio}) \right| \text{ e } ON^k = f(E^k.Wh)$$

onde ON^k é a saída do neurônio intermediário e ON_médio é a média aritmética dos valores de ON^k. As mudanças no vetor de pesos Wh são, então, feitas da seguinte maneira, usando a regra Delta:

$$\Delta Wh_j = n \frac{\partial S}{\partial Wh_j} = n \, \text{Sinal} \sum_k (\text{Erro\_res}^k - \text{erro\_médio}).f'(E^k.Wh).E_j^k$$

onde Sinal é o sinal da correlação S, ie:

$$\text{Sinal} = \begin{cases} 1 \text{ se } \sum_k (\text{ON}^k - \text{ON\_médio})(\text{Erro\_res}^k - \text{erro\_médio}) \geq 0 \\ -1 \text{ caso contrário} \end{cases}$$

ou pode-se usar o algoritmo quickprop para modificar o vetor de pesos Wh, com

$$S_j = \Delta Wh_j = n \, \text{Sinal} \sum_k (\text{Erro\_res}^k - \text{erro\_medio}).f'(E^k.Wh).E_j^k$$

O 'gradiente ascendente' termina quando o vetor de pesos Wh não melhora após um certo número de passos (ie, S não aumenta), controlado por um 'nível de paciência'. O vetor de pesos Wh gerado (aquele que gerou o maior valor para S) será 'congelado' e usado posteriormente. Note que é necessário algum algoritmo que se baseia no 'gradiente' para se treinar o vetor de pesos Wh. Assim sendo, pode-se usar a regra Delta ou o quickprop, mas não é possível usar o algoritmo perceptron ou uma das suas variações por exemplo.

O próximo passo do algoritmo é treinar novamente o vetor de pesos W do neurônio inicial. Novamente, isso pode ser feito usando algum algoritmo de treinamento para uma única camada ou fazer novamente um 'gradiente descendente' minimizando o erro, como descrito na Seção 7.1, usando o algoritmo quickprop ou a regra Delta. Agora, no entanto, usa-se uma dimensão a mais como entrada: a saída do neurônio intermediário gerado, ie, $f(E^k.Wh)$. Se depois do treinamento de W a precisão for maior que a precisão mínima tolerável então o algoritmo termina. Caso contrário, cria-se um neurônio intermediário e faz-se um 'gradiente ascendente' maximizando a correlação, como descrito anteriormente, mas com uma entrada extra: a saída do neurônio intermediário anterior, ie, $f(E^k.Wh)$. O processo recomeça novamente, mas agora com duas entradas extras e assim por diante até que a precisão mínima tolerável seja atingida ou algum critério de parada (como número máximo de neurônios intermediários) seja satisfeito.

A Figura 25 mostra a arquitetura de uma rede cascade-correlation com dois neurônios intermediários e dois atributos de entrada, além do termo constante *bias*. As flechas contínuas representam pesos que não mudam e as fechas pontilhadas representam pesos que mudam em cada treinamento de W.

Ao se fazer um 'gradiente ascendente', como descrito anteriormente, é possível gerar um conjunto de neurônios intermediários e não apenas um neurônio. Cada neurônio terá diferentes valores iniciais para os seus vetores de peso. No final do 'gradiente ascendente', o melhor neurônio intermediário (aquele que o seu vetor de pesos gerou o maior valor para a correlação S) é 'congelado' e o resto descartado. Esse procedimento é bastante eficiente em máquinas paralelas (pois se pode treinar o conjunto de vetores de peso paralelamente, ganhando tempo). Também é possível usar diferentes funções em diferentes neurônios do conjunto, por exemplo, usar funções senoidais em alguns neurônios e funções gaussianas em outros neurônios.

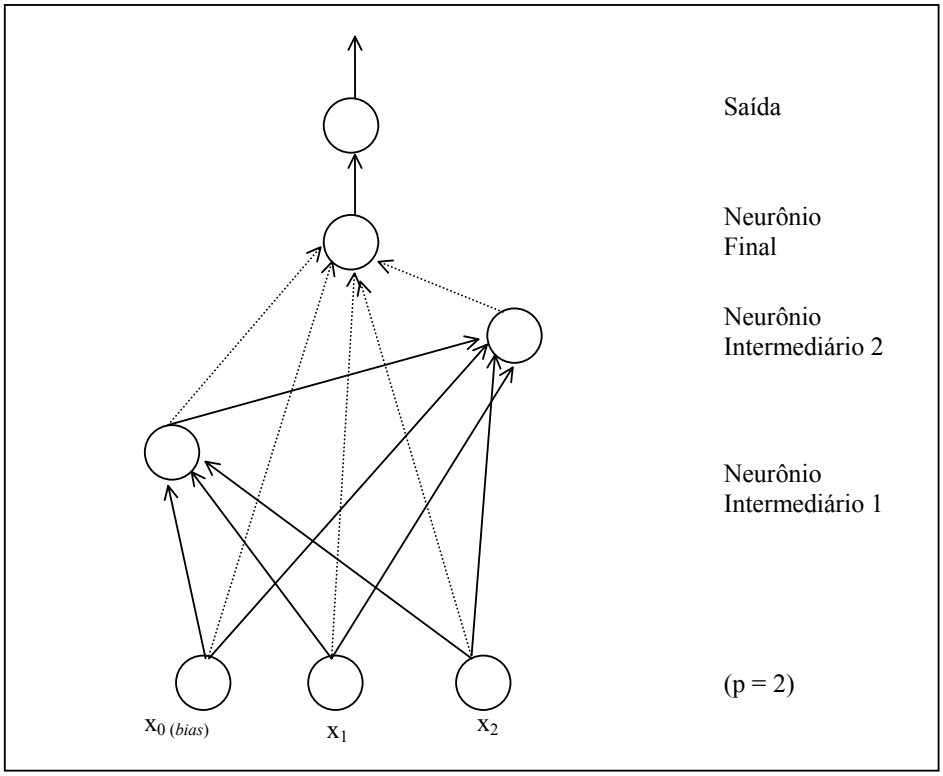


Figura 25. Arquitetura de uma rede cascade-correlation

Na proposta original do cascade-correlation descrita em [Fahlman & Lebiere 1990], a função usada no algoritmo é a tangente hiperbólica, ie, uma função sigmóide simétrica, que varia entre −1 e 1. Em [Squires & Shavlik 1991], é usada a função:

$$f(x) = \frac{1}{1 - e^{-x}} - 0.5$$

que varia de –0.5 a 0.5.

Existem outras variações do cascade-correlation. Existe, por exemplo, uma versão que usa mais de um neurônio intermediário em cada camada. Consultar [Campbell 1997] para maiores detalhes. As Figuras 26, 27 e 28 mostram o pseudocódigo do cascade-correlation, usando 'gradiente descendente' e 'gradiente ascendente' com o algoritmo quickprop para treinar os vetores de peso.

```
Inicialização de variáveis globais:
camada = 0 {representa o número de camadas/neurônios intermediários}
MAX = número máximo de neurônios intermediárias da rede
intermediários[MAX] {um vetor de vetores de peso que guarda os
                     neurônios intermediários. Também é uma
                     parte da saída do sistema}
Winicial {vetor de pesos inicial. É parte da saída}
PMT = precisão mínima tolerável {usuário define a precisão mínima
                                 tolerável }
paciencia = nível de paciência { nível de paciência }
procedure cascade_correlation(E)
{Entradas: E conjunto de treinamento com n elementos da forma:
 E^k = (x^k_0, x^k_1, x^k_2, …, x^k_p, C^k) representando n instâncias pertencentes
 às classes C^k = {1,-1}                                                    }
begin
 acha_w_descendente(E,W) {Executa um "gradiente descendente" minimizando
                          o erro para gerar o vetor de pesos W}
 if((precisao(E, W) < PMT) and (camada < MAX)) then
  begin
  camada = camada + 1
  acha_erro_residual(E,W,Erro_res,erro_medio) {Encontra o erro residual
                                               para cada E^k e o erro médio)
  acha_w_ascendente(E,W_aux,Erro_res, erro_medio){Executa um "gradiente
                                               ascendente"maximizando
                                               a correlação para achar o
                                               vetor de pesos W_aux }
  intermediários[camada] = W_aux  {guarda o vetor de pesos do neurônio
                                   intermediário da camada atual}
  modifica_cascade(E,E1,W_aux)     {expande o conjunto de treinamento}
  cascade_correlation(E1)
  end
 else
  Winicial = W {Vetor de pesos inicial ( que é parte da saída do
                sistema) recebe o vetor de pesos atual e o
                algoritmo termina}
end
procedure modifica_cascade(E,E1,W)
begin
 i = camada + p          {índice de entrada equivalente à saída do
                          último neurônio intermediário }
 E1 = E
 for all E1^k ∈ E1 do
    E1^k_i = função(E^k.W)
end
```

Figura 26. Pseudocódigo do algoritmo cascade-correlation

```
procedure acha_w_descendente (E,melhor_w)
begin
 melhor_w = W = acha_w_inicial
 contador = 0
 while (contador < paciência}
 begin
  for all E^k ∈ E do {para cada exemplo de treinamento}
   for all E^k_p ∈ E^k do {para cada atributo do exemplo de treinamento}
    W_p = W_p + quickprop_erro(W,E^k,p, ultimo_delta_w, ultimo_S)
  if(calcula_erro(E, W) < calcula_erro(E, melhor_w)) then
    melhor_w = W
  else
   contador = contador + 1
 end
end
```

Figura 27. Pseudocódigo do algoritmo cascade-correlation (cont.)

```
procedure acha_w_ascendente (E,melhor_w,erro,erro_médio)
begin
 melhor_w = W = acha_w_inicial
 contador = 0
 while(contador < paciência}
 begin
  for all E^k ∈ E do {para cada exemplo de treinamento}
   for all E^k_p ∈ E^k do {para cada atributo do exemplo de treinamento}
    W^p = W^p + quickprop_correlação (W, E^k, p, erro^k, erro_médio,
                                          ultimo_delta_w, ultimo_S)
  if(calcula_correlacao(E,W,erro^k,erro_médio)>calcula_correlacao(E,
                                    melhor_w,erro^k, erro_médio)) then
    melhor_w = W
  else
   contador = contador + 1
 end
end
```

Figura 28. Pseudocódigo do algoritmo cascade-correlation (cont.)

O procedimento *função*(W,E) é a implementação da função usada durante o algoritmo cascade-correlation e o procedimento *função_derivada*(W,E) é a derivada dessa função.

O procedimento *acha_w_descendente*(E,melhor_w) encontra um vetor de pesos denominado melhor_w através de um 'gradiente descendente' feito no conjunto de treinamento E. O vetor de pesos melhor_w será aquele que gerar o menor valor de erro entre todos os vetores de peso encontrados durante o 'gradiente descendente'. Para se achar

melhor_w, o procedimento faz uso de um outro procedimento denominado *calcula_erro*(E,W) que irá encontrar o valor do erro no conjunto de treinamento E para o vetor de peso W.

O procedimento *quickprop_erro*(W,E^k,p,ultimo_delta_w,ultimo_S) acha a variação do vetor de pesos W. Para tal, o procedimento usa o algoritmo quickprop, definido na Seção 7.1. As variáveis ultimo_delta_w e ultimo_S são usadas no início do procedimento e depois atualizadas no final para serem, então, usadas na próxima vez que o procedimento for chamado.

Similarmente, o procedimento *acha_w_ascendente*(E, melhor_w, erro, erro_médio) encontra um vetor de pesos denominado melhor_w através de um 'gradiente ascendente' feito no conjunto de treinamento E. O vetor de pesos melhor_w será aquele que gerar o maior valor de correlação entre todos os vetores de peso encontrados durante o 'gradiente ascendente'. Para se achar melhor_w, o procedimento faz uso de um outro procedimento denominado *calcula_correlação*(E,W,erro,erro_médio) que irá encontrar o valor do módulo da correlação no conjunto de treinamento E para o vetor de peso W. Note que o valor do erro residual para cada exemplo de treinamento E^k e a média desses valores (encontrados anteriormente pelo procedimento *acha_erro_residual*) são usados tanto no *acha_w_ascendente* como no *calcula_correlação*.

O procedimento *quickprop_correlação*(W,E^k,p,$erro^k$,erro_médio, ultimo_delta_w, ultimo_S) acha a variação do vetor de pesos W. Para tal, o procedimento usa o algoritmo quickprop, definido na Seção 7.1. As variáveis ultimo_delta_w e ultimo_S são usadas no início do procedimento e depois atualizadas no final para serem, então, usadas na próxima vez que o procedimento for chamado.

O procedimento *acha_w_inicial* retorna um vetor de pesos W qualquer. Os valores de W podem ser encontrados aleatoriamente ou podem ser iniciados com um valor constante.

O procedimento *precisao*(E,W) calcula a precisão do vetor de pesos W no conjunto de treinamento E. Para classificação em duas categorias 1 e –1, um exemplo de treinamento E^k com classe positiva estará classificado corretamente se WE^k for maior do que zero e um exemplo de treinamento E^k com classe negativa estará classificado corretamente se WE^k for menor ou igual à que zero.

O procedimento *modifica_cascade*(E,E1,W) expande o conjunto E num novo conjunto E1 por meio da introdução de uma nova dimensão, que representa o valor do último neurônio intermediário adicionado à rede (que tem W como vetor de pesos). E1 é então usado como conjunto de entrada na próxima camada. A cada camada, os exemplos de treinamento do conjunto E1 são expandidos em uma dimensão a mais. Note que o procedimento *modifica_cascade* é bastante parecido com o procedimento *modifica2* do algoritmo pyramid (ver Capítulo 3, Seção 3.2).

7.3 EXEMPLO DE USO DO CASCADE-CORRELATION

Para exemplificar o funcionamento do algoritmo cascade-correlation, nesta seção ele é usado para o aprendizado do conceito de paridade-2 (XOR). Os exemplos de treinamento E^k estão definidos na Tabela 17 da Seção 5.5.1. O treinamento dos vetores de peso dos neurônios intermediários é feito através de um 'gradiente ascendente' com o algoritmo quickprop ou a regra Delta, como descrito anteriormente. A função usada no algoritmo e a sua derivada são respectivamente:

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

onde tanh(x) significa tangente hiperbólica de x. Será usada a precisão mínima tolerável de 100%, ie, o algoritmo só terminará quando todos exemplos de treinamento estiverem corretos.

O primeiro passo do algoritmo é treinar o vetor de pesos W inicial (que representa os pesos das conexões entre entrada e saída) através de um algoritmo qualquer para uma única camada. Assim sendo, um possível valor encontrado pode ser W = <0 0 0>. Esse vetor de pesos gerado classifica corretamente apenas duas das quatro instâncias de treinamento (E^2, E^3), e, portanto, será adicionado um neurônio intermediário com o objetivo de melhorar a classificação. O próximo passo é, então, treinar o vetor de pesos Wh desse neurônio intermediário através de um 'gradiente ascendente' maximizando a correlação. Suponha que o vetor obtido seja Wh = <0 0 1>.

Tendo gerado o primeiro neurônio intermediário e o seu vetor de pesos Wh, então é necessário encontrar novamente o vetor de pesos W inicial através de um algoritmo

qualquer para uma única camada. Agora, no entanto, teremos que adicionar uma dimensão extra ao conjunto de treinamento E. Essa dimensão extra será o valor de $f(E^k.Wh)$. A Tabela 28 mostra o valor da dimensão extra para cada exemplo de treinamento e a Tabela 29 mostra o novo conjunto de treinamento.

Tabela 28. Valores para a dimensão extra

Exemplo	$x = E^k.Wh$	$f(x) = \tanh(x/2)$
E^1	0	0
E^2	1	0.4
E^3	0	0
E^4	1	0.4

Tabela 29. Primeiro novo conjunto de treinamento

Exemplo	x_0 (bias)	x_1	x_2	x_3 ($f(E^k.Wh)$)	Classe
E^1	1	0	0	0	–1
E^2	1	0	1	0.4	1
E^3	1	1	0	0	1
E^4	1	1	1	0.4	–1

Treinando novamente o vetor de pesos W, usando o conjunto de treinamento da Tabela 29, um possível valor pode ser W = <–4.4 –8.9 7.1 3.4>. Note que esse vetor classifica corretamente três das quatro instâncias de treinamento (E^1, E^2, E^4).

Assim sendo, é necessário adicionar mais um neurônio intermediário e treinar o seu vetor de pesos através de um 'gradiente ascendente' maximizando a correlação. Esse vetor de pesos será treinado no conjunto de treinamento descrito na Tabela 29. Um possível valor para o vetor pode ser Wh_2 = <–7.5 15.9 –14.4 –4.5>.

Tendo gerado o segundo neurônio intermediário e o seu vetor de pesos Wh_2, então é necessário encontrar novamente o vetor de pesos W inicial. Agora, no entanto, teremos que adicionar mais uma dimensão extra ao conjunto de treinamento da Tabela 29. Essa dimensão extra será o valor de $f(E^k.Wh_2)$. A Tabela 30 mostra o novo conjunto de treinamento.

Tabela 30. Segundo novo conjunto de treinamento

Exemplo	x_0 (bias)	x_1	x_2	x_3 ($f(E^k.Wh)$)	x_4 ($f(E^k.Wh_2)$)	Classe
E^1	1	0	0	0	–0.9	–1
E^2	1	0	1	0.4	–0.9	1
E^3	1	1	0	0	0.9	1
E^4	1	1	1	0.4	–0.9	–1

Treinando novamente o vetor de pesos W, usando o conjunto de treinamento da Tabela 30, um possível valor pode ser: W = <6.9 –13.7 11.2 6.0 14.1>. Esse vetor classifica corretamente todas as instâncias e o algoritmo termina.

CAPÍTULO 8. AVALIAÇÃO EMPÍRICA DOS ALGORITMOS

Esse capítulo trata da avaliação empírica dos diferentes algoritmos neurais construtivos, usando suas implementações disponibilizadas no sistema CONEB (ver Anexo D). As experimentações com o algoritmo Distal foram realizadas com a implementação enviada pelo autor do método, alterada em alguns aspectos (ver Introdução e Anexo C). Para os experimentos foram escolhidos quatro domínios de conhecimento a saber: Vestibular, Paridade-5, Íris e Monks (versões 1, 2 e 3). O domínio Íris foi desdobrado em três domínios (Íris1, Íris2 e Íris3) por razões discutidas na Seção 8.3. A Tabela 31 identifica e mostra as características básicas de cada um dos domínios.

Tabela 31. Principais características dos domínios de conhecimento

Domínios de Conhecimento	Número de Atributos	Número Total de Instâncias	Número de Instâncias Positivas	Número de Instâncias Negativas	Número de Classes	Artificial ou Real
Vestibular	6	198	101	97	2	Real
Paridade-5	5	32	16	16	2	Artificial
Ìris1	4	100	50	50	2	Real
Ìris2	4	100	50	50	2	Real
Ìris3	4	100	50	50	2	Real
Monks1	6	432	216	216	2	Artificial
Monks2	6	432	142	290	2	Artificial
Monks3	6	432	228	204	2	Artificial

Os domínios Íris, Monks e Paridade-5 são domínios freqüentemente utilizados para a validação de diferentes técnicas de aprendizado de máquina. O domínio Vestibular é um domínio ainda não muito explorado. Foi utilizado para experimentação com algoritmos construtivos em [Palma Neto et al 2003] [Palma Neto et al 2004] e com redes neurais não construtivas em [Volpini et al 2002]. Os domínios Íris, Monks (1, 2 e 3) podem ser obtidos junto ao UCI Machine Learning Repository [Blake & Merz 1998]. Paridade-5 pode ser

facilmente construído, dado o seu número pequeno de instâncias. Os experimentos foram conduzidos observando que:

- Para testar a eficiência dos algoritmos, foi usada a técnica de validação conhecida como *10-fold cross-validation*. Ela consiste em dividir o conjunto original em 10 partes iguais ou quase iguais (quando o número de instâncias do conjunto original não for divisível por 10) e executar o mesmo procedimento, aprendizado-teste 10 vezes, utilizando, a cada vez, 9 dessas partes para treinamento e a parte restante para teste. Os resultados apresentados nas tabelas desse capítulo são as médias aritméticas desses 10 resultados acompanhadas dos seus desvios padrões.
- Os algoritmos baseados em perceptron (todos, exceto o DistAl e cascade-correlation) foram executados com N, $N.10^2$, $N.10^4$ e, em alguns casos, $N.10^6$ iterações. N é o número de instâncias de treinamento usadas em cada execução. Por exemplo, o domínio de conhecimento Paridade-5 contém 32 instâncias sendo que 28 ou 29 são usadas para o treinamento e o restante para teste. Desse modo, a construção da rede foi feita usando 28/29, $28/29.10^2$, $28/29.10^4$ e $28/29.10^6$ iterações.
- O algoritmo cascade-correlation foi executado com nível de paciência com valores 1, 5 e 10.
- Em todos os experimentos, o máximo de 10 camadas foi usado para os algoritmos tower, pyramid, tiling e cascade-correlation e o máximo de 10 neurônios intermediários para o algoritmo DistAl. O algoritmo tiling foi limitado num máximo de 16 neurônios auxiliares por camada.
- Em todos os experimentos, o limite máximo de 64 neurônios auxiliares para o algoritmo upstart foi usado.
- Para o algoritmo cascade-correlation, a constante de aprendizado assumiu o valor 1 no domínio de conhecimento Paridade-5, o valor 0.1 nos domínios Vestibular e Monks (versões 1, 2 e 3) e o valor 0.01 nos domínios Íris1, Íris2 e Íris3.

As tabelas das próximas subseções mostram os resultados obtidos com o uso dos algoritmos nos diferentes domínios. Os dados do pocket com modificação ratchet (PMR)

são fornecidos para referência. Para simplificar a legenda, cada uma das tabelas apenas informa o algoritmo utilizado e em qual domínio de conhecimento.

Cada uma das próximas subseções diz respeito aos experimentos e análise dos resultados em um particular domínio de conhecimento. Nos resultados relativos a cada um dos domínios, os melhores valores de treinamento e teste entre todos os algoritmos estão em negrito nas tabelas.

Algumas métricas de distância do algoritmo DistAl usam algum tipo de normalização. Assim sendo, as letras E, M, e VM, existentes nas tabelas, significam respectivamente Euclideana, Manhattan e Valor Máximo e as letras NL e ND significam normalização baseada em limites e normalização baseada em desvio padrão respectivamente. Por exemplo, a sigla E.NL. significa métrica de distância euclideana com normalização baseada em limites. As métricas Camberra e atr-based (baseada em atributos) não serão usadas nos experimentos das próximas subseções. A justificativa é que os resultados obtidos usando o sistema com alguma dessas duas métricas não foram corretos em alguns domínios de conhecimento, por problemas do próprio software.

8.1 SISTEMA VESTIBULAR

De acordo com [Volpini et al 2002], "a base de dados Vestibular contém 198 instâncias, cada uma delas descrita por seis atributos (três relativos ao olho esquerdo e três relativos ao olho direito) e uma classe associada. Cada instância 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.

Movimentos sacádicos são produzidos quando o 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 frente do mesmo. Para analisar essa situação, alguns eletrodos são posicionados próximos aos olhos (direito e esquerdo) do paciente com o intuito de medir os potencias elétricos produzidos pelos movimentos sacádicos". Pacientes com classe positiva (101) possuem problemas nos seus sistemas vestibulares e pacientes com classe negativa (97) não os possuem. As Tabelas 32 a 37 mostram os resultados dos algoritmos nesse domínio.

Tabela 32. PMR / Vestibular

PMR		
No. de Iterações	Precisão de Treinamento	Precisão de Teste
178/179	89.0~1.0	87.9~5.5
$178/179.10^2$	90.1~0.7	87.3~3.2
$178/179.10^4$	90.0~0.8	88.4~5.0

Tabela 33. Tower e Pyramid / Vestibular

TOWER					
No. de Iterações	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
178/179	89.9~0.9	88.6~2.4	2.3~1.6	1	7
$178/179.10^2$	90.5~0.6	88.9~4.8	2.4~1.0	1	4
$178/179.10^4$	90.6~0.8	88.4~5.0	6.0~2.7	1	7
PYRAMID					
178/179	89.6~0.8	89.4~5.2	2.4~1.2	1	4
$178/179.10^2$	90.5~0.7	88.9~4.8	3.1~1.6	1	6
$178/179.10^4$	90.5~0.8	**90.5~3.9**	6.6~3.9	1	8

Tabela 34. Tiling / Vestibular

TILING					
No. de Iterações	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários Mestres	Menor no. de Neurônios Intermediários Mestres	Maior no. de Neurônios Intermediários Mestres
178/179	89.7~0.9	88.4~5.0	2.1~1.5	1	6
$178/179.10^2$	90.4~0.9	87.9~5.0	2.7~1.8	1	6
$178/179.10^4$	90.8~0.9	89.3~4.0	5.9~2.2	2	6

Tabela 35. Upstart / Vestibular

UPSTART – TREINAMENTO		
No. de Iterações	Precisão de Treinamento	Precisão de Teste
178/179	88.6~1.2	87.9~6.3
$178/179.10^2$	92.3~1.1	87.3~4.7
$178/179.10^4$	93.1~0.7	90.4~5.6
UPSTART – SOMA		
178/179	91.5~1.0	87.9~5.0
$178/179.10^2$	92.3~0.9	88.9~5.3
$178/179.10^4$	92.6~0.6	88.4~5.9

Tabela 36. DistAl / Vestibular

DISTAL					
Métrica de Distância	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
Euclideana	**100.0~0.0**	86.8~7.5	7.9~0.9	7	8
Manhattan	**100.0~0.0**	88.4~8.4	8.2~0.7	7	9
Valor Máximo	**100.0~0.0**	90.0~5.4	8.1~0.8	7	9
Chi square	98.8~0.3	82.6~9.7	9.4~0.6	8	10
Correlation	**100.0~0.0**	88.4~6.1	7.6~0.4	7	8
N-Chi square	**100.0~0.0**	85.7~6.6	9.2~0.9	8	10
Jaccard	**100.0~0.0**	88.9~4.9	7.2~0.6	6	8
Dice	**100.0~0.0**	88.9~4.9	7.2~0.6	6	8
Cosine	99.7~0.5	84.2~5.7	9.5~0.9	8	10
E.NL.	**100.0~0.0**	84.7~12.7	8.0~0.7	7	9
M.NL.	**100.0~0.0**	88.4~6.9	8.2~1.0	7	10
VM.NL.	**100.0~0.0**	86.8~6.3	8.7~0.6	7	9
E.ND.	**100.0~0.0**	87.8~6.6	7.2~0.7	6	8
M.ND.	**100.0~0.0**	85.2~8.7	8.5~1.1	7	10
VM.ND.	**100.0~0.0**	85.7~5.2	8.4~0.6	7	9

Tabela 37. Cascade-Correlation / Vestibular

CASCADE-CORRELATION					
Nível de Paciência	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
1	64.8~16.1	63.3~20.0	5.1~3.2	1	10
5	61.8~14.5	62.9~18.8	6.8~2.8	1	10
10	61.6~13.5	55.5~21.8	6.6~3.2	1	10

Pode ser observado nas Tabelas 32 a 37 que apenas o algoritmo DistAl, usando diversas métricas, conseguiu obter 100% de precisão no conjunto de treinamento. No conjunto de teste, no entanto, a sua melhor precisão foi de 90.0%, inferior à melhor precisão de teste do algoritmo pyramid e do algoritmo upstart-treinamento. No conjunto de treinamento, exceto o DistAl e o cascade-correlation, os desempenhos dos demais algoritmos foram similares, com o algoritmo upstart (nas suas duas abordagens) sendo levemente superior aos demais.

No conjunto de teste, a melhor precisão aconteceu usando o algoritmo pyramid com $178/179.10^4$ iterações. Os desempenhos dos algoritmos, no entanto, foram bastante parecidos também nesse conjunto se considerarmos apenas as métricas com os melhores desempenhos do algoritmo DistAl e desconsiderarmos o algoritmo cascade-correlation.

Nos algoritmos baseados em perceptron, o aumento do número de iterações quase sempre melhorou a precisão de treinamento. Nos algoritmos baseados em perceptron que usam neurônios intermediários, o número de neurônios intermediários aumentou com o aumento do número de iterações. A precisão de teste, no entanto, não necessariamente aumentou. Os algoritmos upstart com abordagem treinamento, tiling e pyramid tiveram melhor desempenho de teste com o maior número de iterações ($178/179.10^4$), mas essa tendência não se observou nos demais algoritmos.

No algoritmo cascade-correlation, o aumento do nível de paciência curiosamente sempre piorou as precisões de treinamento e também de teste. Os resultados do algoritmo cascade-correlation foram fortemente inferiores aos demais algoritmos. É injusto, no entanto, considerar que o algoritmo cascade-correlation é particularmente ruim para esse domínio de conhecimento, uma vez que ele foi testado apenas usando uma função (ver Anexo D). A razão desse baixo desempenho se deve muito provavelmente ao uso dessa função.

O número de neurônios intermediários não variou muito entre os algoritmos tower e pyramid, considerando os mesmos números de iterações. Os números de neurônios intermediários mestres do algoritmo tiling também foram similares aos números do pyramid e tower, novamente considerando os mesmos números de iterações. O tiling, no entanto, também possui neurônios intermediários auxiliares e, desse modo, pode-se considerar que ele necessitou de mais neurônios do que os algoritmos tower e pyramid para construir a sua rede neural. O algoritmo Distal necessitou de mais neurônios do que tower e pyramid usando qualquer uma de suas métricas. O cascade-correlation também necessitou de mais neurônios do que tower e pyramid, exceto quando foi usado nível de paciência 1.

As métricas euclideana, Manhattan e valor máximo com ou sem normalização obtiveram 100% de precisão no conjunto de treinamento. No conjunto de teste, no entanto, as métricas normalizadas Manhattan e valor máximo foram iguais ou inferiores às suas versões sem normalização. A métrica euclideana com normalização baseada em limites foi inferior à sua versão sem normalização, mas a euclideana com normalização baseada em desvio padrão foi superior. Desse modo, podemos concluir que apenas uma das seis métricas normalizadas foi necessária, ie, apenas uma conseguiu ser superior à sua versão sem normalização.

Não se pode concluir qual dos algoritmos foi melhor no domínio Vestibular. Como dito anteriormente, as precisões de teste foram altas e as diferenças entre elas estatisticamente não representativas. A superioridade do algoritmo DistAl no conjunto de treinamento não se reflete no conjunto de teste. O algoritmo pocket com modificação ratchet pode ser a melhor escolha nesse domínio de conhecimento, se simplicidade for o critério, já que o desempenho não está muito aquém do melhor desempenho em testes.

8.2 PARIDADE-5 PAR

O domínio de conhecimento Paridade-5 par é um domínio artificial. O domínio contém 5 atributos e 32 instâncias, sendo 16 positivas e 16 negativas. Esse domínio está descrito na Tabela 11 do Capítulo 3, Seção 3.2.1. As Tabelas 38 a 43 mostram os resultados dos experimentos feitos com os algoritmos nesse domínio.

Tabela 38. PMR / Paridade-5

PMR		
No. de Iterações	Precisão de Treinamento	Precisão de Teste
28/29	50.6~4.0	28.1~20.9
$28/29.10^2$	52.4~2.7	14.9~18.8
$28/29.10^4$	55.5~6.8	15.7~15.8
$28/29.10^6$	63.4~10.1	11.6~18.2

Tabela 39. Tower e Pyramid / Paridade-5

TOWER					
No. de Iterações	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
28/29	51.6~3.3	24.8~16.9	2.5~2.0	1	7
$28/29.10^2$	62.7~14.2	25.7~24.2	5.4~3.3	1	10
$28/29.10^4$	70.4~19.7	25.8~38.8	3.8~2.6	1	10
$28/29.10^6$	91.3~17.5	51.5~36.8	5.2~2.8	1	9
PYRAMID					
28/29	50.9~5.9	34.8~26.3	3.5~2.5	1	7
$28/29.10^2$	67.7~17.6	25.7~32.4	4.5~3.5	1	10
$28/29.10^4$	92.6~15.7	58.2~38.9	4.5~1.7	1	7
$28/29.10^6$	100.0~0.0	61.5~35.0	3.7~2.6	1	8

Tabela 40. Tiling / Paridade-5

TILING					
No. de Iterações	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários Mestres	Menor no. de Neurônios Intermediários Mestres	Maior no. de Neurônios Intermediários Mestres
28/29	53.8~2.6	18.2~18.3	3.7~2.6	1	9
$28/29.10^2$	99.3~2.1	31.4~18.7	4.5~1.9	3	10
$28/29.10^4$	**100.0~0.0**	24.8~22.6	2.8~0.4	2	3
$28/29.10^6$	**100.0~0.0**	20.7~24.8	2.4~0.5	1	3

Tabela 41. Upstart / Paridade-5

UPSTART – TREINAMENTO		
No. De Iterações	Precisão de Treinamento	Precisão de Teste
28/29	51.6~3.3	39.6~28.7
$28/29.10^2$	75.6~8.8	27.4~29.8
$28/29.10^4$	89.3~6.1	22.3~20.8
$28/29.10^6$	90.4~5.7	15.7~21.6
UPSTART – SOMA		
28/29	51.7~5.6	30.6~23.9
$28/29.10^2$	69.1~10.6	9.9~21.1
$28/29.10^4$	82.9~10.1	18.2~21.1
$28/29.10^6$	85.8~4.6	21.5~14.3

Tabela 42. DistAl / Paridade-5

DISTAL					
Métrica de Distância	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
Euclideana	**100.0~0.0**	**83.3~26.8**	3.8~0.4	2	4
Manhattan	**100.0~0.0**	**83.3~26.8**	3.8~0.4	2	4
Valor Máximo	72.0~4.2	42.1~30.0	10.0~0.0	10	10
Chi square	63.0~5.4	46.6~20.3	10.0~0.0	10	10
Correlation	90.0~30.0	40.0~35.9	9.2~1.1	8	10
N-Chi square	63.0~5.4	46.6~20.3	10.0~0.0	10	10
Jaccard	100.0~0.0	80.0~22.1	4.0~0.0	4	4
Dice	100.0~0.0	80.0~22.1	4.0~0.0	4	4
Cosine	67.2~14.7	44.3~10.3	10.0~0.0	10	10
E.NL.	**100.0~0.0**	**83.3~26.8**	3.8~0.4	2	4
M.NL.	**100.0~0.0**	**83.3~26.8**	3.8~0.4	2	4
VM.NL.	72.0~4.2	42.1~30.0	10.0~0.0	10	10
E.ND.	**100.0~0.0**	76.6~30.0	3.8~0.4	2	4
M.ND.	**100.0~0.0**	80.0~30.5	3.8~0.4	2	4
VM.ND.	70.3~4.9	36.6~27.6	10.0~0.0	10	10

Tabela 43. Cascade-Correlation / Paridade-5

CASCADE-CORRELATION					
Nível de Paciência	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
1	**100.0~0.0**	66.4~29.4	5.4~1.1	4	7
5	**100.0~0.0**	66.5~36.6	5.2~1.6	3	7
10	98.6~2.8	74.9~31.0	5.7~2.5	2	10

Observando as Tabelas 38 a 43, podemos constatar que os algoritmos tiling, pyramid, DistAl e cascade-correlation conseguiram 100% de precisão no conjunto de treinamento, mas isso não ocorreu com os demais algoritmos. No conjunto de teste, o desempenho do algoritmo DistAl usando métricas euclideana ou de Manhattan com normalização baseada em limites ou sem normalização foi bastante superior aos demais algoritmos.

O domínio Paridade-5 par contém apenas 32 instâncias. Assim sendo, usando a técnica *10-fold cross-validation* descrita anteriormente, o conjunto de teste tem apenas 3 ou 4 instâncias. Esse baixo número de instâncias de teste explica, em termos, a inconstância dos valores encontrados para as precisões de teste.

Entre os algoritmos com neurônios intermediários, o número de neurônios variou bastante e, portanto, não se pode concluir quais são os algoritmos que necessitam de mais neurônios para construir a sua rede neural ou se o aumento do número de iterações aumenta o número de neurônios gerados nesse domínio de conhecimento.

O algoritmo DistAl conseguiu bons resultados no conjunto de teste usando as métricas euclideana e de Manhattan (83.3%), com normalização baseada em limites ou sem normalização. Nenhum algoritmo baseado em perceptron conseguiu precisão de teste acima de 61.5%, sendo que a maioria dos testes registrou precisão inferior a 30% nesses algoritmos. O algoritmo cascade-correlation teve todas as suas precisões de teste superiores a todas as precisões de testes dos algoritmos baseados em perceptron, mas teve precisão de teste inferior às melhores precisões de teste do algoritmo Distal.

Focalizando apenas os algoritmos baseados em perceptron, o aumento do número de iterações sempre aumentou a precisão de treinamento. Essa tendência, entretanto, não se verificou com relação a precisão no conjunto de teste. O algoritmo tiling conseguiu 100%

de precisão duas vezes, ie, com $28/29.10^4$ e $28/29.10^6$ iterações e o algoritmo pyramid conseguiu uma vez com $28/29.10^6$. Usando o maior número de iterações ($28/29.10^6$) as precisões de treinamento do algoritmo upstart em suas duas abordagens foram inferiores a todos os algoritmos, exceto o pocket com modificação ratchet.

Com o uso de níveis de paciência 1 e 5, o algoritmo cascade-correlation atingiu 100% de precisão de treinamento. Usando nível de paciência 10, o algoritmo atingiu 98.6% de precisão de treinamento. A melhor precisão de teste, curiosamente, foi atingida quando se obteve a menor precisão de treinamento, ie, com o nível de paciência maior (10). Assim sendo, conclui-se que o aumento do nível de paciência não necessariamente melhora a precisão de treinamento e precisões de treinamento maiores não necessariamente implicam em precisões de teste maiores, nesse domínio de conhecimento usando o algoritmo cascade-correlation.

O melhor desempenho geral nesse domínio de conhecimento foi obtido pelo algoritmo DistAl. Ele conseguiu 100% de precisão de treinamento usando diversas métricas e também conseguiu a melhor precisão de teste entre todos os algoritmos em várias métricas de distância. O algoritmo cascade-correlation com 100% de precisão de treinamento usando dois diferentes níveis de paciência e 74.9% de precisão de teste com nível de paciência 10 pode ser considerado o segundo melhor. Os algoritmos baseados em perceptron obtiveram um desempenho ruim no conjunto de teste.

8.3 ÍRIS

Íris é um domínio de conhecimento real que possui 150 instâncias distribuídas em três classes diferentes e descritas por 4 atributos. Cada classe faz referência a um tipo de planta Íris. Como as instâncias estão distribuídas em 3 classes (Íris setosa, Íris versicolour e Íris virginica) foi necessário adequá-lo para os experimentos. Assim, partindo dos dados originais, foram criados 3 novos domínios, cada um deles com instâncias apenas relativas a duas das três classes. Íris1 possui as instâncias pertencentes às classes versicolour e virginica, Íris2 possui as instâncias das classes setosa e virginica e Íris3 possui as instâncias das classes setosa e versicolour. Cada novo domínio possui 4 atributos e 100 instâncias. É importante ressaltar que os domínios Íris2 e Íris3 são linearmente separáveis e o domínio Íris1 tem alto grau de "linearidade", ie, apenas uma pequena parte das instâncias faz com

que o domínio seja não linearmente separável. As próximas três subseções mostram os resultados dos experimentos nos domínios Íris1, Íris2 e Íris3, respectivamente.

8.3.1 Íris1 (classes versicolour e virginica)

As Tabelas 44 a 49 mostram os resultados dos algoritmos no domínio de conhecimento Íris1.

Tabela 44. PMR / Íris1

PMR		
No. de Iterações	Precisão de Treinamento	Precisão de Teste
90	80.2~18.6	71.0~28.4
90.10^2	97.2~1.1	96.0~4.9
90.10^4	97.6~0.9	96.0~4.9

Tabela 45. Tower e Pyramid / Íris1

TOWER					
No. de Iterações	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
90	91.5~4.1	91.0~14.4	4.5~2.9	1	10
90.10^2	97.4~1.0	95.0~6.7	2.6~3.2	1	9
90.10^4	98.2~0.6	95.0~6.7	2.0~0.7	1	3
PYRAMID					
90	92.4~5.7	94.0~8.0	2.8~2.3	1	9
90.10^2	97.3~1.1	95.0~6.7	1.0~0.0	1	1
90.10^4	98.4~0.8	94.0~8.0	1.8~0.7	1	3

Tabela 46. Tiling / Íris1

TILING					
No. de Iterações	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários Mestres	Menor no. de Neurônios Intermediários Mestres	Maior no. de Neurônios Intermediários Mestres
90	95.8~0.9	96.0~6.6	3.6~1.6	1	6
90.10^2	98.4~0.8	95.0~6.7	2.0~0.6	1	3
90.10^4	98.4~0.8	94.0~10.1	1.8~0.6	1	3

Tabela 47. Upstart / Íris1

UPSTART – TREINAMENTO		
No. de Iterações	Precisão de Treinamento	Precisão de Teste
90	90.3~7.8	94.0~9.1
90.10^2	97.8~1.2	96.0~6.6
90.10^4	98.8~0.9	96.0~6.6

UPSTART – SOMA		
90	95.1~1.3	94.0~6.6
90.10^2	98.1~1.0	95.0~6.7
90.10^4	98.2~0.6	**97.0~4.5**

Tabela 48. DistAl / Íris1

DISTAL					
Métrica de Distância	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
Euclideana	95.0~15.0	89.0~16.4	4.4~4.2	2	10
Manhattan	90.0~30.0	82.0~28.9	4.1~4.4	2	10
Valor Máximo	90.0~30.0	82.0~29.2	5.4~2.0	3	8
Chi square	**100.0~0.0**	90.0~8.9	3.8~0.4	2	4
Correlation	**100.0~0.0**	94.0~8.0	3.6~0.4	2	4
N-Chi square	**100.0~0.0**	92.0~6.0	4.6~0.4	3	5
Jaccard	**100.0~0.0**	86.0~14.2	4.1~0.3	4	5
Dice	**100.0~0.0**	86.0~14.2	4.1~0.3	4	5
Cosine	**100.0~0.0**	94.0~6.6	3.8~0.6	3	5
E.NL.	**100.0~0.0**	87.0~10.0	4.5~0.5	4	5
M.NL.	**100.0~0.0**	89.0~9.4	4.4~0.4	4	5
VM.NL.	**100.0~0.0**	89.0~14.4	4.1~0.5	3	5
E.ND.	**100.0~0.0**	92.0~9.8	3.7~0.4	3	4
M.ND.	**100.0~0.0**	87.0~10.0	4.6~0.4	4	5
VM.ND.	**100.0~0.0**	88.0~13.2	5.2~0.7	4	6

Tabela 49. Cascade-Correlation / Íris1

CASCADE-CORRELATION					
Nível de Paciência	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
1	99.1~0.5	95.0~6.7	6.8~1.6	4	10
5	99.1~0.5	94.0~10.1	3.6~0.8	3	5
10	99.1~0.5	94.0~8.0	3.1~0.9	3	6

Observando os dados das Tabelas 44 a 49, podemos constatar que o algoritmo DistAl foi o único a conseguir 100% de precisão no conjunto de treinamento, usando diversas métricas. Essa superioridade, no entanto, não se verificou no conjunto de teste, pois a melhor precisão de teste do algoritmo DistAl, ie, 94% usando métricas cosine e correlation, foi inferior ou igual às precisões de teste de todos os outros algoritmos usando qualquer número de iterações ou níveis de paciência, exceto pelo pocket com modificação ratchet e tower usando 90 iterações.

Entre os algoritmos iterativos (todos exceto Distal), usando 90.10^2 ou 90.10^4 iterações ou qualquer nível de paciência, os desempenhos foram bastante parecidos. A precisão de treinamento manteve-se entre 97.2% (pocket com modificação ratchet) e 99.1% (cascade-correlation) e a precisão de teste manteve-se entre 94.0% (tiling, pyramid e cascade-correlation) e 97.0% (upstart-soma). Assim sendo, não se pode concluir qual dos algoritmos foi o mais eficiente. Curiosamente, as precisões de treinamento do algoritmo cascade-correlation foram iguais, independente do nível de paciência.

Entre os algoritmos baseados em perceptron, usando 90 iterações, o desempenho do algoritmo pocket com modificação ratchet foi bem inferior aos demais, pois a sua precisão de treinamento e teste foram inferiores a todos os outros algoritmos. Os outros algoritmos baseados em perceptron tiveram desempenhos parecidos.

O algoritmo DistAl teve os seus dois melhores desempenhos usando as métricas correlation e cosine que não utilizam nenhum tipo de normalização. O algoritmo teve 100% de precisão de treinamento e 94% de precisão de teste usando uma dessas métricas. O seu pior desempenho no conjunto de treinamento e também no conjunto de teste aconteceu usando as métricas valor máximo e de Manhattan.

O número de neurônios intermediários foi diminuindo conforme se aumentou o número de iterações ou o nível de paciência. Assim sendo, nesse domínio de conhecimento, conclui-se que com o aumento do número de iterações ou o nível de paciência, são necessários menos neurônios para a construção da rede dos algoritmos sem que se prejudique o desempenho dos mesmos.

Os algoritmos iterativos usando 90.10^2 ou 90.10^4 iterações ou qualquer valor para o nível de paciência sempre obtiveram precisão de teste superior ou igual ao algoritmo DistAl. Assim sendo, conclui-se que os algoritmos iterativos foram mais eficientes do que o algoritmo DistAl nesse domínio de conhecimento. Não se pode concluir, no entanto, qual entre os algoritmos iterativos foi o melhor. O algoritmo upstart com abordagem soma obteve a melhor precisão no conjunto de teste entre todos os algoritmos e bom desempenho de treinamento e teste em qualquer uma das execuções e, portanto, pode ser considerado uma boa escolha.

8.3.2 Íris2 (classes setosa e virginica)

As Tabelas 50 a 52 mostram os resultados dos algoritmos no domínio de conhecimento Íris2.

Tabela 50. PMR / Íris2

PMR		
No. de Iterações	Precisão de Treinamento	Precisão de Teste
90	**100.0~0.0**	**100.0~0.0**
90.10^2	**100.0~0.0**	**100.0~0.0**
90.10^4	**100.0~0.0**	**100.0~0.0**

Tabela 51. DistAl / Íris2

DISTAL					
Métrica de Distância	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
Euclideana	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
Manhattan	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
Valor Máximo	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
Chi square	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
Correlation	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
N-Chi square	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
Jaccard	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
Dice	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
Cosine	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
E.NL.	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
M.NL.	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
VM.NL.	**100.0~0.0**	99.0~3.0	2.0~0.0	2	2
E.ND.	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
M.ND.	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
VM.ND.	**100.0~0.0**	96.0~4.9	2.1~0.3	2	3

Tabela 52. Cascade-Correlation / Íris2

CASCADE-CORRELATION					
Nível de Paciência	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
1	**100.0~0.0**	**100.0~0.0**	1.8~0.8	1	3
5	**100.0~0.0**	**100.0~0.0**	1.7~0.7	1	3
10	**100.0~0.0**	**100.0~0.0**	1.9~0.9	1	3

Observando os dados das Tabelas 50 a 52, referentes a Íris2, podemos constatar que os algoritmos pocket com modificação ratchet e cascade-correlation conseguiram 100% de precisão de treinamento e também de teste em todas as suas execuções. O algoritmo DistAl também obteve 100% de precisão de treinamento e teste, exceto na métrica valor máximo com algum tipo de normalização. Na maioria das execuções o algoritmo DistAl necessitou de apenas 2 neurônios intermediários para a sua classificação.

Os algoritmos upstart, pyramid, tower e tiling começam a construção de suas redes neurais usando o algoritmo pocket com modificação ratchet. Se esse algoritmo classificar erroneamente alguma instância de treinamento então são adicionadas camadas (tower, pyramid e tiling) ou neurônios filhos/auxiliares (upstart). Como não houve nenhum erro nos testes, então todos os algoritmos baseados em perceptron funcionam exatamente como o pocket com modificação ratchet.

É importante lembrar que Íris2 é um domínio linearmente separável e, portanto, o algoritmo pocket com modificação ratchet consegue facilmente 100% de precisão de treinamento. Os algoritmos DistAl e cascade-correlation também obtiveram resultados excelentes nesse domínio linearmente separável.

O melhor algoritmo para esse domínio é certamente o pocket com modificação ratchet uma vez que ele conseguiu 100% de precisão de treinamento e também de teste em todas as suas execuções e é mais simples que os demais algoritmos (não adiciona camadas, neurônios auxiliares, intermediários, etc). Os piores são o Distal e cascade-correlation, uma vez que eles necessitaram de mais de um neurônio para construir as suas respectivas redes e o cascade-correlation, particularmente, leva um tempo maior de treinamento do que os demais algoritmos.

8.3.3 Íris3 (classes setosa e versicolour)

As Tabelas 53 a 55 mostram os resultados dos algoritmos no domínio de conhecimento Íris3.

Tabela 53. PMR / Íris3

PMR		
No. de Iterações	Precisão de Treinamento	Precisão de Teste
90	**100.0~0.0**	**100.0~0.0**

90.10^2	**100.0~0.0**	**100.0~0.0**
90.10^4	**100.0~0.0**	**100.0~0.0**

Tabela 54. DistAl / Íris3

DISTAL					
Métrica de Distância	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
Euclideana	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
Manhattan	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
Valor Máximo	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
Chi square	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
Correlation	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
N-Chi square	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
Jaccard	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
Dice	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
Cosine	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
E.NL.	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
M.NL.	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
VM.NL.	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
E.ND.	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
M.ND.	**100.0~0.0**	**100.0~0.0**	2.0~0.0	2	2
VM.ND.	**100.0~0.0**	95.0~6.7	2.0~0.0	2	2

Tabela 55. Cascade-Correlation / Íris3

CASCADE-CORRELATION					
Nível de Paciência	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
1	**100.0~0.0**	**100.0~0.0**	3.6~1.2	2	5
5	**100.0~0.0**	**100.0~0.0**	3.6~0.8	2	4
10	**100.0~0.0**	**100.0~0.0**	3.4~0.9	2	5

Observando os dados das Tabelas 53 a 55, referentes a Íris3, podemos constatar que as considerações para Íris3 são praticamente as mesmas que as para Íris2. O algoritmo DistAl não conseguiu 100% de precisão de teste em apenas uma das suas métricas (VM. ND.) e não em duas como acontecia em Íris2. O número de neurônios do cascade-correlation aumentou. O algoritmo precisou de 3.4 a 3.6 neurônios intermediários médios nesse domínio de conhecimento contra 1.7 a 1.9 no domínio Íris2. Novamente, o melhor algoritmo para esse domínio é o pocket com modificação ratchet. Podemos considerar que o cascade-correlation foi o pior, pois precisou de mais neurônios intermediários do que qualquer outro algoritmo em qualquer execução, mas ainda assim, esse número é pequeno.

8.4 MONKS

Monks1, Monks2 e Monks3 são domínios de conhecimento artificiais retirados do UCI Machine Learning Repository. Cada domínio possui 432 instâncias distribuídas em duas classes diferentes e descritas por 6 atributos. De acordo com [Blake & Merz 1998], "esse domínio de conhecimento foi a base para a primeira comparação a nível internacional de algoritmos de aprendizado de máquina. Os resultados dessa comparação podem ser encontrados em [Thrun et al 1991]". As instâncias que compõem os domínios Monks1, Monks2 e Monks3 são exatamente iguais, salvo pela classe associada a cada instância. Os domínios foram construídos da seguinte maneira:

- O primeiro (A1), segundo (A2) e quarto (A4) atributo dos domínios podem assumir os valores 1, 2 ou 3.
- O terceiro (A3) e o sexto (A6) atributo dos domínios podem assumir os valores 1 ou 2.
- O quinto (A5) atributo pode assumir os valores 1, 2, 3 ou 4.
- As 432 instâncias dos três domínios foram construídas a partir de todas as combinações dos valores que cada atributo pode assumir.
- O domínio Monks1 possui classe positiva quando A1 = A2 ou A5 = 1. Classe negativa, caso contrário.
- O domínio Monks2 possui classe positiva quando exatamente dois atributos possuem o valor 1. Classe negativa caso contrário.
- O domínio Monks3 possui classe positiva quando (A5 = 3 e A4 = 1) ou (A5 ≠ 4 e A2 ≠ 3). Classe negativa caso contrário.

As tabelas das próximas três subseções mostram os experimentos feitos com os algoritmos construtivos nos três domínios de conhecimento.

8.4.1 Monks1

As Tabelas 56 a 61 mostram os resultados dos algoritmos no domínio de conhecimento Monks1.

Tabela 56. PMR / Monks1

PMR		
No. de Iterações	Precisão de Treinamento	Precisão de Teste
388/389	65.8~6.6	66.2~6.4
$388/389.10^2$	72.8~1.5	71.8~5.1
$388/389.10^4$	73.3~2.1	73.5~3.2

Tabela 57. Tower e Pyramid / Monks1

TOWER					
No. de Iterações	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
388/389	70.9~2.4	70.9~4.1	3.7~1.7	1	6
$388/389.10^2$	76.4~1.3	75.1~2.3	6.2~2.4	1	10
$388/389.10^4$	79.4~2.6	76.9~4.2	7.7~1.6	5	10
PYRAMID					
388/389	70.4~4.0	71.0~4.7	2.5~1.9	1	7
$388/389.10^2$	76.1~0.8	74.0~4.2	8.1~2.6	3	10
$388/389.10^4$	78.4~5.0	77.4~7.7	9.5~0.8	8	10

Tabela 58. Tiling / Monks1

TILING					
No. de Iterações	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários Mestres	Menor no. de Neurônios Intermediários Mestres	Maior no. de Neurônios Intermediários Mestres
388/389	85.2~10.3	80.3~10.8	4.5~2.6	1	9
$388/389.10^2$	97.6~3.1	87.9~6.0	5.1~2.4	3	10
$388/389.10^4$	98.2~3.6	90.3~5.3	3.2~1.6	2	7

Tabela 59. Upstart / Monks1

UPSTART – TREINAMENTO		
No. de Iterações	Precisão de Treinamento	Precisão de Teste
388/389	71.9~2.5	68.8~6.1
$388/389.10^2$	76.1~3.3	72.5~7.0
$388/389.10^4$	81.7~2.7	78.9~6.0
UPSTART – SOMA		
388/389	72.9~3.3	71.6~3.6
$388/389.10^2$	78.8~2.9	73.3~6.2
$388/389.10^4$	81.6~4.4	77.3~4.1

Tabela 60. DistAl / Monks1

DISTAL					
Métrica de Distância	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
Euclideana	**100.0~0.0**	78.2~10.7	9.6~0.6	8	10
Manhattan	89.3~3.9	65.7~16.8	10.0~0.0	10	10
Valor Máximo	67.8~3.2	62.7~15.3	10.0~0.0	10	10
Chi square	**100.0~0.0**	75.5~16.8	7.1~0.5	6	8
Correlation	**100.0~0.0**	82.3~10.7	7.3~0.7	6	8
N-Chi square	**100.0~0.0**	62.9~16.0	9.0~0.6	8	10
Jaccard	**100.0~0.0**	64.9~18.0	8.4~0.4	8	9
Dice	**100.0~0.0**	64.9~18.0	8.4~0.4	8	9
Cosine	**100.0~0.0**	76.0~13.5	7.6~0.4	7	8
E.NL.	**100.0~0.0**	73.6~8.7	8.6~1.6	6	10
M.NL.	84.2~2.9	63.7~16.8	10.0~0.0	10	10
VM.NL.	67.1~2.2	61.7~12.9	10.0~0.0	10	10
E.ND.	**100.0~0.0**	65.2~12.3	8.6~1.2	7	10
M.ND.	82.3~7.9	65.3~17.8	10.0~0.0	10	10
VM.ND.	66.8~4.2	60.7~10.3	10.0~0.0	10	10

Tabela 61. Cascade-Correlation / Monks1

CASCADE-CORRELATION					
Nível de Paciência	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
1	82.0~3.7	78.4~11.0	6.1~2.5	2	10
5	97.5~4.2	97.3~4.4	9.1~0.9	8	10
10	**100.0~0.0**	**100.0~0.0**	8.5~1.0	7	10

Observando os dados das Tabelas 56 a 61, referentes a Monks1, podemos constatar que os algoritmos DistAl e cascade-correlation foram os únicos com 100% de precisão no conjunto de treinamento. O algoritmo tiling conseguiu 98.2% de precisão no conjunto de treinamento usando $388/389.10^4$ iterações. Os outros algoritmos tiveram desempenhos similares entre si sendo que nenhum deles conseguiu precisão maior do que 82% em alguma execução. O pior desempenho no conjunto de treinamento aconteceu quando se usou o algoritmo pocket com modificação ratchet com 388/389 iterações.

No conjunto de teste, o cascade-correlation foi o único a conseguir 100% de precisão, sendo assim melhor do que qualquer outro. O algoritmo tiling, usando $388/389.10^4$ e $388/389.10^2$, obteve 90.3% e 87.9% de precisão de teste, respectivamente, e foi o segundo melhor.

Entre os algoritmos iterativos, o aumento do número de iterações ou nível de paciência sempre aumentou a precisão de treinamento e teste. É importante notar que o algoritmo cascade-correlation usando nível de paciência 10 foi o único a conseguir 100% de precisão nos conjuntos de treinamento e também de teste, sendo assim absolutamente melhor do que os outros. O algoritmo pocket com modificação ratchet foi absolutamente pior do que os outros algoritmos iterativos, ie, ele sempre teve precisão de treinamento e teste inferior aos demais. As precisões de teste dos algoritmos tower, pyramid e upstart nas suas duas abordagens foram bastante similares em qualquer uma das execuções. As precisões de treinamento, no entanto, foram levemente melhores quando se usou o algoritmo upstart em qualquer uma das suas abordagens.

Embora o algoritmo DistAl, usando de suas algumas métricas, tenha conseguido precisão no conjunto de teste inferior apenas aos algoritmos cascade-correlation e tiling, usando outras de suas métricas o desempenho de teste foi pior do que o pior desempenho entre todos os outros algoritmos. Assim sendo, fica mostrado que, pelo menos nesse domínio de conhecimento, a escolha de uma métrica de distância adequada é essencial para um bom desempenho do algoritmo.

O algoritmo cascade-correlation obteve o melhor desempenho nesse domínio quando foi usado nível de paciência 10. É importante ressaltar, no entanto, que as precisões de treinamento e teste dos algoritmos baseados em perceptron sempre aumentaram com o aumento do número de iterações. Assim sendo, existe a probabilidade dos algoritmos tower, pyramid, upstart e tiling atingirem precisões de treinamento e teste tão boas quanto as do algoritmo cascade-correlation com um número mais alto de iterações. O algoritmo Distal, no entanto, não pode melhorar as suas precisões.

8.4.2 Monks2

As Tabelas 62 a 67 mostram os resultados dos algoritmos no domínio de conhecimento Monks2.

Tabela 62. PMR / Monks2

PMR		
No. de Iterações	Precisão de Treinamento	Precisão de Teste
388/389	66.7~0.6	66.6~6.3

388/389.10^2	66.7~0.6	66.1~5.9
388/389.10^4	66.6~0.6	67.8~6.1

Tabela 63. Tower e Pyramid / Monks2

TOWER					
No. de Iterações	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
388/389	66.7~0.6	66.6~6.3	1.0~0.0	1	1
388/389.10^2	66.8~0.7	66.4~6.6	1.7~2.1	1	8
388/389.10^4	67.0~0.6	66.4~6.4	3.7~3.5	1	9
PYRAMID					
388/389	66.7~0.6	66.6~6.3	1.0~0.0	1	1
388/389.10^2	66.8~0.7	66.4~6.6	2.6~2.5	1	8
388/389.10^4	66.8~0.6	67.8~6.1	3.6~3.3	1	10

Tabela 64. Tiling / Monks2

TILING					
No. de Iterações	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários Mestres	Menor no. de Neurônios Intermediários Mestres	Maior no. de Neurônios Intermediários Mestres
388/389	66.9~0.7	66.6~6.3	2.3~2.6	1	8
388/389.10^2	71.5~6.8	64.9~7.8	6.4~3.3	1	10
388/389.10^4	80.2~5.7	66.4~5.5	9.6~0.6	8	10

Tabela 65. Upstart / Monks2

UPSTART – TREINAMENTO		
No. de Iterações	Precisão de Treinamento	Precisão de Teste
388/389	66.7~0.6	66.6~6.3
388/389.10^2	67.0~1.0	66.8~6.3
388/389.10^4	67.8~1.8	66.2~6.5
UPSTART – SOMA		
388/389	66.7~0.6	66.6~6.3
388/389.10^2	67.4~1.0	67.0~5.9
388/389.10^4	68.5~1.1	65.8~6.2

Tabela 66. DistAl / Monks2

DISTAL					
Métrica de Distância	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
Euclideana	92.7~8.1	68.0~14.7	9.2~2.4	2	10
Manhattan	80.1~27.1	56.5~22.0	9.2~2.4	2	10
Valor Máximo	44.8~14.9	38.0~16.6	9.2~2.4	2	10
Chi square	92.0~24.0	57.6~26.1	7.6~2.6	2	10
Correlation	89.9~29.9	50.0~22.3	8.4~2.9	4	10

N-Chi square	**100.0~0.0**	63.5~14.1	8.8~0.6	6	10
Jaccard	**100.0~0.0**	**79.2~10.4**	7.7~0.7	7	8
Dice	**100.0~0.0**	78.4~9.8	7.7~0.7	7	8
Cosine	**100.0~0.0**	77.3~16.1	7.7~0.6	7	8
E.NL.	90.7~10.4	65.0~12.1	9.2~2.4	2	10
M.NL.	78.2~20.0	59.5~18.3	9.2~2.4	2	10
VM.NL.	34.3~24.9	35.2~20.4	9.2~2.4	2	10
E.ND.	88.4~9.1	68.0~14.7	9.2~2.4	2	10
M.ND.	80.0~25.7	56.7~20.0	9.2~2.4	2	10
VM.ND.	40.4~13.2	30.0~16.6	9.2~2.4	2	10

Tabela 67. Cascade-Correlation / Monks2

CASCADE-CORRELATION					
Nível de Paciência	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
1	70.2~2.7	64.9~6.1	6.9~2.7	2	10
5	75.8~4.0	66.5~5.2	8.0~3.5	1	10
10	80.9~2.7	66.2~5.1	9.5~0.5	9	10

Observando os dados das Tabelas 62 a 67, referentes a Monks2, podemos constatar que o algoritmo DistAl usando a métrica de Jaccard teve melhor precisão de treinamento e também melhor precisão de teste entre todos os algoritmos. Os algoritmos iterativos tiveram precisões de treinamento e teste bastante similares, salvo pelos algoritmos tiling e cascade-correlation que obtiveram precisões de treinamento maiores em algumas execuções.

Entre os algoritmos iterativos, o aumento do número de iterações ou nível de paciência aumentou levemente a precisão de treinamento dos algoritmos pocket com modificação ratchet, tower, pyramid e upstart e aumentou consideravelmente as precisões de treinamento dos algoritmos cascade-correlation e tiling. As precisões de teste, no entanto, não necessariamente melhoraram com o aumento do número de iterações. As diferenças no conjunto de treinamento e também de teste entre os algoritmos pocket com modificação ratchet, tower, pyramid e upstart foram insignificantes, sendo que as precisões de treinamento e teste desses algoritmos foram absolutamente iguais quando se usou 388/389 iterações.

As precisões de treinamento e teste do algoritmo DistAl variaram muito com a métrica de distância. A precisão de treinamento foi 34.3% usando VM.NL. e 100% em diversas outras métricas. A precisão de teste foi de 35.2% usando VM.NL. e 79.2% usando

métrica de Jaccard. Assim sendo, nesse domínio de conhecimento, o bom desempenho do algoritmo DistAl dependeu fundamentalmente de uma apropriada métrica de distância.

O número de neurônios intermediários aumentou levemente quando se aumentou o número de iterações nos algoritmos tower e pyramid e aumentou bruscamente com o aumento no algoritmo tiling. Com o aumento do nível de paciência no algoritmo cascade-correlation, o número de neurônios intermediários aumentou moderadamente. O aumento do número de neurônios intermediários nesses algoritmos, no entanto, não necessariamente melhorou os desempenhos nos conjuntos de treinamento e teste. No algoritmo Distal, o número de neurônios intermediários foi próximo de 10 (que é o limite utilizado), usando a maioria das métricas de distância. O melhor desempenho no conjunto de teste, no entanto, foi atingido quando o número de neurônios intermediários foi um dos menores entre todas as métricas utilizadas.

O melhor desempenho nesse domínio aconteceu quando foi usado o algoritmo DistAl com a métrica de Jaccard. Os melhores valores para treinamento e teste aconteceram usando esse algoritmo com essa métrica. Entre os algoritmos iterativos, não se pode concluir qual foi o melhor, pois as precisões de teste foram bastante similares.

8.4.3 Monks3

As Tabelas 68 a 73 mostram os resultados dos algoritmos no domínio de conhecimento Monks3.

Tabela 68. PMR / Monks3

PMR		
No. de Iterações	Precisão de Treinamento	Precisão de Teste
388/389	73.0~4.6	73.3~6.5
$388/389.10^2$	83.6~2.0	80.0~5.4
$388/389.10^4$	85.3~1.7	83.6~4.8

Tabela 69. Tower e Pyramid / Monks3

TOWER					
No. de Iterações	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
388/389	78.6~2.3	77.4~8.1	4.2~1.8	1	7
$388/389.10^2$	89.9~3.8	88.8~7.4	3.6~2.8	1	10
$388/389.10^4$	91.2~3.7	89.6~5.0	4.6~3.1	1	10

PYRAMID					
388/389	78.4~2.5	76.0~5.6	4.3~2.7	2	10
$388/389.10^2$	86.8~2.4	85.8~4.9	4.7~2.2	1	9
$388/389.10^4$	88.2~2.4	88.4~6.9	4.1~2.9	1	10

Tabela 70. Tiling / Monks3

TILING					
No. de Iterações	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários Mestres	Menor no. de Neurônios Intermediários Mestres	Maior no. de Neurônios Intermediários Mestres
388/389	93.8~3.1	93.7~3.1	4.4~1.5	2	7
$388/389.10^2$	97.9~3.6	**96.7~3.2**	4.2~1.8	2	8
$388/389.10^4$	98.0~3.5	95.3~4.9	3.3~2.0	2	9

Tabela 71. Upstart / Monks3

UPSTART – TREINAMENTO		
No. de Iterações	Precisão de Treinamento	Precisão de Teste
388/389	72.4~4.3	72.5~5.5
$388/389.10^2$	89.2~5.4	87.6~9.6
$388/389.10^4$	88.9~6.2	86.8~6.6
UPSTART – SOMA		
388/389	75.1~1.9	73.8~5.9
$388/389.10^2$	86.6~4.1	82.6~7.5
$388/389.10^4$	89.3~3.4	87.7~4.0

Tabela 72. DistAl / Monks3

DISTAL					
Métrica de Distância	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
Euclideana	95.1~14.5	82.5~13.9	4.9~4.2	2	10
Manhattan	89.8~29.9	66.2~24.4	8.6~3.8	4	10
Valor Máximo	92.0~24.0	86.2~29.2	4.0~4.0	2	10
Chi square	92.0~24.0	67.5~28.3	6.2~4.1	3	10
Correlation	92.0~24.0	71.2~26.2	5.7~4.9	2	10
N-Chi square	**100.0~0.0**	77.5~12.5	6.6~2.2	2	9
Jaccard	**100.0~0.0**	83.7~16.2	5.5~0.5	5	6
Dice	**100.0~0.0**	82.5~16.0	5.5~0.5	5	6
Cosine	**100.0~0.0**	78.7~14.8	7.4~2.5	3	10
E.NL.	95.1~14.5	82.5~13.9	5.3~3.8	2	10
M.NL.	87.5~24.2	60.0~23.2	8.6~3.8	4	10
VM.NL.	89.0~18.7	82.2~26.2	4.4~4.0	2	10
E.ND.	93.7~8.8	80.5~7.9	4.8~4.5	2	10
M.ND.	82.6~24.3	65.2~22.4	8.6~3.8	4	10
VM.ND.	92.0~24.0	86.2~29.2	4.0~4.0	2	10

Tabela 73. Cascade-Correlation / Monks3

CASCADE-CORRELATION					
Nível de Paciência	Precisão de Treinamento	Precisão de Teste	No. de Neurônios Intermediários	Menor no. de Neurônios Intermediários	Maior no. de Neurônios Intermediários
1	93.7~6.8	91.5~7.3	8.9~0.9	8	10
5	97.1~2.8	96.2~3.5	9.0~0.8	8	10
10	96.5~3.4	95.0~4.2	8.7~1.0	7	10

Observando os dados das Tabelas 68 a 73, referentes a Monks3, podemos constatar que novamente o algoritmo DistAl foi o único a conseguir 100% de precisão no conjunto de treinamento usando diversas métricas de distância. Os algoritmos tiling e cascade-correlation conseguiram bons resultados no conjunto de treinamento também, atingindo 98% e 97.1% de precisão respectivamente. Os desempenhos dos demais algoritmos no conjunto de treinamento foram similares, exceto pelo algoritmo pocket com modificação ratchet que obteve precisão de treinamento menor.

Apesar do algoritmo DistAl ter sido o único a conseguir 100% de precisão no conjunto de treinamento, a sua melhor precisão de teste (86.2%) foi pior do que as precisões de teste de todos os outros algoritmos baseados em perceptron quando foi usado $388/389.10^4$ iterações, exceto pelo pocket com modificação ratchet. A melhor precisão de teste do Distal também foi pior do que todas as precisões de teste obtidas pelo cascade-correlation. Assim sendo, podemos considerar que o algoritmo DistAl foi o segundo pior, pois todos os outros, exceto o pocket com modificação ratchet, conseguiram precisão de teste melhores quando foi usado um número apropriado de iterações ou nível de paciência.

Os algoritmos tiling e cascade-correlation foram muito melhores do que os outros algoritmos no conjunto de teste. A pior precisão de teste desses algoritmos em qualquer uma das execuções foi melhor do que qualquer precisão de teste dos outros algoritmos. Não se pode concluir qual destes dois algoritmos foi melhor, embora a melhor precisão de teste absoluta tenha sido adquirida pelo algoritmo tiling.

Entre os algoritmos iterativos, o aumento do número de iterações ou nível de paciência quase sempre aumentou as precisões de treinamento e teste. A melhor precisão de teste, no entanto, foi atingida usando $388/389.10^2$ iterações (pelo algoritmo tiling) e não $388/389.10^4$.

Os melhores algoritmos nesse domínio de conhecimento foram o tiling e o cascade-correlation. Eles obtiveram as melhores precisões de teste e boas precisões de treinamento em qualquer uma das suas execuções. No geral, o tiling obteve desempenho levemente superior ao cascade-correlation, mas não podemos concluir qual dos dois é melhor para esse domínio. Como comentado anteriormente, é importante ressaltar que existe a probabilidade dos algoritmos baseados em perceptron obterem precisões de treinamento e teste tão boas quanto as do tiling ou cascade-correlation, se o número de iterações for aumentada. O algoritmo DistAl, em contrapartida, não tem como melhorar o seu desempenho nesse domínio.

8.5 CONCLUSÕES E TRABALHOS FUTUROS

Aprendizado neural construtivo pode ser bastante eficiente na classificação de instâncias e reconhecimento de padrões. Em especial, não existe a necessidade de se definir inicialmente a topologia da rede, pois estas são geradas dinamicamente, durante o treinamento. Sendo assim, não existe a possibilidade (bastante comum) de se estimar uma topologia de rede erroneamente. Alguns algoritmos construtivos, como o DistAl, criam uma única camada de neurônios intermediários e adicionam neurônios a esta camada enquanto for necessário. Há ainda alguns algoritmos, como o pyramid, tower e cascade-correlation, que criam várias camadas com apenas um neurônio em cada camada e existem aqueles que criam várias camadas com vários neurônios, como tiling. De qualquer forma, devido à própria natureza de algoritmos construtivos, a adição de neurônios intermediários durante o treinamento só é feita se necessário e, conseqüentemente, o tamanho da rede será diretamente proporcional à complexidade das instâncias de treinamento, ie, as redes geradas por estes algoritmos são bastante pequenas, se comparadas às geradas por algoritmos neurais não construtivos, e podem ser altamente precisas em alguns domínios. Por estas duas razões, aprendizado neural construtivo pode ser especialmente atrativo para tarefas de grande porte, onde a velocidade do aprendizado e o tamanho da rede gerada são fatores de suma importância.

As propostas originais dos algoritmos construtivos apresentados nessa dissertação são, exceto o Distal e o cascade-correlation, limitadas ao aprendizado de duas classes apenas. Existem, no entanto, extensões desses algoritmos para lidar com várias classes. A

referência [Parekh et al 1997] descreve um algoritmo chamado Mupstart, que é uma adaptação do algoritmo upstart para várias classes. A referência [Parekh et al 2000] descreve o Mtiling e o Mpyramid, extensões para aprendizado de várias classes dos algoritmos tiling e pyramid, respectivamente. A referência [Parekh 1998] descreve e analisa o algoritmo Mtower, extensão do algoritmo tower para várias classes, além de analisar os algoritmos Mpyramid, Mtiling e Mupstart. A versão do algoritmo cascade-correlation apresentada nessa dissertação é limitada ao aprendizado em duas classes. A versão original, no entanto, não limita o número de classes. As referências [Squires & Shavlik 1991] [Fahlman & Lebiere 1990] mostram versões do algoritmo cascade-correlation para aprendizado em domínios com diversas classes.

Esse trabalho mostrou as características dos algoritmos neurais construtivos tower, pyramid, tiling, upstart, Distal e cascade-correlation, descrevendo-os usando uma mesma notação e padronização, apresentou um exemplo do uso de cada algoritmo, um sistema que implementa os algoritmos e os comparou, através de testes feitos em diferentes domínios de conhecimento. Na literatura existem várias referências que descrevem algoritmos construtivos; não existem, entretanto, muitas comparações experimentais entre os diversos algoritmos, principalmente envolvendo o Distal. Os autores do algoritmo Distal, por exemplo, apenas comparam o Distal com os resultados de uma implementação NN (Nearest-Neighbor) em [Yang et al 1997], não o comparando com qualquer neural construtivo. Com base nos resultados dos experimentos descritos nesse trabalho, pode-se dizer que:

- O algoritmo Distal realmente é muito dependente da métrica de distância utilizada. Na maioria dos domínios de conhecimento, há uma diferença significativa entre as precisões de treinamento e teste quando o algoritmo usa métricas distintas.
- O algoritmo Distal conseguiu 100% de precisão no conjunto de treinamento em todos os domínios, utilizando a maioria de suas métricas. Essa superioridade no conjunto de treinamento não reflete, no entanto, no conjunto de teste. Esse fato sugere que Distal faz um "overfitting" dos exemplos durante o treinamento.
- Os algoritmos tower e pyramid tiveram praticamente o mesmo desempenho. As diferenças entre as precisões de treinamento e teste entre esses dois algoritmos nos domínios de conhecimento utilizados foram sempre estatisticamente irrelevantes.

- As versões upstart-soma e upstart-treinamento do algoritmo upstart tiveram desempenhos bastante similares. As diferenças entre as precisões de treinamento e teste foram estatisticamente irrelevantes.
- O algoritmo tiling pode ser considerado o melhor entre os baseados em perceptron (pocket com modificação ratchet, tower, pyramid, upstart e tiling), tendo como base os experimentos realizados nesse trabalho. As precisões de treinamento e teste desse algoritmo em alguns domínios foram insignificantemente piores do que as precisões de outros algoritmos baseados em perceptron, mas foram absolutamente melhores nos domínios Monks1 e Monks3.
- Como dito anteriormente, existem várias versões para o algoritmo cascade-correlation. Além disso, cada versão do algoritmo pode usar diferentes funções na construção da rede neural. Assim sendo, apenas podemos concluir que a versão do cascade-correlation (ver Capítulo 7 e Anexo D) experimentada nessa dissertação é altamente eficiente em alguns domínios (Monks1, Monks3) e altamente ineficiente em outros (Vestibular).
- Em alguns domínios, os algoritmos construtivos não têm utilidade nenhuma. Nos domínios linearmente separáveis, como Íris2 e Íris3, é muito mais interessante usar o algoritmo pocket com modificação ratchet, pois é de fácil implementação, mais simples, gasta menos espaço em memória e é mais rápido do que os algoritmos construtivos. No domínio Monks2, levando em conta a precisão de teste, todos os algoritmos tiveram desempenhos similares ao algoritmo pocket com modificação ratchet, exceto o Distal.
- Em alguns domínios, os algoritmos construtivos tiveram desempenhos muito satisfatórios. O resultados do algoritmo Distal no domínio Paridade-5, do algoritmo cascade-correlation no domínio Monks1 e do algoritmo tiling no domínio Monks3 foram bastante interessantes e demonstram que os algoritmos construtivos podem ser uma boa opção em alguns domínios.

Esse trabalho não almeja, no entanto, apresentar uma conclusão definitiva sobre eficiência e aplicabilidade dos algoritmos neurais construtivos. Pesquisas futuras são

necessárias para avaliar melhor essa família de algoritmos. Possíveis continuidades para esse trabalho são:

- Investigar outros algoritmos neurais construtivos como o shift e pti [Amaldi & Guenin 1997], sequential [Marchand et al 1990] ou perceptron-cascade [Burgess 1994].
- Avaliar empiricamente todos os algoritmos neurais construtivos citados nesse documento em diversos outros domínios, buscando, desse modo, determinar quais são os melhores algoritmos para determinados domínios.
- Avaliar empiricamente outras versões do algoritmo cascade-correlation, usando diferentes funções em cada versão.
- Comparar algoritmos neurais construtivos com redes neurais convencionais backpropagation.
- Fazer um estudo quanto à complexidade dos algoritmos neurais construtivos, buscando determinar, desse modo, a velocidade dos mesmos nos diferentes domínios de dados.
- Avaliar as propostas para situação de aprendizado multi-classe dos algoritmos construtivos analisados nessa dissertação.

ANEXO A. FUNDAMENTOS MATEMÁTICOS E NOTAÇÃO

O conteúdo dessa subseção foi adaptado de [Nicoletti & Rocha 2002]. De uma maneira simplista vetores podem ser considerados como representações de quantidades que têm magnitude e direção. A magnitude de um vetor v geralmente é indicada como ||v||. Assim sendo, um vetor pode ser definido como um par (||v||, θ), onde θ é o ângulo que o vetor faz com algum sistema direcional, conforme mostra a Figura 29.

Visando tanto a sua generalização para espaços n dimensionais quanto o estabelecimento da relação entre vetores e a representação do espaço de instâncias, é mais conveniente descrever vetores tendo como referência o sistema cartesiano. Num sistema cartesiano bidimensional um vetor é representado como o seu comprimento projetado sobre os dois eixos coordenados, como mostra a Figura 30. Ele é descrito como o par ordenado dos valores de suas componentes $v = (v_1,v_2)$ i.e., uma lista ordenada de valores numéricos. Num espaço n-dimensional um vetor é definido como uma lista de n valores numéricos $v = (v_1,v_2,.....,v_n)$.

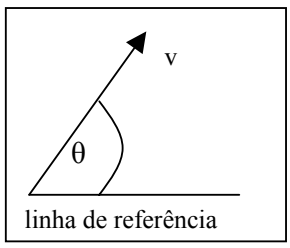


Figura 29. Vetores como quantidade com magnitude e direção

Definição 1. Num espaço n-dimensional o comprimento de um vetor $v = (v_1,v_2,.....,v_n)$ é definido como:

$$\| v \| = \left[\sum_{i=1}^{n} v_i^{\,2} \right]^{1/2}$$

A Figura 30 mostra a representação cartesiana de um vetor num espaço bidimensional, onde pode ser verificado que $\| v \| = \sqrt{v_1^2 + v_2^2}$.

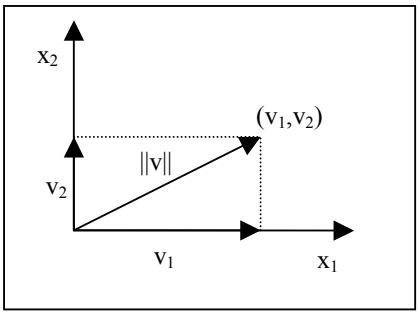


Figura 30. Vetor descrito como par ordenado (v_1,v_2)

Definição 2. Num espaço n-dimensional, a multiplicação de um vetor $u = (u_1,u_2,....,u_n)$ por um escalar t resulta no vetor $tu = (tu_1,tu_2,....,tu_n)$.

Definição 3. Dado um espaço n-dimensional, a soma dos vetores $u = (u_1,u_2,....,u_n)$ e $v= (v_1,v_2,....,v_n)$ é o vetor $w = (w_1,w_2,....,w_n)$ onde $w_i = u_i + v_i$, para i = 1,...,n.

Definição 4. Dado um espaço n-dimensional, a subtração do vetor $u = (u_1,u_2,....,u_n)$ por $v = (v_1,v_2,....,v_n)$ é o vetor $w = (w_1,w_2,....,w_n)$ onde $w_i = u_i – v_i$, para i = 1,...,n. A subtração de v é precisamente a adição de –v.

Definição 5. Num espaço n-dimensional, o produto escalar (produto interno) dos vetores $v = (v_1,v_2,....,v_n)$ e $w = (w_1,w_2,....,w_n)$ é definido como:

$$v.w = \sum_{i=1}^{n} v_i w_i$$

e satisfaz as seguintes condições, onde α é um escalar:

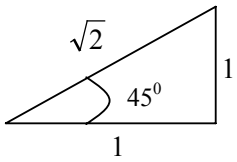
1. $u.u \geq 0$

   $u.u = 0$ se e só se $u = 0$
2. $\alpha u.v = \alpha(u.v)$

   $u.\alpha v = \alpha(u.v)$
3. $u.(v + w) = u.v + u.w$

   $(u + v).w = u.w + v.w$
4. $u.v = v.u$

Considere os vetores v = (1,1) e w = (2,0), como mostra a Figura 31. O produto interno dos vetores da Figura 31 é v.w = 2. Note que para esse exemplo, $\| v \| = \sqrt{2}$ e $\| w \|$ = 2. Além disso, observe que o cosseno do ângulo entre os vetores é $1/\sqrt{2}$, pois:



cos(45°)=cateto adjacente/hipotenusa
$= 1/\sqrt{2}$

.

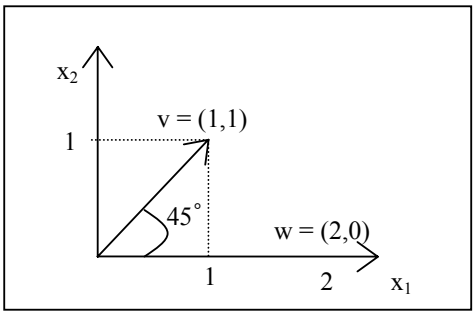


Figura 31. Representação dos vetores v = (1,1) e w = (2,0)

Portanto, $v.w = 2 = \sqrt{2}.2.1/\sqrt{2} = \| v \| \| w \| \cos 45$. Esse resultado é um resultado geral, definido a seguir.

Definição 6. Se o ângulo entre dois vetores v e w é θ, a definição do produto interno dada na Definição 5 é equivalente a

$$v.w = \| v \| \| w \| \cos\theta .$$

ANEXO B. PROVA DO TEOREMA DA CONVERGÊNCIA DO PERCEPTRON

O conteúdo desse anexo foi extraído de [Gallant 1994].

Teorema da Convergência do Perceptron. *Seja E um (possivelmente infinito) conjunto de exemplos de treinamento, com cada exemplo tendo tamanho máximo K. Se existir um vetor de pesos* W^* *e um número* $\delta > 0$ *tal que* $W^*.E^k \geq \delta$ *para todo* $E^k \in E$ *então o algoritmo perceptron irá executar o passo de mudança do vetor de pesos* ($W \leftarrow W + C^kE^k$) *no máximo*

$$\left(\frac{K\left\|W^*\right\|}{\delta} \right)^2 \textit{ vezes}$$

Prova do teorema. A idéia da prova é mostrar que o cosseno do ângulo θ entre W e W^*, $\cos\theta = \frac{W^*.W}{\left\|W^*\right\|\left\|W\right\|}$ será maior do que 1 (uma contradição) se muitos passos de mudança forem executados. Note que $\cos\theta\left\|W^*\right\|\left\|W\right\| = W.W^*$. (1)

O vetor de pesos na t-ésima execução do passo de mudança é notado por W^t. Inicialmente W^0 = <0 0 0....0> embora qualquer outro ponto inicial possa ser escolhido (com pequenas modificações). Depois de t + 1 modificações do vetor de pesos tem-se:

$W^*.W^{t+1} = W^*.\left(W^t + E^k\right)$ onde E^k causou a execução do passo de mudança.

$= W^*.W^t + W^*.E^k$

$\geq W^*.W^t + \delta$

Portanto,

$W^*.W^t \geq t\delta$ pois em cada passo mudança, há um aumento máximo de δ no valor de $W^*.W^t$. (2)

Considerando o quadrado do tamanho de W:

$$\left\|W^{t+1}\right\|^2 = W^{t+1}.W^{t+1}$$

$$= \left(W^t + E^k\right)\left(W^t + E^k\right)$$ onde E^k causou a execução do passo de mudança.

$$= W^t.W^t + 2\,W^t.E^k + E^k.E^k$$

$$\leq \left\|W^t\right\|^2 + K^2$$ porque $W^t.\ E^k \leq 0$ foi o que causou o passo de mudança.

Portanto

$\left\|W^t\right\|^2 \leq tK^2$ pois em cada passo mudança, há uma diminuição máxima de K^2 no valor de $\left\|W^t\right\|^2$. (3)

Combinando as equações acima se tem:

$t\delta \leq W^*.W^t$ pela equação (2)

$\leq \left\|W^*\right\| \left\|W^t\right\| \cos\theta$ pela equação (1)

$\leq \left\|W^*\right\| \left\|W^t\right\|$ Note que se o cos θ fosse maior do que 1, não seria possível essa passagem. Por isto que a idéia da prova é mostrar que ocorrerá uma contradição (cos θ > 1) quando muitos passos de mudança forem executados.

$\leq \left\|W^*\right\| K\sqrt{t}$ pela equação (3)

Assim

$$tt \leq \left(\frac{K\left\|W^*\right\|}{\delta}\right)^2 t$$ e portanto,

$$t \leq \left(\frac{K\left\|W^*\right\|}{\delta}\right)^2$$

como estabelece o teorema.

É importante notar que o teorema de convergência do perceptron depende da existência de algum W^* que classifica corretamente todos os exemplo de treinamento. Em outras palavras, o teorema é aplicável apenas se E for um conjunto de exemplos linearmente separáveis.

ANEXO C. IMPLEMENTAÇÃO DO SISTEMA DISTAL

Muito embora os autores do DistAl disponibilizaram uma implementação do algoritmo, optamos por reprogramá-la em muitas de suas partes pelas seguintes razões: a implementação fornecida pelos autores não era documentada, a saída era confusa e a interação com o sistema era difícil, dado a inexistência de:

- como interagir com o sistema,
- quais informações o sistema espera,
- quais, em qual formato e em qual seqüência os dados devem ser passados para o sistema.

A implementação desenvolvida é um refinamento da existente que buscou contemplar os aspectos mencionados acima, entre outros, que estão faltando na implementação original. As próximas duas subseções detalham o sistema com o refinamento já implementado.

C.1 USANDO O SISTEMA

Para se usar o sistema DistAl é necessário, inicialmente, um arquivo de entrada (arquivo de treinamento). Os atributos de uma mesma instância devem ser seqüenciais, na mesma linha e com um espaço simples os separando. A classe da instância deve ser o último valor. Outras instâncias devem vir seqüencialmente nas linhas abaixo.

O arquivo deve ser composto apenas por valores numéricos. Caso o arquivo de treinamento tenha instâncias com valores nominais ou valores de atributos ausentes, o

usuário deve realizar um pré-processamento que o transforma em um arquivo com valores numéricos. Por exemplo, o arquivo dado na Figura 32.

```
sunny, 85, 85, false, Don't Play
sunny, 80, 90, true, Don't Play
overcast, 83, 78, false, Play
rain, 70, 96, false, Play
rain, 68, 80, false, Play
rain, 65, 70, true, Don't Play
overcast, 64, 65, true, Play
sunny, 72, 95, false, Don't Play
sunny, 69, 70, false, Play
rain, 75, 80, false, Play
sunny, 75, 70, true, Play
---, 72, 90, true, Play
overcast, 81, 75, false, Play
rain, 71, 80, true, Don't Play
```

Figura 32. Arquivo de entrada original, sem o pré-processamento inicial

deve ser convertido em:

```
0 85 85 0 0
0 80 90 1 0
1 83 78 0 1
2 70 96 0 1
2 68 80 0 1
2 65 70 1 0
1 64 65 1 1
0 72 95 0 0
0 69 70 0 1
2 75 80 0 1
0 75 70 1 1
1.0e+10 72 90 1 1
1 81 75 0 1
2 71 80 1 0
```

Figura 33. Arquivo de entrada pré-processado

O pré-processamento transforma os valores *sunny*, *rain* e *overcast* do primeiro atributo em 0, 1 e 2 respectivamente. Os valores do segundo atributo, *false* e *true*, são transformados em 0 e 1 respectivamente. Os valores de classe: *Don't Play* e *Play* se tornam 0 e 1 respectivamente. O pré-processamento transforma valores ausentes de atributos no valor 1.0e+10.

Ao ser executado, o sistema interroga o usuário por: nome do arquivo de treinamento, o número de atributos que cada instância de treinamento possui, número total de classes existentes no conjunto de treinamento, número total de instâncias, número total de valores nominais para cada atributo e dois números que significam o intervalo de métricas de distâncias adotadas (com 0 e 23, cobre-se todas as métricas de distância

implementadas). Para exemplificar, usaremos o exemplo do conjunto de treinamento relativo à Figura 33, supondo que o nome do arquivo é golf.data. As perguntas e respectivas respostas para esse exemplo são mostradas na Figura 34.

```
Enter file name: golf.data
Enter the number of input atributes: 4
Enter the number of possible output classes: 2
Enter the total number of patterns: 14
How many possible nominal values the atribute 1 can assume? 3
How many possible nominal values the atribute 2 can assume? 0
How many possible nominal values the atribute 3 can assume? 0
How many possible nominal values the atribute 4 can assume? 2
What is the lower distance metric number ? 0
What is the upper distance metric number ? 23
```

Figura 34. Perguntas e respostas para o exemplo golf.data

C.2 SAÍDA DO SISTEMA

Como saída, o programa mostra inicialmente, para cada métrica de distância escolhida, os testes feitos no conjunto de treinamento e no conjunto de teste. A Figura 35 mostra a primeira parte da saída, para a métrica de distância euclidiana.

```
(Euclidean)     training accuracy after hidden neuron 1  is: 7.69231
testing accuracy after hidden neuron 1  is: 100
training accuracy after hidden neuron 2  is: 100
testing accuracy after hidden neuron 2  is: 0

Best Trainning accuracy - based on testing accuracy(bestTracc) : 7.69231
Best testing accuray(bestTsacc) : 100
Best Number of hidden neurons - based on testing accuray(bestHid) : 1
Trainning Accuracy(Tracc): 100
Testing Accuracy(Tsacc) : 0
Number of hidden neurons(Hid) : 2
```

Figura 35. Trecho da primeira parte da saída do sistema DistAl

O sistema implementa a técnica de *10-fold cross-validation*. Portanto, trechos como os da Figura 35 são mostrados 10 vezes (uma vez para cada validação) para cada métrica. Pode-se alterar esse valor modificando uma linha no código fonte do programa.

A segunda parte da saída mostra diversas médias calculadas após a execução do programa. Primeiramente, são mostradas as médias dos testes feitos no conjunto de teste. São mostradas a média das melhores precisões de teste (best testing accuracy) e a média das precisões de teste (Testing Accuracy) e os seus respectivos desvios padrões, para cada

métrica de distância. A seguir, são mostradas as médias dos testes feitos no conjunto de treinamento e as médias dos números de neurônios gerados. São mostradas a média das melhores precisões de treinamento (best trainning accuracy), a média dos menores números de neurônios intermediários (best number of hidden neurons), média das precisões de treinamento (trainning accuracy) e a média do número de neurônios gerados (number of hidden neurons) e os seus respectivos desvios, para cada métrica de distância. É importante ressaltar que os melhores números de neurônios e as melhores precisões de treinamento são classificados de acordo com a precisão no conjunto de teste. Ou seja, se existir 1 neurônio intermediário com 100% de precisão de treinamento e 30% de precisão de teste e existirem 2 neurônios com precisão de treinamento de 50% e precisão de teste de 100%, os valores para melhor precisão de treinamento e menor número de neurônios serão, respectivamente, 50% e 2. A Figura 36 mostra a segunda parte da saída, para cinco métricas e para a abordagem baseada em atributo (Attr-based).

```
final averaged results: bestTsacc,Tsacc; bestTracc,bestHid,Tracc,Hid <<
(Euclidean)     90.00~30.00      70.00~45.83
(Manhattan)     80.00~40.00      60.00~48.99
(Maximum value) 90.00~30.00      70.00~45.83
(Attr-based)    90.00~30.00      80.00~40.00
(NE:r,p)        100.00~0.00      80.00~40.00
(NX:s,p)        100.00~0.00      80.00~40.00
---
(Euclidean)     71.54~43.52      1.60~0.66      100.00~0.00  2.00~0.00
(Manhattan)     52.31~39.64      0.80~0.40      100.00~0.00  2.00~0.00
(Maximum value) 71.54~43.52      1.60~0.66      100.00~0.00  2.00~0.00
(Attr-based)    68.46~32.35      0.90~0.30      90.77~27.69  2.70~0.64
(NE:r,p)        81.54~36.92      1.80~0.40      100.00~0.00  2.00~0.00
(NX:s,p)        81.54~36.92      1.80~0.40      100.00~0.00  2.00~0.00
```

Figura 36. Trecho da segunda parte da saída do sistema DistAl

ANEXO D. O SISTEMA CONEB

Para facilitar os estudos e pesquisas e para viabilizar o estabelecimento de um ambiente computacional para experimentação, foi desenvolvido junto ao LIAA (Laboratório de Inteligência Artificial e Automação) da Universidade Federal de São Carlos um sistema que implementa vários algoritmos neurais construtivos. O sistema é chamado de CONEB (COnstructive NEural Builder); seu uso é fácil e sua interface é auto-explicativa. A tela inicial do sistema, mostrada na Figura 37, permite que o usuário escolha qual algoritmo usar: perceptron, pocket, pocket com modificação ratchet, tiling, upstart, tower, pyramid ou cascade-correlation. Os algoritmos perceptron, pocket e pocket com modificação ratchet, muito embora não sejam algoritmos construtivos, estão também disponibilizados no CONEB como parte do subsistema 3P.

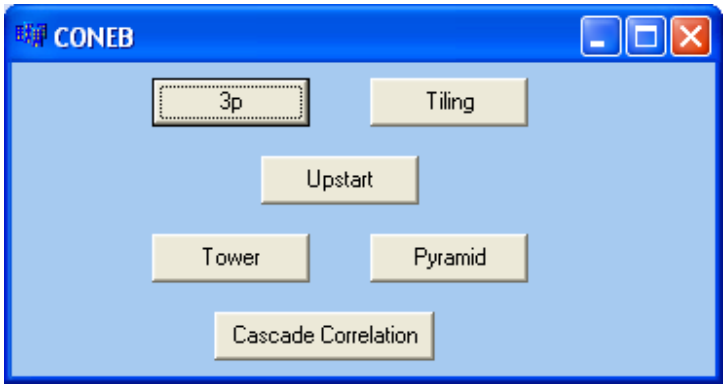


Figura 37. Tela inicial do sistema CONEB

D.1 CONSIDERAÇÕES INICIAIS

Para se usar o sistema, é necessário um arquivo texto, com extensão txt, onde devem estar gravadas as instâncias de treinamento ou teste. Esse arquivo deve estar em um formato compreendido pelo sistema. Para tal, os atributos de uma mesma instância devem ser

seqüenciais, na mesma linha e com apenas caracteres de espaço os separando. A classe da instância deve ser o último valor da linha e sempre deve possuir os valores 1 ou −1. Outras instâncias devem vir seqüencialmente nas linhas abaixo. A Tabela 74 ilustra como deve ser o arquivo de entrada para o sistema, relativo aos dados mostrados na Tabela 4.

Tabela 74. Arquivo de entrada
correspondente aos dados da Tabela 4

−1	−1	−1
−1	1	1
1	−1	1
1	1	−1

Os algoritmos disponibilizados no CONEB foram implementados de acordo com os pseudocódigos apresentados nessa dissertação. As diferenças entre os algoritmos e o código são mínimas, insignificantes e decorrentes do estilo de programação do autor dessa dissertação. As próximas subseções detalham as implementações de cada um dos algoritmos implementados no CONEB.

D.2 O SUBSISTEMA 3P – PERCEPTRON, POCKET E POCKET COM MODIFICAÇÃO RATCHET

A escolha da opção 3p na tela inicial do sistema CONEB leva o usuário ao subsistema 3P, que disponibiliza simultaneamente, o perceptron, pocket e pocket com modificação ratchet, como mostra a Figura 38.

Para a definição do arquivo de treinamento basta clicar com o botão direito do mouse sobre a opção **Abrir Arquivo de Treinamento**. É também necessário especificar o número de atributos e o número de instâncias de treinamento contidas nesse arquivo. Para o exemplo da Tabela 4, esses valores são 2 e 4 respectivamente. Feito isso, basta clicar com o botão direito do mouse sobre a opção **Gerar W.** Os vetores de peso e suas precisões de treinamento são automaticamente gerados pelos três algoritmos, perceptron, pocket e pocket com ratchet e, então, são exibidos na tela. É possível, também, definir o número máximo de iterações que devem ser executadas pelos algoritmos. O valor "default" para esse número é 1000.

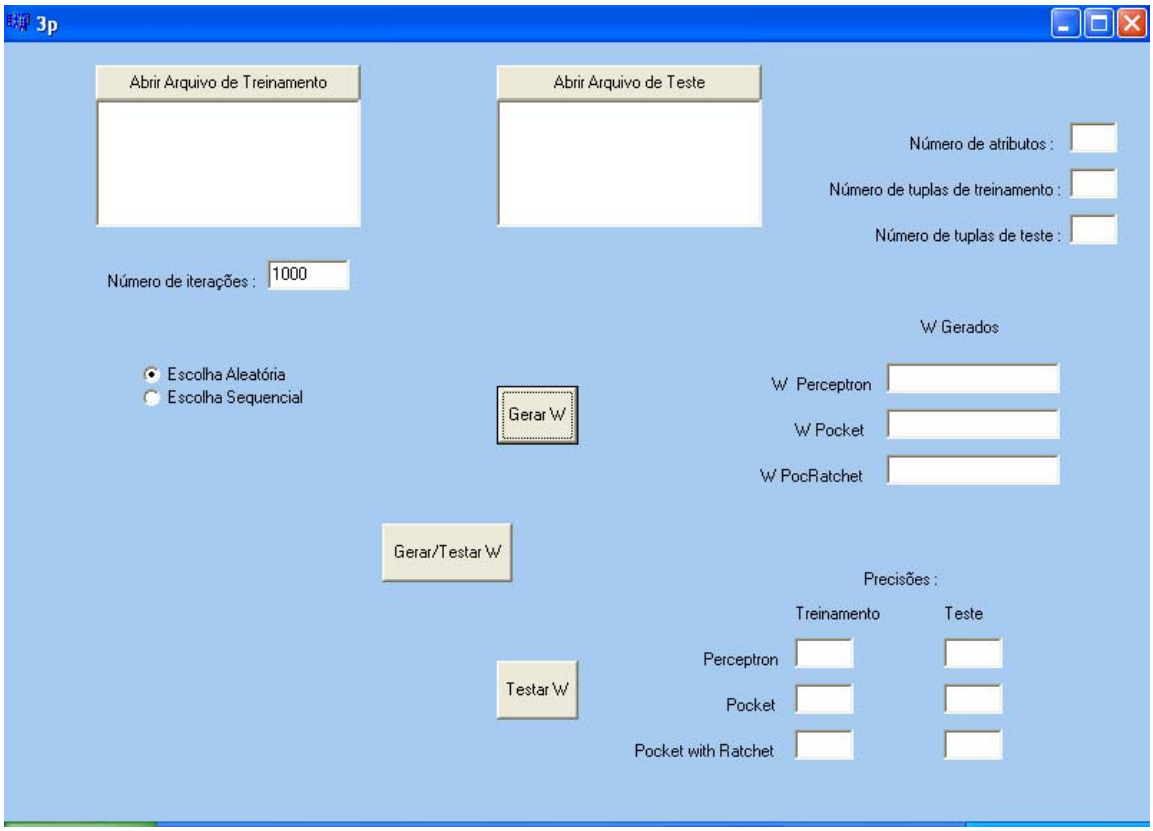


Figura 38. Tela da implementação dos algoritmos perceptron, pocket e pocket com modificação ratchet

O usuário pode abrir um arquivo de teste para a avaliação dos vetores de pesos gerados. Para isso, basta clicar com o botão direito do mouse sobre a opção **Abrir Arquivo de Teste**. É necessário também definir o número de instâncias de teste desse arquivo. Feito isso, basta clicar sobre a opção **Testar W** e as precisões de teste para os 3 algoritmos serão mostradas. Note que para usar um arquivo de teste é necessário, primeiro, gerar os vetores de peso (através da opção **Gerar W**).

Existe também a opção de usar um único arquivo para fazer treinamento e teste. Para tal, basta abrir um arquivo de treinamento e teste clicando com o botão direito do mouse sobre a opção **Abrir Arquivo de Treinamento**. O usuário deve especificar o número de instâncias totais desse arquivo (no campo **Número de tuplas de treinamento**), o número de atributos e clicar sobre o botão Gerar/Testar W**.** Quando isso é feito 80% das instâncias do arquivo são usadas para treinamento e as restantes são usadas para teste. A escolha das instâncias de treinamento e de teste é feita de uma maneira aleatória.

D.3 IMPLEMENTAÇÕES DOS ALGORITMOS TOWER, PYRAMID E TILING

A interface das implementações dos algoritmos tower, pyramid e tiling são bastante parecidas. Para se usar os algoritmos tower, pyramid ou tiling, é necessário clicar sobre os botões **tower**, **pyramid** ou **tiling**, respectivamente, na tela inicial do CONEB. A Figura 39 mostra a tela do algoritmo tower. A tela algoritmo pyramid é idêntica a do tower, exceto pelo nome que nela aparece. A tela do algoritmo tiling também é bastante parecida com a tela do tower. As diferenças são o nome do algoritmo e uma caixa de texto a mais que especifica o número máximo de neurônios auxiliares por camada desejado pelo usuário do sistema.

Para a definição do arquivo de treinamento basta clicar com o botão direito do mouse sobre a opção **Abrir Arquivo de Treinamento**. É também necessário especificar o número de atributos e o número de instâncias de treinamento contidas nesse arquivo. Para o exemplo da Tabela 4, esses valores são 2 e 4 respectivamente. Feito isso, basta clicar com o botão direito do mouse sobre a opção **Gerar W**. Os vetores de peso para, no máximo, as dez primeiras camadas e suas precisões de treinamento são automaticamente gerados pelos algoritmos pyramid, tower e tiling e, então, são exibidos na tela. Se algum critério de parada dos algoritmos for satisfeito antes da décima camada, são mostrados os vetores de peso e suas respectivas precisões de treinamento até a camada onde a condição foi satisfeita. É também possível limitar o número máximo de camadas, o número máximo de iterações executadas pelos algoritmos em cada camada e, no caso do algoritmo tiling, o número máximo de neurônios auxiliares por camada.

As precisões de treinamento para cada camada também são gravadas em um arquivo denominado Treinamento.txt. Esse arquivo pode ser encontrado no mesmo diretório do arquivo de treinamento. É necessário consultar esse arquivo apenas quando o número de camadas geradas exceder dez (os vetores de peso e suas precisões de classificação apenas são mostrados pela tela do sistema até a décima camada).

Os neurônios auxiliares gerados pelo algoritmo tiling não são mostrados na tela, mas o sistema grava os valores desses neurônios e a qual camada cada neurônio auxiliar

pertence em um arquivo denominado Auxiliares-tiling.txt. Esse arquivo pode ser encontrado no mesmo diretório do arquivo de treinamento.

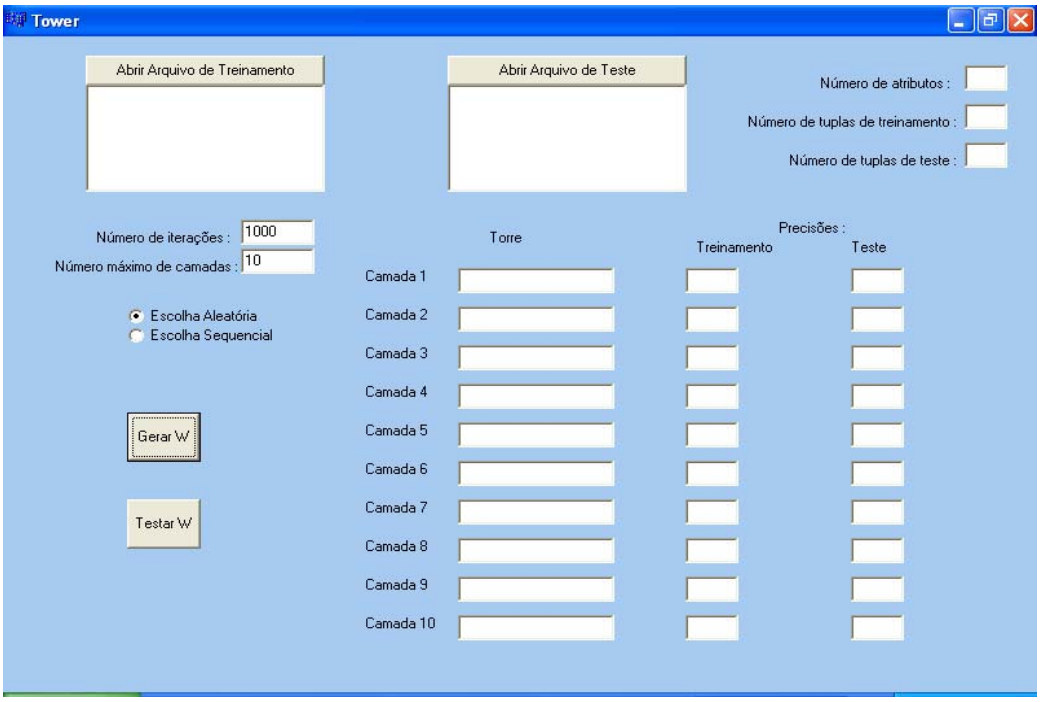


Figura 39. Tela do tower

O usuário pode abrir um arquivo de teste para a avaliação dos vetores de pesos gerados. Para isso, basta clicar com o botão direito do mouse sobre a opção **Abrir Arquivo de Teste**. É necessário também definir o número de instâncias de teste desse arquivo. Feito isso, basta clicar sobre a opção **Testar W** e as precisões de teste para cada camada do tower, pyramid ou tiling serão mostradas na tela. Note que para usar um arquivo de teste é necessário, primeiro, gerar os vetores de peso (através da opção **Gerar W**).

As precisões de teste para cada camada do algoritmo tower, pyramid ou tiling são gravadas em um arquivo denominado Teste.txt. Esse arquivo pode ser encontrado no mesmo diretório do arquivo de teste. É necessário consultar esse arquivo apenas quando o número de camadas geradas exceder dez (as precisões de teste são mostradas apenas para as dez primeiras camadas).

D.4 IMPLEMENTAÇÕES DOS ALGORITMOS UPSTART E CASCADE-CORRELATION

Para se usar as implementações dos algoritmos upstart ou cascade-correlation, é necessário clicar sobre os botões **upstart** ou **cascade-correlation**, respectivamente, da tela inicial do CONEB. Se o usuário clicar sobre o botão **upstart**, então o sistema irá solicitar ao usuário qual das duas abordagens do algoritmo ele pretende usar (ver Capítulo 5, Seções 5.2 e 5.3).

Para a definição do arquivo de treinamento basta clicar com o botão direito do mouse sobre a opção **Abrir Arquivo de Treinamento.** É também necessário especificar o número de atributos e o número de instâncias de treinamento contidas nesse arquivo. Para o exemplo da Tabela 4, esses valores são 2 e 4 respectivamente. Feito isso, basta clicar com o botão direito do mouse sobre a opção **Gerar W**.

Para o algoritmo cascade-correlation, a opção **Gerar W** gera os vetores de peso para, no máximo, as dez primeiras camadas (ie, com dez neurônios intermediários), e suas precisões de treinamento são automaticamente exibidas na tela. Se algum critério de parada do algoritmo cascade-correlation for satisfeito antes da décima camada, são mostrados os vetores de peso e suas respectivas precisões de treinamento até a camada onde a condição foi satisfeita. É possível também fornecer ao sistema o número máximo de neurônios intermediários, nível de paciência, precisão mínima tolerável e a constante de aprendizado.

Os neurônios intermediários não são exibidos na tela, mas podem ser encontrados em um arquivo denominado Auxiliares-cascade.txt que é gerado pelo sistema como conseqüência do uso do algoritmo correspondente. Esse arquivo pode ser encontrado no mesmo diretório do arquivo de treinamento.

As precisões de treinamento para cada camada do cascade-correlation também são gravadas em um arquivo denominado Treinamento-cascade.txt. Esse arquivo pode ser encontrado no mesmo diretório do arquivo de treinamento. É necessário consultar esse arquivo apenas quando o número de camadas geradas exceder dez (os vetores de peso e suas precisões de classificação apenas são mostrados pela tela do sistema até a décima camada).

Para o algoritmo upstart, a opção **Gerar W** gera o vetor de peso inicial (o primeiro vetor de pesos encontrado pelo algoritmo pocket com modificação ratchet que foi gerado sem os reforços) e final (o mesmo vetor de pesos, mas agora com os reforços) e suas precisões de treinamento são automaticamente exibidas na tela. É possível também fornecer ao sistema o número máximo de iterações por neurônio, o número máximo de neurônios auxiliares positivos e negativos e o número grande.

Os neurônios auxiliares não são exibidos na tela, mas podem ser encontrados em um arquivo denominado Auxiliares-upstart.txt que é gerado pelo sistema sempre que este é executado. Esse arquivo pode ser encontrado no mesmo diretório do arquivo de treinamento.

Para os dois algoritmos, o usuário pode abrir um arquivo de teste para a avaliação dos vetores de pesos gerados. Para isso, basta clicar com o botão direito do mouse sobre a opção **Abrir Arquivo de Teste**. É necessário também definir o número de instâncias de teste desse arquivo. Feito isso, basta clicar sobre a opção **Testar W** e as precisões de teste para cada camada do cascade-correlation e para os dois vetores de pesos do upstart (inicial e final) serão mostradas na tela. Note que para usar um arquivo de teste é necessário, primeiro, gerar os vetores de peso (através da opção **Gerar W**).

As precisões de teste para cada camada do algoritmo cascade-correlation são gravadas em um arquivo denominado Cascade_Teste.txt. Esse arquivo pode ser encontrado no mesmo diretório do arquivo de teste. É necessário consultar esse arquivo apenas quando o número de camadas geradas exceder dez (as precisões de teste são mostradas apenas para as dez primeiras camadas).

A função usada no cascade-correlation e a sua derivada são, respectivamente:

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

onde tanh(x) significa tangente hiperbólica de x. A Figura 40 mostra a tela de iteração com o upstart e a Figura 41 com o cascade-correlation.

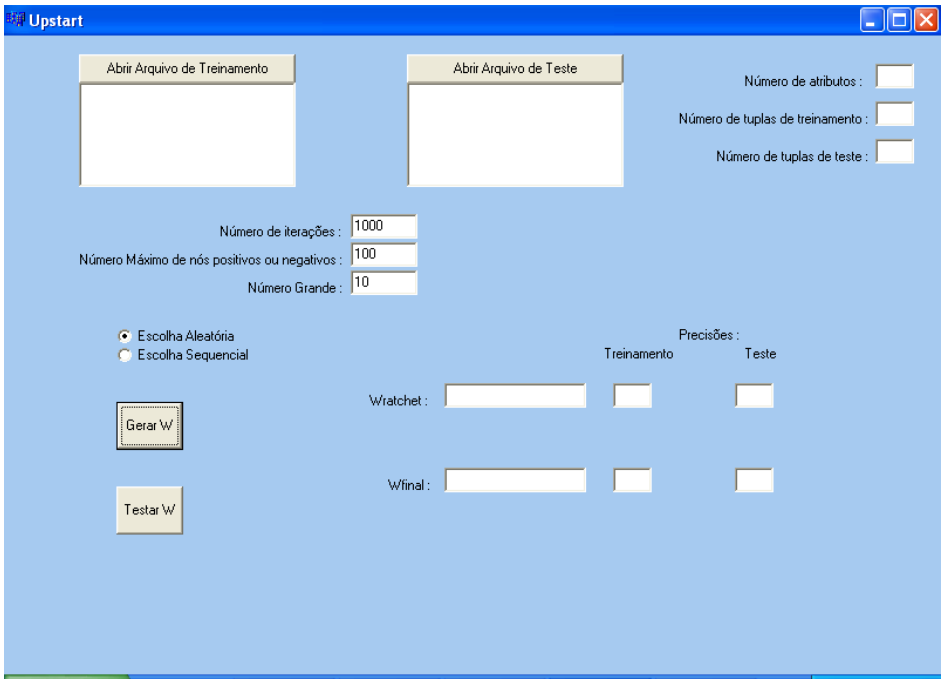


Figura 40. Tela do upstart

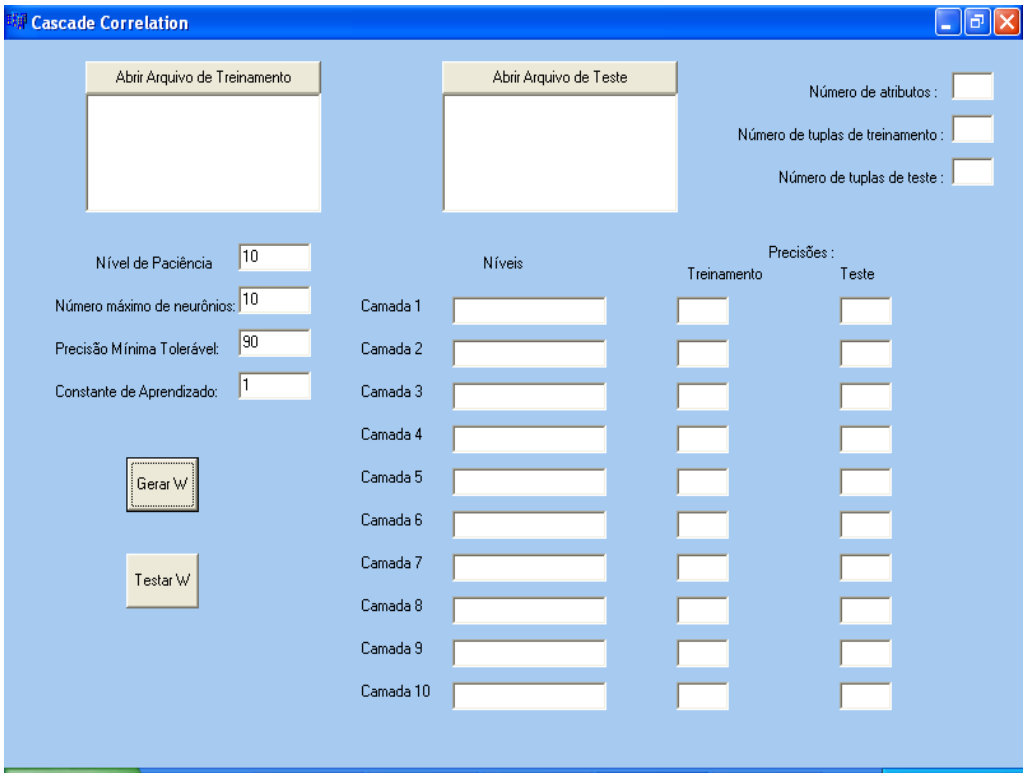


Figura 41. Tela do cascade-correlation

REFERÊNCIAS BIBLIOGRÁFICAS

[Amaldi & Guenin 1997] Amaldi, E.& Guenin, B. Two constructive methods for designing compact feedforward networks of threshold units. International Journal of Neural Systems, Vol. 8, N. 5 & 6, pp 629-645, 1997.

[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]. Irvine, CA: University of California, Department of Information and Computer Science, 1998.

[Block & Levin 1970] Block, H.D. & Levin, S.A. On the boundness of an iterative procedure for solving a system of linear inequalities. Proc. AMS 26, pp 229-235, 1970.

[Burgess 1994] Burgess, N. A constructive algorithm that converges for real-valued input patterns. International Journal of Neural Systems, 5(1), pp 59-66, 1994.

[Campbell 1997] Campbell, C. Constructive Learning Techniques for Designing Neural Network Systems. San Diego: Academic Press, 1997.

[Fahlman 1988] Fahlman, S. Faster Learning Variations on Back-Propagation: An Empirical Study. In Proceedings of the 1988 Connectionist Models Summer School, Morgan Kaufmann, 1988.

[Fahlman & Lebiere 1990] Fahlman, S. & Lebiere, C. The cascade correlation architecture. In Advances in neural information processing systems 2, Morgan Kaufman, San Maateo, CA, pp 524-532, 1990.

[Frean 1990a] Frean, M. The upstart algorithm: a method for constructing and training feedforward neural networks. Neural Computation, 2:198-209, 1990.

[Frean 1990b] Frean, M. Small Nets and Short Paths: Optimising Neural Computation. Ph.D. thesis, University of Edinburgh, Center for Cognitive Science, 1990.

[Frean 1992] Frean, M. A thermal perceptron learning rule, Neural Computation, 4:946-957, 1992.

[Gallant 1986] Gallant, S. I. Three constructive algorithms for network learning. Proc. Eighth Annual Conference of the Cognitive Science Society, Amherst, MA, pp 652-660, August 15-17, 1986.

[Gallant 1990] Gallant, S. I. Perceptron-based learning algorithm. IEEE Transactions on Neural Networks 1, no. 2, pp 179-192, June 1990.

[Gallant 1994] Gallant, S. I. Neural network learning & expert systems. The MIT Press London, England, 1994.

[Hrycej 1992] Hrycej, T. Modular learning in neural networks. Wiley, New York, 1992.

[Lavrac & Dzeroski 1994] Lavrac, N. & Dzeroski S. Inductive logic programming: Techniques and Applications. Prentice Hall, 1994.

[Marchand et al 1990] Marchand, M. & Golea, M. & Rujan, P. A convergence theorem for sequential learning in two layer perceptrons. Europhysics Letters, 11(6): pp 487-492, 1990.

[Mézard & Nadal 1989] Mézard, M. & Nadal. J. Learning feedforward networks: the tiling algorithm. J. Phys. A: Math Gen., 22:2191-2203, 1989.

[Mitchell 1997] Mitchell, T. M. Machine learning. The McGraw-Hill Companies, Inc., NY, 1997.

[Nadal 1989] Nadal, J. Study of a growth algorithm for a feedforward network. Internation Journal of Neural Systems, Vol.1, N. 1, pp 55-59, 1989.

[Nicoletti & Rocha 2002] Nicoletti, M. C. & Rocha, M. G. B. Árvores Perceptron, RT-02 002/2002.

[Palma Neto & Nicoletti 2003a] Palma Neto, L.G. & Nicoletti, M.C. Perceptron, algoritmo pocket e algoritmo pocket com modificação ratchet. RT-DC 001, 2003.

[Palma Neto & Nicoletti 2003b] Palma Neto, L.G. & Nicoletti, M.C. Aprendizado Neural Construtivo – o Algoritmo DistAl. RT-DC 002, 2003.

[Palma Neto & Nicoletti 2003c] Palma Neto, L.G. & Nicoletti, M.C. Aprendizado Neural Construtivo Usando Algoritmos Tower, Pyramid e Tiling. RT-DC 004, 2003.

[Palma Neto & Nicoletti 2004] Palma Neto, L.G. & Nicoletti, M.C. Aprendizado Neural Construtivo Usando os Algoritmos Upstart e Cascade-Correlation. RT-DC submetido para avaliação, 2004.

[Palma Neto et al 2003] Palma Neto, L.G. & Figueira, L. B. & Nicoletti, M. C. Using a family of perceptron-based neural networks for detecting central vestibular system problems. International Conference on Machine Learning and Applications (ICMLA), June, LA, USA, pp 193-199, 2003.

[Palma Neto et al 2004] Palma Neto, L.G. & Figueira, L. B. & Nicoletti, M. C. Application of constructive neural networks in identifying central vestibular system problems. Submetido para publicação junto à International Journal of Artificial Intelligence in Medicine.

[Parekh 1998] Parekh, R. Constructive learning: inducing grammars and neural networks. PhD dissertation. Iowa State University, Ames, Iowa, 1998.

[Parekh et al 1997] Parekh, R. & Yang, Y. & Honovar, 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, 1997.

[Parekh et al 2000] Parekh, R. & Yang, Y. & Honovar, V. Constructive Neural-Network Learning Algorithms for Pattern Classification. IEEE Transactions on Neural Networks, Vol. 11, No.2, March 2000.

[Poulard 1995] Poulard, H. Barycentric correction procedure: a fast method of learning threshold units. In: Proc. of WCNN'95 (vol. 1), Washington D.C., pp 710–713, 1995.

[Rosenblatt 1962] Rosenblatt, F. Principles of Neurodynamics: Perceptron and the Theory of Brain Mecanisms. Washington DC: Spartan, 1962.

[Rumelhart et al 1986] Rumelhart, D. E. & Hinton, G. E. & Williams, R. J. Learning Internal Representations by Error Propagation. In Rumelhart, D. E. & McClelland, J. I. Parallel Distributed Processing: Explorations and Microstructure of Cognition, MIT Press, 1986.

[Śmieja 1991] Śmieja, F.J. Neural Network Constructive Algorithms: Trading Generalization for learning Efficiency? November, 1991.

[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 Sciences Department, University of Wisconsin-Madison, 1991.

[Thrun et al 1991] Thrun, S.B. & Bala, J. & Bloedorn, E. & Bratko, I. & Cestnik, B. & Cheng, J. & De Jong, K. & Dzeroski, S. & Fahlman, S.E. & Fisher, D. & Hamann, R. & Kaufman, R. & Keller, S. & Kononenko, I. & Kreuziger, J. & Michalski, R. S. & Mitchell, T. & Pachowicz, P. & Reich H. Vafaie, Y. & Van de Welde, W. & Wenzel, W. & Wnek, J. & Zhang, J. The MONK's Problems - A Performance Comparison of Different Learning Algorithms. Technical Report CS-CMU-91-197, Carnegie Mellon University in Dec. 1991.

[Volpini et al 2002] Volpini, P. & Figueira, L. B. & Colafemina, J. F. & Roque, A. C. A neural network-based system for the diagnosis of central vestibular lesion. In: Valafar, F. (ed.). Proc. of the International Conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences-METMBS'02, CSREA Press, pp. 29-33, 2002.

[Wilson & Martinez 1997] Wilson, D. R. & Martinez, T. R. Improved heterogeneous distance functions. Journal of Artificial Intelligence Research 6, pp 1-34, 1997.

[Yang et al 1997] Yang, J. & Parekh, R. & Honavar, V. DistAl: An inter-pattern distance-based constructive learning algorithm. Tech. Rep. ISU-CS-TR 97-05, Iowa State University, 1997.

[Yang et al 1999] Yang, J. & Parekh, R. & Honovar, V. DistAl: An inter pattern distance based constructive learning algorithm. Intelligent Data Analysis 3, pp 53-73, 1999.