Universidade Federal de Alagoas
Instituto de Computação

Trabalho de Conclusão de Curso

Desenvolvimento de Rede Neural SOM: Um Estudo de Caso para Segmentação de Perfis

Manoel Jorge Ribeiro Neto
manoeljorge.neto@gmail.com

Orientadores:

Evandro de Barros Costa
Rômulo Nunes de Oliveira

Maceió, Março de 2007

Manoel Jorge Ribeiro Neto

Desenvolvimento de Rede Neural SOM: Um Estudo de Caso para Segmentação de Perfis

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciências da Computação do Instituto de Computação da Universidade Federal de Alagoas.

Orientadores:

Evandro de Barros Costa

Rômulo Nunes de Oliveira

Maceió, Março de 2007

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciências da Computação do Instituto de Computação da Universidade Federal de Alagoas, aprovada pela comissão examinadora que abaixo assina.

Evandro de Barros Costa - Orientador
Instituto de Computação
Universidade Federal de Alagoas

Rômulo Nunes de Oliveira - Orientador
Campus Arapiraca
Universidade Federal de Alagoas

Luis Cláudius Coradine - Examinador
Instituto de Computação
Universidade Federal de Alagoas

Roberta Vilhena Vieira Lopes - Examinador
Instituto de Computação
Universidade Federal de Alagoas

Maceió, Março de 2007

Resumo

Neste trabalho, apresenta-se um estudo sobre Redes Neurais com foco nas redes SOM (*Self-Organizing Map*, ou Mapa Auto-Organizável). Decorrente deste estudo, desenvolve-se um sistema de *software* capaz de agrupar os diferentes perfis de um domínio. O sistema de *software* criado é composto por um componente que implementa o algoritmo SOM. A visualização do mapa gerado pelo algoritmo, desta forma, é efetuada por um Mapa Contextual, por meio do qual é possível identificar os agrupamentos do domínio e, desta maneira, segmentar os diferentes grupos de perfis pertinentes ao mesmo.

Abstract

In this work, it is presented a study about Neural Networks with focus in the SOM networks (Self-Organizing Map). Decurrent of this study, it is developed a software system capable to group the different profiles of a domain. The developed system software is composed by a component that implements the SOM algorithm. The visualization of the map generated by the algorithm, by the way, is effected by a Contextual Map, by means of which it is possible to identify the groupings of the domain and, in this way, to segment the different groups of profiles pertinent it.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Lista de Códigos

Capítulo 1

Introdução

Antes mesmo da popularização da Informática, muitas instituições ofereciam serviços destinados a usuários em massa. Como exemplo, há os bancos, que oferecem muitos serviços – entre eles a oferta de crédito – cujo conhecimento, e possível classificação, dos perfis de seus clientes é de grande importância para se evitar prejuízos, melhorar os serviços ofertados etc.

Com a evolução da tecnologia, os serviços que antes eram ofertados localmente, hoje o são praticamente sem restrições geográficas. Além disso, permitiu-se o surgimento de novos serviços, que se beneficiam das novas possibilidades proporcionadas pelas tecnologias recentes. Exemplos desses serviços são lojas virtuais, *sites* de relacionamento, bancos *on-line*, portais de conteúdo etc. Em comum, eles possuem a possibilidade de agregar milhares (ou até milhões) de usuários, tornando humanamente impossível a tarefa de analisar os mais diversos perfis, classificando-os de acordo com suas similaridades. Vale ressaltar que nem sempre se sabe quais são os parâmetros de similaridade, o que torna essa tarefa ainda mais complexa.

Os perfis, numa categorização mais geral, representam os objetos (pessoas, documentos, recursos etc) pertinentes a um domínio e cujas características consideradas importantes para o mesmo são especificadas.

Diante disso, a Inteligência Artificial oferece muitas técnicas para abordar esse problema supramencionado. Entre elas, inclui-se a utilização de redes SOM (*Self-Organizing Map*), que é um tipo especial de Rede Neural Artificial, capaz de realizar agrupamentos e classificações em mapas bidimensionais (Haykin, 1999). No processo de aprendizagem em uma rede SOM, os neurônios competem entre si, e como resultado obtém-se uma saída organizada que pode ser utilizada como matriz indexadora. Dessa forma, cria-se um mecanismo que facilite a segmentação de perfis de um domínio.

Os objetivos do presente trabalho são, portanto, pormenorizar o assunto, com o estudo de Redes Neurais SOM e, como estudo de caso, implementar um sistema segmentador de perfis. Esse tipo de sistema é demandado em vários domínios de aplicação, nos quais o

conhecimento a respeito dos perfis é de fundamental importância. Exemplos de domínios assim encontram-se no comércio, na medicina, na indústria, entre outros.

A presente monografia, além deste capítulo, está estruturada do seguinte modo:

- No Capítulo 2, há uma introdução sobre Redes Neurais, constando de seções sobre características das Redes Neurais, representação do neurônio artificial, arquiteturas de rede e formas de aprendizagem de máquina.

- No Capítulo 3, há a apresentação do *Mapa Auto-Organizável de Kohonen*, onde características deste tipo especial de Rede Neural são explicitadas e o algoritmo de aprendizagem é apresentado. Além disso, são descritos nesse capítulo métodos de visualização do mapa produzido pelo SOM e considerações sobre os parâmetros utilizados no algoritmo.

- No Capítulo 4, o problema da segmentação de perfis é abordado, onde há a sua descrição. Além disso, como proposta para resolver o problema, um sistema de *software* é criado com o objetivo de implementar o algoritmo SOM, que servirá como ferramenta que agrupe os diferentes perfis de algum domínio de aplicação, ajudando na tarefa de segmentá-los. A documentação e alguns experimentos feitos com o *software* implementado também estão nesse capítulo.

- No Capítulo 5, há a conclusão do trabalho.

- No Apêndice A, o código-fonte do sistema de *software* criado é mostrado. Dessa forma, caso o leitor tenha interesse em implementar o algoritmo SOM, pode utilizá-lo como referência.

Capítulo 2

Redes Neurais

2.1 Introdução

Inspirados no funcionamento do cérebro, cuja capacidade de processamento de informações é creditada a redes de neurônios (onde cada neurônio tem a constituição ilustrada na Figura 2.1), os primeiros pesquisadores de IA dedicaram parte de seus trabalhos à criação de redes neurais artificiais. Hebb, McCullock e Pitts, Turing, von Neumann, Minsky e McCarthy são exemplos de cientistas que estudaram o assunto (Russell & Norvig, 2004; Luger, 2004).

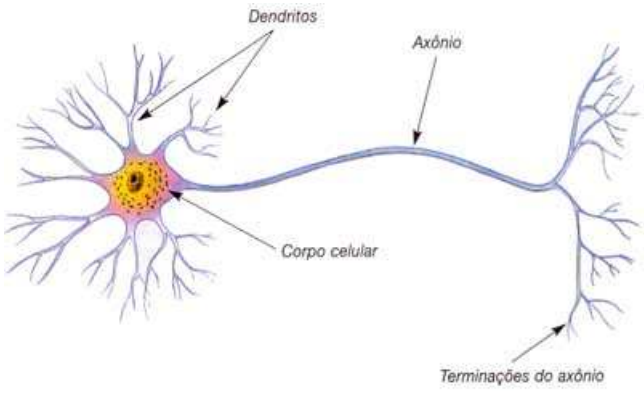

Figura 2.1: Representação de um neurônio natural

Segundo Luger (2004), os modelos de Redes Neurais (também conhecidos por **conexionismo**, **processamento paralelo distribuído** e **computação neural**) trabalham com o conceito de que a inteligência provém de sistemas de componentes simples, interativos (os neurônios biológicos ou artificiais), que passam por um processo de aprendizado, no qual as conexões entre os componentes são ajustadas. Nesses sistemas, o processamento é paralelo e distribuído entre os componentes, sendo que todos os neurônios do conjunto processam as suas entradas de forma independente e simultânea. Esses sistemas também tendem a degradar-se suavemente, pois a informação e o processamento são distribuídos

através dos componentes. As Redes Neurais, dessa forma, são capazes de aprender e, então, generalizar (generalização refere-se à capacidade da Rede Neural de produzir respostas razoáveis para entradas que não foram dadas no processo de aprendizado). Haykin (1999) dá a seguinte definição de uma Rede Neural[1]:

> *Uma Rede Neural é um processador distribuído maciçamente paralelo feito de simples unidades de processamento, que tem a propensão natural de guardar conhecimento experimental e torná-lo disponível para uso. Ela assemelha-se ao cérebro em dois apectos:*
>
> 1. *O conhecimento de um domínio é adquirido pela rede durante o processo de aprendizado;*
> 2. *As forças de conexões entre neurônios, conhecidas como pesos sinápticos, são usadas para guardar o conhecimento adquirido.*

Nos modelos neurais, contudo, há um forte caráter representacional tanto para a criação de parâmetros de entrada como na interpretação dos valores de saída. Na construção de uma Rede Neural, os padrões do mundo devem ser codificados como vetores numéricos, por meio de algum esquema criado para isso, devido ao fato de que as interconexões entre os neurônios também são representadas dessa forma (como será visto na Seção 2.3). A escolha de um esquema de codificação pode definir o sucesso ou o fracasso da rede em aprender (Luger, 2004).

Vale ressaltar, entretanto, que ainda há um grande caminho a percorrer para que se consiga construir sistemas computacionais que simulem bem o funcionamento do cérebro humano.

Segundo de Oliveira (2006), as Redes Neurais são indicadas quando o domínio e solução de determinado problema atendem aos seguintes requisitos:

- O padrão a ser aprendido não pode ser traduzido com facilidade por funções matemáticas;
- Há um conjunto de treinamento que represente bem o domínio do problema;
- Geralmente, a semântica para o conhecimento aprendido não é de grande importância, apenas sua aplicação. Isso decorre do fato de que o conhecimento aprendido pela rede está distribuído entre as "sinapses" dos neurônios, de forma que não é fácil dar uma semântica ao conhecimento representado.

Segundo Luger (2004), algumas tarefas para as quais as Redes Neurais adequam-se são:

[1]Neste trabalho, os termos "Rede Neural" e "neurônio" irão referir-se, respectivamente, às Redes Neurais artificiais e aos neurônios artificiais.

- *classificação*, para classificar os integrantes de um grupo;
- *reconhecimento de padrões*, para identificar os padrões nos dados;
- *predição*, identificando causas a partir de efeitos;
- *otimização*, achando a "melhor" organização de restrições;
- *filtragem de ruído*, separando o sinal de ruído de fundo em um sinal.

2.2 Benefícios das Redes Neurais

Na seção anterior, foi visto que uma Rede Neural é uma estrutura paralela, maciçamente distribuída, com capacidade para aprender e, então, generalizar. Com essas características, uma Rede Neural é possível de resolver problemas complexos (em larga escala) atualmente intratáveis (Haykin, 1999).

O uso de Redes Neurais, segundo Haykin (1999), oferece as seguintes propriedades e capacidades:

1. *Não-linearidade:* Um neurônio artificial pode ser linear ou não. Uma Rede Neural feita de interconexões de neurônios não-lineares é também não-linear. Além disso, a não-linearidade é distribuída por toda a rede. A não-linearidade é uma propriedade muito importante, especialmente se o mecanismo responsável pelo sinal de entrada (por exemplo, sinal de voz) for inerentemente não-linear.

2. *Mapeamento de entrada-saída:* No aprendizado supervisionado, um conjunto de exemplos de treinamento são apresentados à Rede Neural. Cada exemplo consiste em um sinal de entrada e uma saída correspondente, onde os pesos sinápticos da rede são modificados para que se minimize a diferença entre a resposta da rede e a saída do exemplo. O treinamento da rede é repetido várias vezes, onde os mesmos exemplos são apresentados em ordens diferentes, até que a rede atinja um estado estável, onde não haja alterações significativas nos pesos sinápticos quando um novo exemplo é dado. Deste modo, a rede aprende através dos exemplos construindo um mapeamento de entrada-saída para o problema.

3. *Adaptatividade:* Redes Neurais têm a capacidade de adaptar seus pesos sinápticos em resposta às mudanças no ambiente. Em particular, uma Rede Neural pode ser re-treinada para lidar com mudanças do domínio. É bom enfatizar que adaptatividade nem sempre significa robustez do sistema. Por exemplo, um sistema adaptativo com constantes temporais curtas pode mudar rapidamente e, então, tender a responder a entradas falsas, causando uma degradação drástica na perfomance do sistema. Para beneficiar-se totalmente da adaptividade, as principais constantes temporais devem

ser longas o suficiente para o sistema ignorar entradas falsas e, ainda, responder às mudanças importantes do ambiente. O problema descrito aqui refere-se ao *dilema da plasticidade-estabilidade*.

4. *Resposta de evidências:* Na classificação de padrões, uma Rede Neural pode ser designada para prover informações não apenas sobre qual padrão selecionar, mas também sobre a certeza da escolha feita. Essa infomação pode ser útil para eliminar padrões ambíguos, melhorando a perfomance de classificação da rede.

5. *Informação contextual:* O conhecimento está distribuído pelos componentes da Rede Neural. Cada neurônio da rede é potencialmente afetado pela atividade global de todos os neurônios da rede. Conseqüentemente, informação contextual é tratada com naturalidade por uma Rede Neural.

6. *Tolerância a erros:* Uma Rede Neural implementada em *hardware* tem a capacidade de tolerância a erros do sistema, de forma que seu desempenho degrada-se suavemente perante situações adversas antes de uma falha catastrófica.

7. *Possibilidade de implementação usando VLSI (Very-Large-Scale-Integrated):* A natureza de ser maciçamente paralela de uma Rede Neural torna-a potencialmente rápida para a computação em certas tarefas. Essa característica das Redes Neurais é apropriada para a sua implementação usando tecnologia *VLSI*.

8. *Uniformidade de análise e projeto:* Em todos os domínios que as Redes Neurais atuam, basicamente a mesma notação é usada. Dessa forma, podem ser observadas as seguintes características:

   - Os neurônios, mesmo de formas diferentes, são um componente comum a todas as Redes Neurais;
   - Dessa forma, é possível compartilhar teorias e algoritmos de aprendizagem em aplicações diversas de Redes Neurais;
   - Redes modulares podem ser criadas sem "emendas" entre os módulos.

9. *Analogia com a Neurobiologia:* O projeto de uma Rede Neural é motivado pela analogia com o cérebro, que é a prova viva de que o processamento paralelo tolerante a falhas não é apenas possível mas também rápido e poderoso. Dessa forma, neurobiólogos estudam Redes Neurais artificiais com o objetivo de ter uma ferramenta para a interpretação dos fenômenos neurobiológicos, enquanto que engenheiros dedicam-se à Neurobiologia para obter novas idéias de como resolver problemas mais complexos do que aqueles que sistemas computacionais convencionais conseguem resolver.

2.3 Representação de um neurônio

O neurônio artificial, conforme representado na Figura 2.2, é a base das Redes Neurais, sendo que os neurônios de uma rede são conectados por vínculos orientados. Um vínculo de uma unidade k para uma unidade l serve para propagar a ativação a_k de k até l. Um neurônio, é constituído pelos seguintes elementos básicos (Russell & Norvig, 2004; Haykin, 1999; Luger, 2004):

1. Um *conjunto de sinapses*, com cada sinapse associada a um peso. A um sinal de entrada x_j, na entrada da sinapse j conectada ao neurônio k, é feita a multiplicação de seu valor pelo peso w_{kj}. Vale ressaltar que o conjunto de sinapses pode ser representado por um vetor numérico $(x_1, \ldots, x_m)$, de dimensão m (onde m é o *número de sinapses*);

2. Um *adicionador*: para somar os sinais de entrada (já devidamente multiplicados pelos pesos w_{kj}). A operação efetuada é uma *combinação linear*;

3. Uma *função de ativação*, para limitar a amplitude da saída do neurônio. Segundo (Russell & Norvig, 2004), a função de ativação tem as seguintes características:

   - Ela tem a função de "ativar" o neurônio (saída próxima de +1), quando as entradas "corretas" forem apresentadas e "desativá-lo" (saída próxima a 0) ao se receber entradas "erradas";
   - É necessário que a função seja *não-linear*, para evitar que a rede entre em colapso.

   A saída da função de ativação geralmente está no intervalo fechado unitário [0,1] ou, alternativamente, [-1,1].

Além das sinapses, é também incluído um peso especial, denotado por b_k, cujo efeito é aumentar ou diminuir a entrada da função de ativação (o peso b_k pode ser chamado de *peso de desvio* (Russell & Norvig, 2004)). Esse peso pode ser externo aos demais, ou contido em uma sinapse com entrada fixa igual a 1 e peso b_k.

Matematicamente, caso b_k esteja "fora" das sinapses, um neurônio k pode ser descrito pelo seguinte par de equações:

$$u_k = \sum_{j=1}^{m} w_{kj} x_j \text{ e } y_k = \varphi(u_k + b_k) \tag{2.1}$$

Onde: $x_1, x_2, \ldots, x_m$ são os *sinais de entrada*; $w_{k1}, w_{k2}, \ldots, w_{km}$ são os *pesos sinápticos* do neurônio k; u_k é a *saída da combinação linear*; b_k é o *peso de desvio*; $\varphi(u_k + b_k)$ é a *função de ativação* e y_k é o *sinal de saída* do neurônio.

Caso b_k esteja "entre" as sinapses, as equações matemáticas que descrevem um neurônio são:

$$v_k = \sum_{j=0}^{m} w_{kj} x_j \text{ e } y_k = \varphi(v_k) \tag{2.2}$$

Onde: v_k é chamado de *potencial de ativação* (ou *campo local induzido*); $x_0 = +1$ e $w_{k0} = b_k$

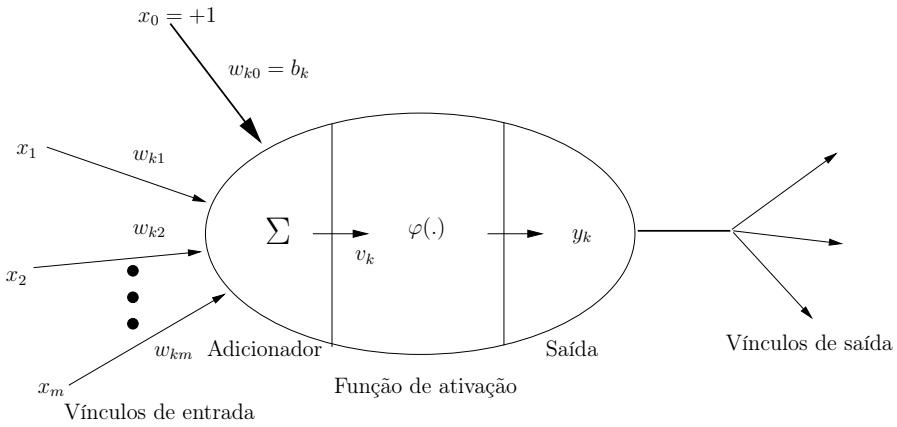

Figura 2.2: Modelo matemático simples para um neurônio

Apesar dessas duas formas de representar os neurônios serem aparentemente diferentes, elas são matematicamente equivalentes (Haykin, 1999).

2.3.1 Tipos de função de ativação

A função de ativação define a saída do neurônio em termos do valor v (que é o resultado obtido pelo adicionador). Em Haykin (1999), são identificados três tipos básicos de função de ativação:

1. *Função de limiar:* que é dada pela seguinte função:

$$\varphi(v) = \begin{cases} 1, & \text{se } v \geq 0 \\ 0, & \text{se } v < 0 \end{cases} \tag{2.3}$$

   O neurônio que utiliza essa função de ativação é conhecido como *modelo de McCulloch-Pitts*, em homenagem ao trabalho pioneiro de McCulloch & Pitts (1943).

2. *Semi-linear:*

$$\varphi(v) = \begin{cases} 1, & \text{se } v \geq +\frac{1}{2} \\ v, & \text{se } -\frac{1}{2} < v < +\frac{1}{2} \\ 0, & \text{se } v \leq -\frac{1}{2} \end{cases} \tag{2.4}$$

3. *Função Sigmóide:* A função sigmóide (também chamada de **função logística** e cujo gráfico lembra a letra S) é a forma mais comum de função de ativação utilizada em Redes Neurais. Ela apresenta um balanço entre o comportamento linear e não linear, além de ser diferenciável (propriedade muito importante para ser utilizada nos algoritmos de treinamento). Um exemplo de função sigmóide é o seguinte:

$$\varphi(v) = \frac{1}{1 + exp(-av)} \tag{2.5}$$

E a sua derivada é definida por:

$$\frac{d\varphi}{dv} = a\varphi(v)\left[1 - \varphi(v)\right] \tag{2.6}$$

Onde a é um *parâmetro de inclinação*. Variando o valor de a, altera-se a inclinação da função. Quando $a \to \infty$, a função tende a tornar-se uma função de limiar. Os gráficos da função sigmóide e de sua derivada estão na Figura 2.3, onde o gráfico à esquerda representa a função sigmóide e o à direita, a sua derivada.

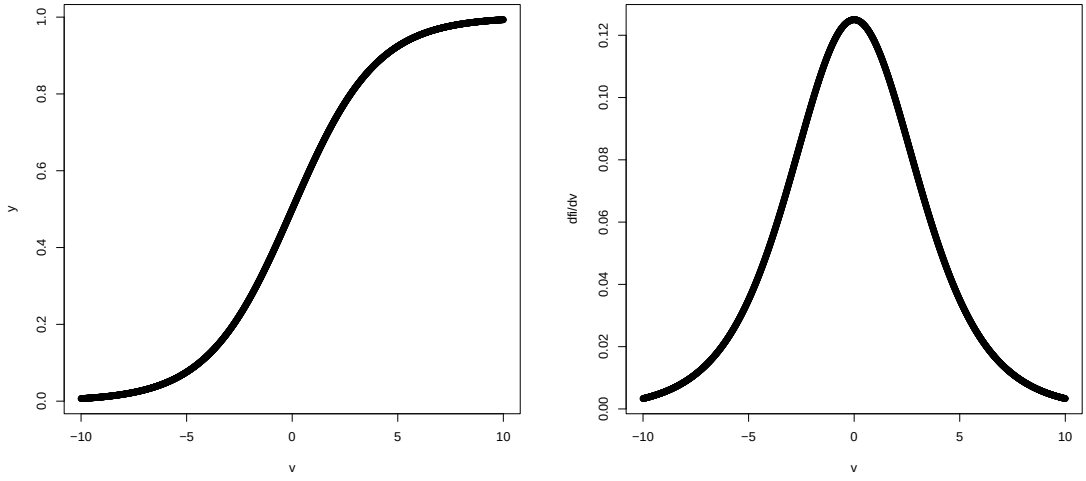

Figura 2.3: Gráficos da função sigmóide e de sua derivada

O modelo de neurônio apresentado nesta seção é apenas um dentre os que são utilizados em projetos de Redes Neurais. No entanto, os diferentes tipos de neurônios empregados em Redes Neurais têm forte semelhança com este apresentado.

2.4 Arquiteturas de rede

A maneira como os neurônios em uma Rede Neural estão estruturados está intimamente ligada com o algoritmo de aprendizado utilizado para treiná-la. Basicamente, existem duas categorias principais de estruturas de Redes Neurais: **redes de alimentação direta** e **redes recorrentes**. Uma rede de alimentação direta representa uma função de sua entrada atual, tendo seu estado definido pelos seus pesos sinápticos. Uma rede recorrente, por sua vez, tem suas entradas alimentadas pelas suas saídas, formando um sistema dinâmico que pode apresentar um estado estável, ou até exibir oscilações, ou até mesmo resultar num sistema caótico (Russell & Norvig, 2004).

As redes de alimentação direta geralmente são organizadas em camadas, sendo que os neurônios de uma determinada camada recebem informações apenas dos neurônios da camada imediatamente precedente. A organização por camadas, entretanto, também pode ser aplicada em outros modelos de Redes Neurais.

Redes recorrentes têm suas respostas dependentes de seus estados anteriores, possibilitando que elas tenham a habilidade de possuir memória de curto prazo, tornando-as interessantes para modelos de cérebro, mas também mais complexas de se entender.

Nas próximas subseções, algumas características dessas diferentes arquiteturas de rede serão vistas.

2.4.1 Redes de alimentação direta de uma única camada (perceptrons)

Uma Rede Neural de alimentação direta de uma única camada (ou rede de *perceptron*), caracteriza-se por possuir apenas uma camada de neurônios ligada à entrada, como exibido na Figura 2.4.

Um perceptron com função de ativação de limiar pode representar funções booleanas diversas, tais como: E, OU, NÃO, MAIORIA etc, desde que elas sejam *linearmente separáveis* (são funções onde é possível traçar um hiperplano no espaço de entrada que separa as diferentes respostas). Um exemplo de função que **não** é linearmente separável é a XOU (OU EXCLUSIVO), que um perceptron não consegue resolver. Os perceptrons de sigmóide, por sua vez, podem representar separadores lineares "temporários" (Russell & Norvig, 2004).

Apesar dessa limitação, os perceptrons de limiar têm suas vantagens. Entre elas, pode-se afirmar que há um algoritmo de aprendizagem simples que adaptará um perceptron de limiar a qualquer conjunto de treinamento linearmente separável, efetuando a memorização do padrão (Russell & Norvig, 2004). Dessa forma, essas redes tornam-se apropriadas para trabalhar com problemas desse tipo.

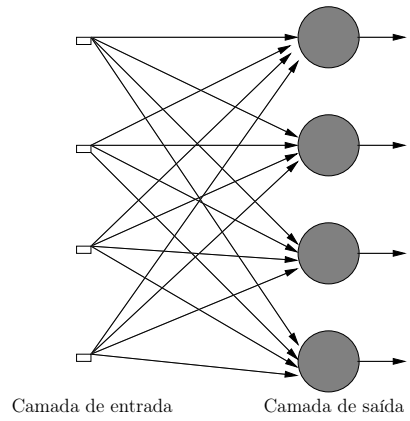

Figura 2.4: Exemplo de Rede Neural de camada simples

2.4.2 Redes de alimentação direta de múltiplas camadas

Essa classe de Redes Neurais tem como característica a presença de uma ou mais camadas ocultas, tal como exibido na Figura 2.5. Com a adição de uma ou mais camadas ocultas suficientemente grandes, a rede torna-se habilitada para representar qualquer função com exatidão arbitrária (Russell & Norvig, 2004; Haykin, 1999), mas é difícil saber quais funções podem ser representadas com uma estrutura de rede específica. O problema de escolher o número de unidades ocultas para representar determinada função ainda não está bem compreendido (Russell & Norvig, 2004).

As redes de múltiplas camadas podem ser *totalmente conectadas* (onde cada neurônio de determinada camada está ligado com todos os neurônios da camada adjacente) ou *parcialmente conectadas* (onde algumas sinapses são removidas).

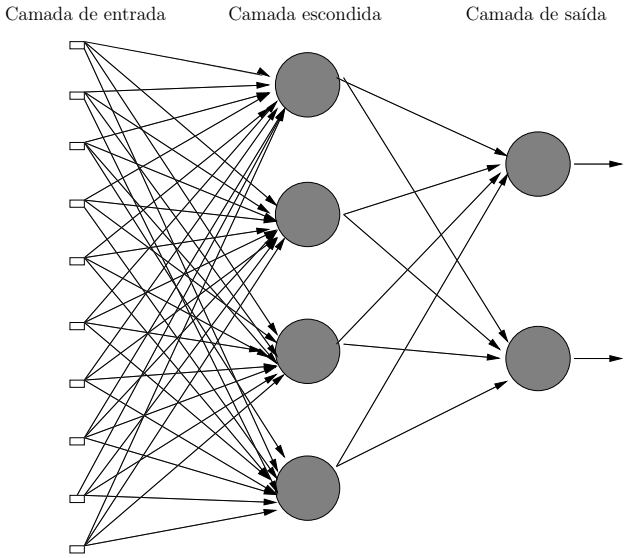

Figura 2.5: Exemplo de rede de múltiplas camadas

2.4.3 Redes recorrentes

Uma rede recorrente, cujo exemplo de uma rede desse tipo é representado na Figura 2.6, tem como característica a existência de pelo menos um ciclo. Com relação à disposição dos neurônios, as redes recorrentes podem apresentar neurônios escondidos ou não.

A presença de ciclos, como dito anteriormente, traz um grande impacto na capacidade de aprender da rede e também na sua perfomance. Além disso, os ciclos envolvem o uso de *elementos de espera* (denotados por z^{-1}), que resulta num comportamento dinâmico não-linear (Haykin, 1999).

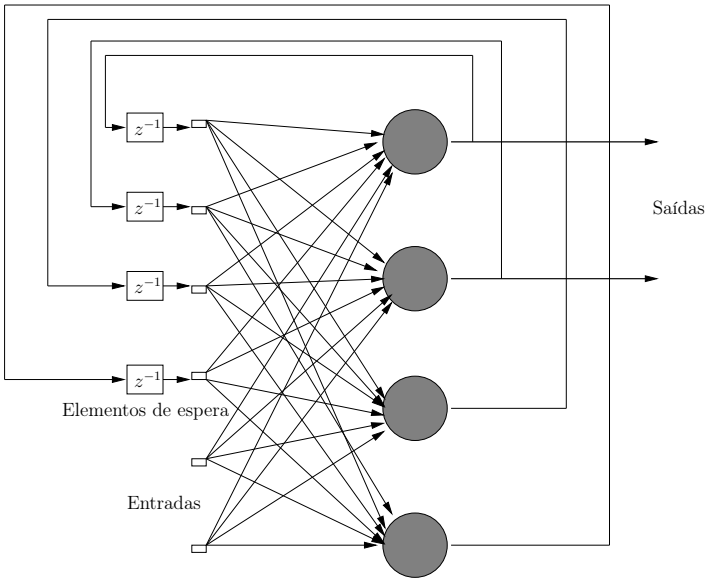

Figura 2.6: Exemplo de rede recorrente com neurônios escondidos

Além dessas estruturas apresentadas, existem modeles de Redes Neurais que apresentam outras estruturas (geralmente derivadas das apresentadas aqui) para realizar as mais diversas tarefas. Entre esses modelos, estão as redes SOM, que serão descritas com detalhes no Capítulo 3.

2.5 Formas de aprendizagem de máquina

Como visto nas seções precedentes, as Redes Neurais têm a capacidade de aprender a partir de um ambiente abordado. Os algoritmos que fazem o treinamento da rede consistem em ajustar os pesos sinápticos dos neurônios, a partir de um conjunto de dados de entrada, a fim de que a rede passe a produzir respostas razoáveis para entradas novas (no Capítulo 3 será apresentado um desses algoritmos, que servirá para treinar uma rede SOM). A aprendizagem de máquina geralmente ocorre de três formas (Russell & Norvig, 2004; Luger, 2004):

- *Supervisionada:* A aprendizagem supervisionada consiste em gerar uma função a

partir de exemplos de entradas e saídas. Os algoritmos que implementam essa forma de aprendizagem assumem a existência de uma entidade externa que faça a classificação dos exemplos de treinamento, gerando uma saída apropriada para cada exemplo.

- *Não-supervisionada:* A aprendizagem não-supervisionada consiste em aprender a partir dos padrões dos exemplos de entrada quando os mesmos não possuem valores de saída específicos. Um sistema que se utiliza de aprendizagem puramente não-supervisionada não possui a habilidade de aprender o que fazer, pois não possui nenhuma informação sobre qual ação é correta ou qual estado é desejável (Russell & Norvig, 2004). Porém, devido à sua capacidade de reconhecer padrões, essa forma de aprendizagem é bastante útil quando há a necessidade de conhecer mais sobre um ambiente desconhecido.

- *Por reforço:* A aprendizagem por reforço é a forma mais geral de aprendizado, a qual consiste em aprender usando medidas de "recompensa". O agente[2] não sabe exatamente qual ação tomar, mas ele acaba descobrindo que determinado conjunto de ações oferecem maior recompensa. Por outro lado, ele também acaba descobrindo quais ações diminuem a recompensa, evitando-as. Geralmente, a aprendizagem por reforço envolve a questão de conhecer como o ambiente funciona (Russell & Norvig, 2004).

Vale ressaltar que, utilizando-se Redes Neurais, é possível implementar as três formas de aprendizagem descritas: Para a aprendizagem supervisionada, pode ser citado o algoritmo *Backpropagation* (Retropropagação), utilizado em redes de múltiplas camadas (Russell & Norvig, 2004; Luger, 2004; Haykin, 1999). Para a aprendizagem não-supervisionada, um exemplo é o *Mapa Auto-Organizável de Kohonen* (SOM), que é objeto de estudo deste trabalho. E, para o aprendizado por reforço, as *Redes Neurais Associativas* de Barto et al. (1995), com a sua utilização em aproximações de funções (Russell & Norvig, 2004).

[2]Um agente é tudo o que pode ser considerado capaz de perceber seu ambiente por meio de sensores e de agir sobre esse ambiente através de atuadores (Russell & Norvig, 2004).

Capítulo 3

Mapas Auto-Organizáveis

3.1 Introdução

O *Mapa Auto-Organizável de Kohonen* (SOM, de *Self-Organizing Map*) (Kohonen, 1982, 2001) é um tipo especial de Rede Neural, baseado em aprendizado competitivo e não-supervisionado. Nele, os neurônios de saída competem entre si pelo direito de representar o dado apresentado, sendo que apenas um é considerado vencedor, segundo a regra do "vencedor-pega-tudo" (Haykin, 1999; Luger, 2004; Kohonen, 2006). Após o processo competitivo, o neurônio vencedor e seus vizinhos têm seus pesos sinápticos modificados no sentido do dado apresentado (geralmente em proporções diferentes), resultando num mapa topologicamente correto do ambiente após a apresentação dos dados de aprendizado repetidas vezes e em ordens diversas.

No Mapa Auto-Organizável[1], os neurônios são dispostos em um arranjo discreto finito geralmente unidimensional ou bidimensional (dimensões maiores são aceitas, mas não são comumente utilizadas) e totalmente conectados com a entrada, possibilitando mapear um conjunto de dados contidos em R^D para o espaço discreto de saída do arranjo dos neurônios em R^P, conforme representado na Figura 3.1. As relações de similaridade entre os dados e os neurônios se dão pelas relações entre os vetores de pesos dos neurônios (que também estão em R^D).

Segundo Zuchini (2003), o SOM realiza uma projeção não-linear do espaço de dados de entrada – em R^D – para o espaço de dados do arranjo – em R^P –, executando uma redução dimensional quando $P < D$. Dessa forma, o algoritmo de aprendizagem do SOM procura preservar ao máximo a topologia do espaço original, ou seja, procura fazer com que os neurônios vizinhos no arranjo tenham vetores de pesos que representem as semelhanças entre os dados do ambiente. A redução da dimensionalidade, em conjunto com a preservação da topologia dos dados, tornam a Rede SOM própria para ampliar a

[1]Auto-organização refere-se ao processo pelo qual estruturas com ordem global são obtidas a partir de interações locais entre os elementos da estrutura (Zuchini, 2003).

capacidade de análise de agrupamentos e de classifição de dados pertencentes a espaços de elevada dimensão. Com essas características, o algoritmo SOM é apropriado como *ferramenta de mineração de dados* (Kohonen, 2006).

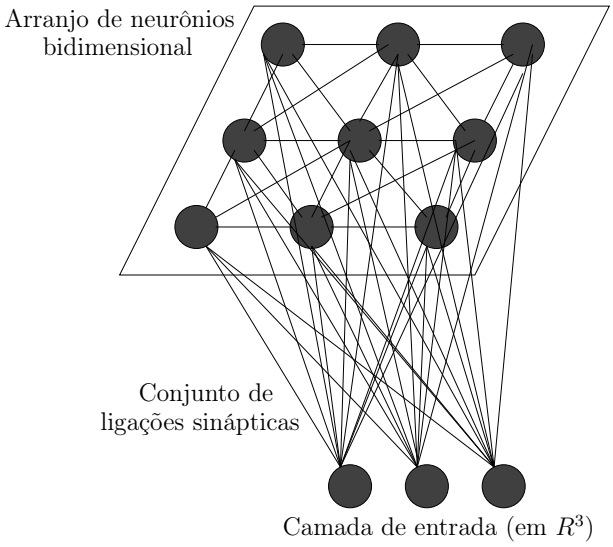

Figura 3.1: Exemplo de Mapa Auto-Organizável

3.1.1 Utilização do SOM

O Mapa Auto-Organizável de Kohonen, originalmente proposto em Kohonen (1982), tinha como objetivo ser um modelo de mapas mentais. Por certas características do modelo original, como a regra "vencedor-pega-tudo", que produz apenas respostas singulares localizadas, e também por não ser possível analisar *misturas superpostas* de padrões de entrada, o SOM não é ainda aceito como modelo biológico de mapas mentais (Kohonen, 2006). O modelo original, contudo, é apropriado para mineração de dados.

Segundo Kohonen (2006), atualmente existem cerca de 7000 artigos científicos publicados que falam sobre o SOM. Em Kaski et al. (1998); Oja et al. (2002) há a bibliografia de grande parte desses artigos.

Entre os artigos cujo assunto é o SOM (ou modelos modificados dele), há o de Honkela et al. (1998), que propõe uma nova arquitetura SOM, chamada WEBSOM, desenvolvida para exploração de textos em mineração de dados. Na pesquisa por artigos cujo trabalho envolva a utilização de redes SOM, foram encontrados artigos para *análise temporal de dados em supermercados* (Lingras et al., 2005), para *extração de regras a partir de um SOM* (Malone et al., 2005), para *segmentação de imagens coloridas* (Yeo et al., 2005), entre outros. Isso mostra que o algoritmo SOM pode ser utilizado em vários domínios, com apenas sua utilização ou em conjunto com outras técnicas.

3.2 Algoritmo

O objetivo principal de um Mapa Auto-Organizável é transformar dados de um ambiente de dimensão arbitrária em um mapa discreto unidimensional ou bidimensional, de tal forma que o mapa tenha uma ordenação topológica que represente o ambiente.

O algoritmo responsável pela formação de Mapas Auto-Organizáveis começa inicializando os vetores de pesos sinápticos dos neurônios do arranjo. Uma forma de fazer isso é inicializar os pesos sinápticos com pequenos valores aleatórios (com a restrição de que os vetores devem ser diferentes). Após a Rede Neural ser inicializada, os dados do conjunto de treinamento são apresentados várias vezes e em ordens diversas (isso pode ser feito escolhendo aleatoriamente o dado que será apresentado), com um número de iterações predeterminado ou até quando não ocorrerem mudanças significativas no mapa. Para a formação de um Mapa Auto-Organizável, há três processos principais, que são (segundo Haykin (1999)):

1. *Competição:* Para cada dado de entrada, os neurônios calculam seus valores em relação a ele e competem entre si pelo direito de representá-lo. O neurônio que melhor representar o dado – segundo alguma métrica pré-estabelecida – é considerado vencedor;

2. *Cooperação:* O neurônio vencedor determina a localização espacial de sua vizinhança, cujos neurônios integrantes serão excitados de acordo com alguma regra;

3. *Adaptação sináptica:* Os neurônios excitados (incluindo o vencedor) são habilitados a alterar seus valores individuais para que se pareçam mais com o dado de entrada, de acordo com alguma fórmula pré-estabelecida.

Nas próximas subseções, esses processos são vistos com mais detalhes.

3.2.1 Processo competitivo

Seja m a dimensão do espaço de entrada. Cada padrão de entrada pode ser denotado da seguinte forma:

$$x = [x_1, x_2, \ldots, x_m]^T \tag{3.1}$$

O vetor de pesos sinápticos de cada neurônio tem a mesma dimensão do espaço de entrada. O conjunto de vetores de pesos sinápticos é denotado por:

$$w_j = [w_{j1}, w_{j2}, \ldots, w_{jm}]^T, \quad j = 1, 2, \ldots, l \tag{3.2}$$

Onde l é o *número total de neurônios do arranjo.*

Para cada x pertencente ao conjunto de entradas e tomado aleatoriamente, todos os neurônios calcularão a distância de seu vetor de pesos w_j com o dado x, de acordo com alguma métrica pré-estabelecida, que no caso da distância euclidiana é dada por[2]:

$$d(x, w_j) = \|x - w_j\| = \sqrt{\sum_{k=1}^{m} |x_k - w_{jk}|^2}, \quad j = 1, 2, \ldots, l \tag{3.3}$$

O neurônio vencedor é aquele que possui a menor distância com o dado x (portanto, é o que melhor representa o dado). Dessa forma, o neurônio vencedor (denotado por $i(x)$) é determinado pela seguinte condição:

$$i(x) = \arg\min_j \|x - w_j\|, \quad j = 1, 2, \ldots, l \tag{3.4}$$

3.2.2 Processo cooperativo

O neurônio vencedor, para viabilizar o requisito de que os neurônios próximos a ele também tenham seus vetores de pesos sinápticos ajustados na direção do dado, deve excitar os neurônios pertencentes a sua vizinhança (definindo graus de vizinhança e excitação diferentes, de acordo com alguma função pré-estabelecida). Segundo Haykin (1999), o processo cooperativo se dá da seguinte forma:

Seja $h_{j,i}$ a *função de vizinhança* definida para o SOM, centrada no neurônio vencedor i, e um conjunto de neurônios excitados por ele (denotados por valores de j).

Seja $d_{j,i}$ a *distância* no arranjo entre o neurônio vencedor e algum neurônio j. Então, a função de vizinhança $h_{j,i}$ é uma função da distância $d_{j,i}$, e tem de satisfazer dois requisitos:

- A função de vizinhança é *simétrica* em relação ao seu ponto de máxima (definido por $d_{j,i} = 0$). Ou seja, o seu valor máximo está no neurônio vencedor i;
- A amplitude da função de vizinhança $h_{j,i}$ decresce monotomicamente, tendendo a zero quando $d_{j,i} \to \infty$. Isto é necessário para a convergência do SOM.

Uma escolha típica de função de vizinhança $h_{j,i}$, que satisfaz esses dois requisitos, é uma função gaussiana da forma (um exemplo de gráfico dessa função está na Figura 3.2):

$$h_{j,i(x)} = \exp\left(-\frac{d_{j,i}^2}{2\sigma^2}\right) \tag{3.5}$$

Onde o parâmetro σ define a *largura da função de vizinhança*.

[2]O neurônio da Rede SOM, se comparado com o apresentado na Seção 2.3, difere-se pela sua "função de ativação", que é definida pela sua medida de distância, e pela utilização da mesma, que serve para competir com os demais neurônios da rede e também para saber a sua relação de proximidade com o dado apresentado e com seus vizinhos.

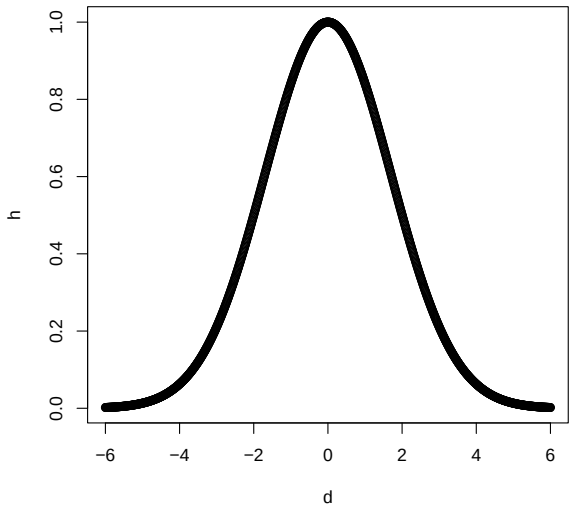

Figura 3.2: Gráfico da função de vizinhança

A distância entre os neurônios i e j, que está na dimensão do arranjo, é definida por:

$$d_{j,i} = \|r_j - r_i\| \tag{3.6}$$

Onde r_j é um vetor discreto que define a *posição no arranjo do neurônio excitado* e r_i define a *posição do neurônio vencedor*. Ambos os vetores estão contidos no espaço discreto de saída.

Outra característica do algoritmo SOM é que a largura da vizinhança deve "encolher" ao longo do treinamento, com o objetivo de alcançar a convergência do mapa. Esse requisito é satisfeito fazendo o tamanho de σ diminuir com o tempo. Uma função bastante utilizada para σ variar de acordo com o tempo discreto n é a *função de decaimento exponencial* (Haykin, 1999):

$$\sigma(n) = \sigma_0 \exp\left(-\frac{n}{\tau_1}\right), \qquad n = 0, 1, 2, \ldots \tag{3.7}$$

Onde σ_0 é o *valor inicial* de σ no início do algoritmo SOM e τ_1 é uma *constante temporal*.

Com a inclusão da função $\sigma(n)$, a função de vizinhança (agora mudando sua forma ao longo do tempo) é a que segue:

$$h_{j,i(x)}(n) = \exp\left(-\frac{d_{j,i}^2}{2\sigma^2(n)}\right), \qquad n = 0, 1, 2, \ldots \tag{3.8}$$

3.2.3 Processo adaptativo

No processo adaptativo, os neurônios excitados têm seus vetores de pesos modificados em direção ao vetor de entrada x. Para tanto, é necessário a existência de alguma regra para que se garanta a convergência do mapa de forma auto-organizada. Segundo Haykin (1999); Zuchini (2003), o novo valor do vetor de pesos sinápticos do j-ésimo neurônio no instante de tempo $(n+1)$ é definido pela seguinte equação:

$$w_j(n+1) = w_j(n) + \eta(n) h_{j,i(x)}(n) \left[x - w_j(n)\right], \quad j = 1, 2, \ldots, l \tag{3.9}$$

Onde $\eta(n)$ define a *taxa de aprendizado* e $h_{j,i(x)}(n)$ define o *grau de adaptação do neurônio* em relação ao vencedor (como visto na Seção 3.2.2).

O parâmetro de aprendizado $\eta(n)$, a fim de que haja a convergência do SOM, deve variar gradualmente com o tempo (Haykin, 1999). Normalmente, $\eta(n) \to 0$ quando $n \to \infty$ (Zuchini, 2003). Uma função utilizada para $\eta(n)$ é a seguinte:

$$\eta(n) = \eta_0 \exp\left(-\frac{n}{\tau_2}\right), \quad n = 0, 1, 2, \ldots \tag{3.10}$$

Onde η_0 é o valor de η no início do algoritmo SOM e τ_2 é outra *constante temporal*.

Duas fases para o processo adaptativo: Ordenação e Convergência

Para que o algoritmo SOM, a partir de um estado inicial de desorganização, evolua para um estado organizado de padrões de ativação, é necessário dividir o algoritmo de treinamento em duas fases: *ordenação* e *convergência* (Haykin, 1999; Zuchini, 2003). A seguir, há algumas características dessas fases (segundo Haykin (1999)):

1. *Ordenação:* É a fase em que ocorre a *ordenação topológica do arranjo*. Ela dura pelo menos as 1000 primeiras iterações do algortimo SOM, com valores relativamente grandes para $\eta(n)$ e $h_{j,i(x)}(n)$. Esses parâmetros são ajustados como segue:

   - O parâmetro de aprendizado $\eta(n)$ deve começar com um valor próximo a $0,1$, que depois decrescerá gradualmente, mas sempre acima de $0,01$. Para esse requisito, devem ser feitas as seguintes escolhas:

   $$\eta_0 = 0,1 \text{ e } \tau_2 = 1000 \tag{3.11}$$

   - A função de vizinhança $h_{j,i(x)}(n)$ deve englobar a maior parte dos neurônios (centrados no vencedor), no começo do algoritmo SOM, e encurtar-se gradativamente com o tempo. Para isso, uma forma de ajustar σ_0 é torná-lo igual ao "raio" do arranjo e τ_1 deve ser ajustado com o seguinte valor:

$$\tau_1 = \frac{1000}{\ln \sigma_0} \tag{3.12}$$

2. *Convergência:* Essa fase é necessária para que haja um "ajuste fino" do arranjo, propiciando respostas precisas da Rede Neural para entradas posteriores. Muitas vezes, o número de iterações pode ir até a casa de dezenas de milhares ou mais para que se obtenha bons resultados, com sua duração muito maior do que a primeira fase. Para essa fase, os parâmetros devem ter as seguintes características:

   - O parâmetro $\eta(n)$ deve ser mantido com um valor pequeno (na ordem de $0,01$). No entanto, sem nunca ir para zero. A função de decaimento exponencial apresentada garante esse requisito;
   - A função de vizinhança $h_{j,i(x)}(n)$ deve conter apenas os neurônios mais próximos do vencedor, podendo reduzir-se para apenas os vizinhos exatamente adjacentes ao neurônio vencedor ou apenas ele.

3.3 Resumo do algoritmo

Segundo Haykin (1999); Zuchini (2003), as partes integrantes do algoritmo SOM (descritas na Seção 3.2) são:

- Um *espaço contínuo de padrões de entrada do ambiente*, contido num conjunto V (de dimensão arbitrária);
- Um *arranjo de neurônios*, que define um espaço discreto de saída geralmente unidimensional ou bidimensional, contendo l neurônios;
- Uma *função de vizinhança* $h_{j,i(x)}(n)$, que varia com o tempo e define o grau de vizinhança em relação ao neurônio vencedor $i(x)$;
- Um *parâmetro de aprendizado* $\eta(n)$, que começa com um valor inicial η_0 e diminui gradualmente, mas nunca chegando ao valor zero.

No algoritmo (mostrado de forma resumida no Algoritmo 1), os pesos sinápticos são atualizados toda vez que um elemento do espaço de entrada é apresentado, sendo por isso conhecido como *incremental* ou "*on-line*". A sua utilização é recomendada quando não se tem antecipadamente todos os dados de treinamento disponíveis, além de ter uma implementação computacional mais barata e exigir menos memória (Zuchini, 2003). Em outra versão do algoritmo, as atualizações são realizadas apenas no final de uma *época de treinamento*[3]. Essa outra versão é conhecida como *em lote* ou "*batch*".

[3]Uma época de treinamento refere-se ao momento em que todos os itens de dados são apresentados exatamente uma vez à Rede Neural (Zuchini, 2003).

Algoritmo 1 Algoritmo de treinamento do SOM (incremental)

1: Inicialize os vetores de pesos ($w_j,\ j = 1, 2, \ldots, l$) dos neurônios do arranjo com pequenos valores aleatórios para os pesos sinápticos (com a restrição de que os vetores devem ser diferentes). Faça $n = 0$, o número de iterações $n\_it = 0$ e inicialize $\sigma(n\_it)$ e $\eta(n\_it)$;
2: **while** O número máximo de iterações pré-estabelecido não tiver sido atingido **do**
3: Faça $V' = V$;
4: **while** $V' \neq \oslash$ **do**
5: Selecione aleatoriamente um vetor de dados x do conjunto V';
6: Faça $V' = V' - \{x\}$;
7: Selecione o neurônio vencedor $i(x) = \arg\min_j \|x - w_j\|,\ j = 1, 2, \ldots, l$;
8: Atualize os neurônios do arranjo, segundo a equação:
9: $w_j(n+1) = w_j(n) + \eta(n\_it)h_{j,i(x)}(n\_it)\,[x - w_j(n)],\ j = 1, 2, \ldots, l$;
10: Faça $n = n + 1$;
11: **end while**
12: Faça $n\_it = n\_it + 1$, $n = 0$ e ajuste $\sigma(n\_it)$ e $\eta(n\_it)$;
13: **end while**

Segundo Zuchini (2003), o algoritmo incremental é sensível à ordem em que os dados são apresentados e também à taxa de aprendizado. O algorimo em lote, por sua vez, não possui esse problema e contorna o segundo. Em vez de atualizar o seu vetor de pesos sinápticos todas as vezes que um item de dados é apresentado, no algoritmo em lote cada neurônio acumula as contribuições parciais de cada vetor x apresentado, segundo a equação:

$$\Delta w_j(n+1) = \Delta w_j(n) + h_{j,i(x)}(n\_it)\,[x - w_j(n)], \qquad j = 1, 2, \ldots, l \tag{3.13}$$

Onde n_it é o *número de épocas de treinamento*.

Ao final de uma época, os pesos sinápticos são atualizados conforme a seguinte equação:

$$w_j(n\_it + 1) = w_j(n\_it) + \frac{1}{l}\eta(n\_it)\Delta w_j, \qquad j = 1, 2, \ldots, l \tag{3.14}$$

O algoritmo em lote é descrito de forma resumida no Algoritmo 2 (Zuchini, 2003).

Normalização dos dados e dos vetores de pesos sinápticos

Quando o SOM é utilizado para fins de mineração de dados e sabemos muito pouco ou nada sobre o ambiente, é necessário **normalizar** os dados de entrada (deixá-los com norma igual a 1), para evitar que um atributo domine sobre os demais, podendo haver outros de maior importância sobrepujados (Zuchini, 2003). Além disso, os pesos sinápticos também devem ser normalizados a cada vez que são atualizados (Kohonen, 2006). Segundo Kohonen (2006), a *normalização euclidiana* pode ser utilizada com esses propósitos, que é definida por:

Algoritmo 2 Algoritmo de treinamento do SOM (em lote)

1: Inicialize os vetores de pesos $(w_j,\ j = 1, 2, \ldots, l)$ dos neurônios. Faça $n = 0$, o número de iterações $n\_it = 0$, as contribuições parciais $\Delta w_j = 0,\ j = 1, 2, \ldots, l$ e inicialize $\sigma(n\_it)$. A taxa de aprendizado $\eta(n\_it)$ recebe um valor fixo e pequeno (0,5 para a fase de ordenação e 0,05 para a fase de convergência);
2: **while** O número máximo de iterações pré-estabelecido não tiver sido atingido **do**
3: Faça $V' = V$;
4: **while** $V' \neq \oslash$ **do**
5: Selecione um vetor de dados x do conjunto V';
6: Faça $V' = V' - \{x\}$;
7: Selecione o neurônio vencedor $i(x) = \arg\min_j \|x - w_j\|,\ j = 1, 2, \ldots, l$;
8: Calcule a contribuição parcial do dado em cada neurônio, segundo a equação:
9: $\Delta w_j(n+1) = \Delta w_j(n) + h_{j,i(x)}(n\_it)\,[x - w_j(n)],\ j = 1, 2, \ldots, l$;
10: Faça $n = n + 1$;
11: **end while**
12: Atualize os pesos sinápticos dos neurônios, segundo a equação:
13: $w_j(n\_it + 1) = w_j(n\_it) + \frac{1}{l}\eta(n\_it)\Delta w_j,\ j = 1, 2, \ldots, l$;
14: Faça $n\_it = n\_it + 1$; $n = 0$; $\Delta w_j = 0,\ j = 1, 2, \ldots, l$ e ajuste $\sigma(n\_it)$ e $\eta(n\_it)$;
15: **end while**

$$\left(v_1^{'}, v_2^{'}, \ldots, v_m^{'}\right) = \frac{(v_1, v_2, \ldots, v_m)}{\sqrt{\sum_{k=1}^{m} v_k^2}} \tag{3.15}$$

Onde $\left(v_1^{'}, v_2^{'}, \ldots, v_m^{'}\right)$ é o *vetor normalizado*.

Dessa forma, o algoritmo de treinamento do SOM deve inicializar os pesos sinápticos já normalizados. Além disso, a cada vez que algum dado de entrada for apresentado, este deve ser normalizado. O mesmo deve ser feito toda vez que há alterações nos vetores de pesos sinápticos dos neurônios, deixando-os sempre normalizados.

3.4 Propriedades do SOM

Após o algoritmo SOM ter convergido, é formado um mapa que representa a *projeção não-linear do espaço de entrada*. Dessa forma, o algoritmo SOM faz uma transformação não-linear representada da seguinte forma (Haykin, 1999):

$$\Phi : X \rightarrow A \tag{3.16}$$

Onde:

- Φ denota o *mapa gerado pelo SOM*, que representa a transformação não-linear do espaço de entrada;

- X denota o *espaço contínuo de entrada* (de dimensão arbitrária), com topologia definida pelas relações métricas entre os elementos;
- A denota o *espaço discreto de saída*, cuja topologia é determinada pelas relações de proximidade entre os neurônios vizinhos no arranjo.

Para saber qual a localização no mapa de um padrão $x \in X$, a transformação que o SOM convergido representa retorna o neurônio vencedor $i(x)$, contido em A. A partir da localização espacial no mapa do neurônio vencedor, é possível conhecer mais a respeito do dado apresentado.

Segundo Haykin (1999), o mapa resultante do algoritmo SOM tem algumas características importantes:

1. **Aproximação do espaço de entrada:** *O mapa Φ, representado pelo conjunto de vetores de pesos sinápticos $\{w_j\}$ no espaço de saída A, provê uma boa aproximação do espaço de entrada X.*

   O objetivo principal do algoritmo SOM é armazenar um conjunto grande de vetores de entrada $x \in X$, descobrindo um conjunto de padrões $w_j \in A$, que provê uma boa aproximação do espaço de entrada X, executando uma redução dimensional semelhante ao processo de quantização vetorial quando a dimensão de A é menor do que a de X (Zuchini, 2003). Desta forma, dados que não foram apresentados ao SOM durante o seu treinamento, mas que possuem probabilidades de ocorrência parecidas com as dos dados de treinamento, podem ser avaliados. Com essa característica, o algoritmo SOM *possui a habilidade de generalizar* (Zuchini, 2003; Haykin, 1999).

2. **Ordenamento topológico:** *O mapa Φ é ordenado topologicamente, de tal forma que a localização espacial do neurônio no arranjo corresponde a um domínio particular ou propriedade dos padrões de entrada.*

   Com essa propriedade, é possível verificar os diferentes domínios dos dados de entrada, visto que cada região do mapa representa alguma característica do espaço de entrada. Com isso, padrões semelhantes serão mapeados em regiões próximas no mapa, possibilitando a detecção dos agrupamentos formados.

3. **Equiparação com a densidade:** *O mapa Φ reflete as variações nas estatísticas da distribuição dos dados do espaço de entrada: regiões no espaço de entrada X, cujos elementos são encontrados com maior probabilidade de ocorrência, são mapeadas em domínios maiores e com melhor resolução no espaço de saída A do que regiões em X cujos elementos aparecem com menor probabilidade de ocorrência.*

   Dessa forma, é possível detectar se os agrupamentos de padrões aparecem com maior ou menor probabilidade verificando-se os tamanhos dos agrupamentos, que

serão maiores e com melhor resolução para os agrupamentos cujos elementos tenham maior probabilidade de ocorrência. Por outro lado, dados extremos[4] situam-se em regiões "esparsas", com agrupamentos pequenos ou em regiões onde não é possível identificar agrupamentos.

Para as provas dessas características, consulte Haykin (1999).

3.5 Interpretação do mapa produzido pelo SOM

Após a formação do mapa pelo algoritmo de aprendizagem do SOM, o resultado gerado é um conjunto de neurônios contidos num arranjo geralmente bidimensional, cujos vizinhos contêm vetores de pesos sinápticos que representam as semelhanças dos dados do ambiente, formando um mapa topologicamente correto do mesmo. Para verificar qual a localização no mapa de algum dado do ambiente, basta apresentá-lo à rede e a sua posição será a do neurônio que possuir o vetor de pesos sinápticos com menor distância do dado, de acordo com a possibilidade que o SOM tem de efetuar uma projeção não-linear do espaço de entrada do dado para o espaço de saída do arranjo de neurônios (como explicitado nas Seções 3.1 e 3.4). Contudo, para que se consiga interpretar o conteúdo do mapa, é necessário a utilização de *algum método de visualização* que auxilie em tal tarefa. Entretanto, a total interpretação do mapa não é garantida (por exemplo, inferir regras a partir do mapa produzido pelo SOM) com a utilização dos métodos aqui descritos, pois sua organização não possui semântica direta, sendo necessário algum método adicional para isso (de Oliveira, 2006; Malone et al., 2005).

Para a visualização do mapa, muitos métodos são utilizados. Entre eles, existe a *visualização do mapa como uma grade elástica*, onde os neurônios vizinhos estão posicionados mais longe ou mais perto de acordo a distância de seus vetores de pesos sinápticos (Haykin, 1999). Entre outros métodos, há a *Matriz-U* e o *Mapa Contextual*, ambos utilizados em arranjos bidimensionais, que serão descritos nas próximas subseções.

3.5.1 Matriz-U

A *matriz de distâncias unificada* (ou Matriz-U), proposta por Ultsch (Ultsch et al., 1993), é uma matriz composta pelas distâncias entre os neurônios vizinhos no arranjo. A Matriz-U tem dimensão $(2L-1)*(2H-1)$, considerando um arranjo retangular plano de tamanho $L*H$ (Zuchini, 2003). Com essa formação, é empregrada alguma forma de visualizar essas distâncias, seja por variações de tons de cinza ou como uma superfície de nível (Zuchini, 2003). Dessa forma, é possível interpretar os agrupamentos criados pelo SOM (Malone et al., 2005; Zuchini, 2003).

[4]São dados com probabilidades baixas de ocorrência, cujos neurônios que os representam aparentemente não possuem vizinhos próximos (Zuchini, 2003).

De acordo com a forma de visualização utilizada, é possível identificar "regiões claras" quando os vetores de pesos sinápticos dos neurônios vizinhos são próximos entre si (ou "vales", caso seja utilizada uma superfície de nível). Por outro lado, quando os vetores de pesos dos neurônios vizinhos estão mais longe, há a formação de "regiões escuras" (ou de elevações, no caso da superfície de nível). Desta maneira, um "vale" é associado a um agrupamento e as elevações que separam os vales identificam o quão esses agrupamentos são distintos entre si, de acordo com o nível das elevações (Zuchini, 2003). Há casos, contudo, em que apenas exibir as distâncias não é suficiente para detectar agrupamentos, sendo necessário algum método adicional (tal como a Matriz-U*, proposta em Ultsch (2003)).

Para o cálculo das distâncias, é necessário estabelecer alguma regra para a vizinhança. Em Zuchini (2003), são identificados dois tipos de vizinhança distintos: a *vizinhança retangular* e a *vizinhança hexagonal*. Para a vizinhança retangular, a distância nas diagonais é obtida pela média aritmética das 2 diagonais envolvidas, procedimento esse desnecessário para a vizinhança hexagonal. A escolha de qual tipo de vizinhança será utilizada depende da função de vizinhança utilizada, sendo que para a função apresentada na Seção 3.2.2, a vizinhança hexagonal é a que deve ser utilizada.

Na Figura 3.3, são apresentados dois exemplos de Matriz-U, um com vizinhança retangular e o outro com vizinhança hexagonal. Os círculos escuros representam os neurônios e os hexágonos cinzas representam as distâncias.

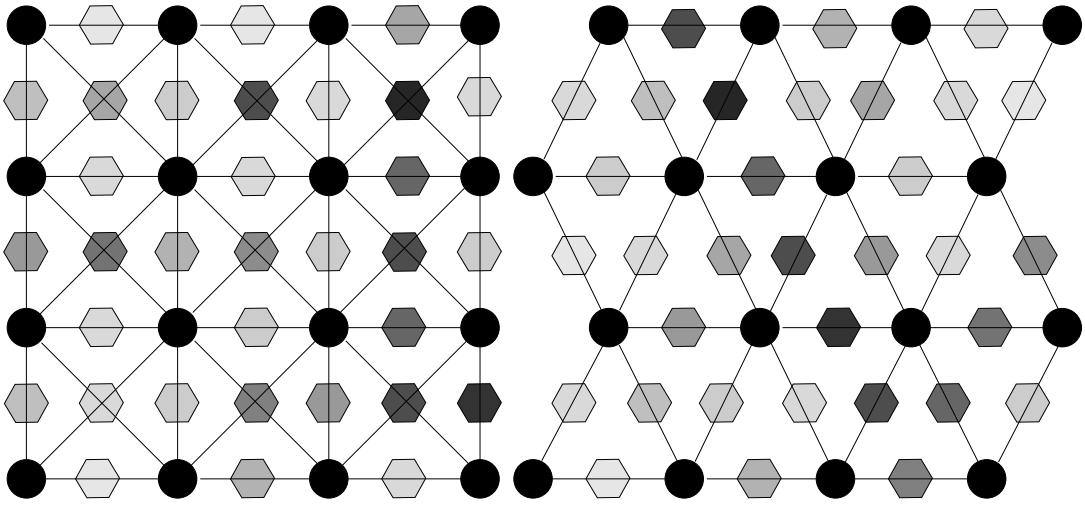

Figura 3.3: Exemplos de Matriz-U (com vizinhaça retangular e hexagonal)

Em Ultsch (2003), há mais informações a respeito da Matriz-U.

3.5.2 Mapas Contextuais

O *Mapa Contextual* (ou *Mapa Semântico*) consiste em um mapa onde cada neurônio é "rotulado" com o valor do padrão de teste que ele melhor representa (os padrões de teste

são padrões pertencentes ao espaço de entrada, mas não necessariamente ao conjunto de treinamento, que servirão para alimentar o Mapa Contextual). O rótulo é algum atributo de um elemento pertencente ao espaço de entrada que o distingue, mas não contém informações ou similaridades entre os demais elementos (por exemplo: Nome, CPF ou RG de um indivíduo). O Mapa Contextual, dessa forma, tem sua utilização em diversas aplicações e entre elas está a *mineração de dados* (Haykin, 1999).

O resultado do algoritmo para a formação de um Mapa Contextual é a geração de um mapa onde os neurônios são rotulados de tal forma que o arranjo seja particionado em regiões coerentes, onde cada grupo de neurônios representa um conjunto distinto de rótulos (Haykin, 1999). Após o mapa ter sido formado, também é possível verificar se as condições corretas foram selecionadas para o desenvolvimento do SOM (Haykin, 1999).

Para a formação do Mapa Contextual, o procedimento é o que está no Algoritmo 3 (Haykin, 1999).

Algoritmo 3 Algoritmo para a geração de um Mapa Contextual

1: Escolha um conjunto de padrões de teste, de tal forma que o número de elementos do conjunto seja menor ou igual ao número de neurônios. O nome do conjunto será chamado de T;
2: **while** Todos os neurônios não forem marcados **do**
3: Faça $U = T$;
4: **while** $U \neq \oslash$ e Todos os neurônios não forem marcados **do**
5: Escolha um padrão x do conjunto U e faça $U = U - \{x\}$;
6: Apresente x à rede e marque o neurônio ainda não marcado que produzir a melhor resposta com o valor de rótulo do padrão;
7: **end while**
8: **end while**
9: Exiba o Mapa Contextual;

Em Haykin (1999), há a menção da utilização de um tipo de padrão de teste que contém apenas o identificador de cada padrão, sem os demais dados (com seus valores iguais a 0), para preencher o Mapa Contextual. O identificador de cada padrão do espaço de entrada, para satisfazer a condição de que não é relevante como dado diferenciador entre os padrões, deve ter seus elementos divididos por uma constante predefinida. A utilização de tal artifício é, contudo, opcional.

Como exemplo de Mapa Contextual, pode ser citado o experimento realizado em Haykin (1999), no qual os dados dos animais {`pombo`, `galinha`, `pato`, `ganso`, `coruja`, `falcão`, `águia`, `raposa`, `cachorro`, `lobo`, `gato`, `tigre`, `leão`, `cavalo`, `zebra`, `vaca`} são codificados segundo a tabela que está na Figura 3.4 e um SOM é treinado com esses dados. Após o treinamento do SOM (após 2000 iterações), o Mapa Contextual resultante no experimento é o apresentado na Figura 3.5, onde há a clara divisão dos animais em três grupos principais (`caçadores`, `aves` e `espécies pacíficas`).

Animal		Pombo	Galinha	Pato	Ganso	Coruja	Falcão	Águia	Raposa	Cachorro	Lobo	Gato	Tigre	Leão	Cavalo	Zebra	Vaca
é	pequeno	1	1	1	1	1	1	1	0	0	0	1	0	0	0	0	0
	médio	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0
	grande	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
tem	2 patas	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
	4 patas	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
	pêlos	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
	cascos	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
	crina	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	0
	penas	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
gosta	caçar	0	0	0	0	1	1	1	1	0	1	1	1	1	0	0	0
	correr	0	0	0	0	0	0	0	0	1	1	0	1	1	1	1	0
	voar	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0
	nadar	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0

Figura 3.4: Tabela com os dados dos animais

cachorro	cachorro	raposa	raposa	raposa	gato	gato	gato	águia	águia
cachorro	cachorro	raposa	raposa	raposa	gato	gato	gato	águia	águia
lobo	lobo	lobo	raposa	gato	tigre	tigre	tigre	coruja	coruja
lobo	lobo	leão	leão	leão	tigre	tigre	tigre	falcão	falcão
lobo	lobo	leão	leão	leão	tigre	tigre	tigre	falcão	falcão
lobo	lobo	leão	leão	leão	coruja	pombo	falcão	pombo	pombo
cavalo	cavalo	leão	leão	leão	pombo	galinha	galinha	pombo	pombo
cavalo	cavalo	zebra	vaca	vaca	vaca	galinha	galinha	pombo	pombo
zebra	zebra	zebra	vaca	vaca	vaca	galinha	galinha	pato	ganso
zebra	zebra	zebra	vaca	vaca	vaca	pato	pato	pato	ganso

Figura 3.5: Mapa Contextual resultante

3.6 Considerações sobre os parâmetros

Segundo Zuchini (2003), os parâmetros que regulam o SOM podem ser agrupados em dois conjuntos: *parâmetros que definem a estrutura do mapa* (dimensão, vizinhança e formato do arranjo, raio e tipo de função de vizinhança $h_{j,i}$) e *parâmetros que controlam o treinamento* (taxa de aprendizado $\eta(n)$, decrescimento do raio de vizinhança $\sigma(n)$ e número de épocas de treinamento). Além disso, podem ser considerados como parâmetros adicionais a *forma de inicialização dos neurônios* (aleatória ou linear), a *divisão do treinamento em duas fases* (adaptação e convergência) e a *normalização dos dados* (Zuchini, 2003).

Devido ao algoritmo SOM ser matematicamente difícil de analisar suas propriedades de forma geral (Haykin, 1999), não existem critérios mensuráveis para os valores dos parâmetros a fim de se obter mapas bem ajustados. Contudo, quando o SOM tem por finalidade a mineração de dados, são propostos os seguintes parâmetros (considerando-se

um arranjo bidimensional) (Zuchini, 2003):

- O número de neurônios do mapa, quando a quantidade de dados de entrada é "pequena" (menor do que 1000), pode ser igual ao de dados. Outra proposta é fazer $l = 5\sqrt{n}$, onde l é o *número de neurônios* e n é a *quantidade de dados de entrada*;
- Utilizar no arranjo a forma de vizinhança hexagonal, propiciando melhor visualização dos agrupamentos com a Matriz-U;
- Utilizar a função de vizinhança $h_{j,i}$ baseada em uma gaussiana, tal como a Equação 3.8;
- Utilizar duas fases de treinamento, tal como explicitado na Seção 3.2.3;
- Ajustar os valores η_0, τ_1 e τ_2 de tal forma como estão nas Equações 3.11 e 3.12;
- Ajustar o raio σ_0 da função de vizinhança com o valor $\frac{md}{4}$ (onde md é maior valor da dimensão plana do arranjo, largura ou altura) e utilizar a função de decaimento exponencial para modificar σ com o tempo de tal forma como mostrada na Seção 3.2.2;
- Ajustar o número de iterações para pelo menos 1000 para a fase de ordenação e, para a fase de convergência, pelo menos 500 vezes o número de neurônios (Haykin, 1999).

Essas são algumas considerações sobre como ajustar os parâmetros, mas é recomendado realizar diversos testes com várias configurações do SOM para verificar o seu comportamento e decidir qual configuração utilizar (Zuchini, 2003).

Capítulo 4

Segmentação de perfis

4.1 Introdução

Em muitos domínios, como no comércio, na medicina, na indústria, entre outros, o conhecimento a respeito dos perfis com os quais esses domínios atuam é de fundamental importância, no qual o conhecimento adquirido servirá para uma maior compreensão sobre os mesmos (Malone et al., 2005; Lingras et al., 2005).

Os perfis, no contexto abordado neste trabalho, representam objetos (pessoas, documentos, recursos etc) cujas características são consideradas importantes para o domínio abordado. A análise desses perfis, com o intuito de ter um conhecimento maior sobre o domínio, em muitos casos não é trivial, na qual muitas características relacionadas ao domínio podem não ser identificadas sem a utilização de alguma ferramenta que auxilie em tal tarefa. A área da *Mineração de Dados e Descoberta do Conhecimento*, por sua vez, com a utilização de técnicas de segmentação e classificação de perfis, cujos algoritmos realizam um processo de aprendizado acerca do domínio, tem como um de seus objetivos abordar esse problema.

Em domínios onde há um conhecimento prévio sobre o mesmo, podem ser utilizadas técnicas que empregam aprendizagem de máquina supervisionada para classificar os grupos de perfis (Lingras et al., 2005). Por outro lado, em domínios onde há pouco ou nenhum conhecimento, é necessário empregar *técnicas de agrupamento de perfis de forma não-supervisionada*. Dessa forma, a segmentação de perfis e, conseqüentemente, o maior conhecimento sobre o domínio abordado, é realizada através da análise dos agrupamentos encontrados.

Em Lingras et al. (2005), são identificadas três abordagens diferentes para o problema do agrupamento de perfis de forma não-supervisionada: *evolucionário*, *estatístico* e *neural*. Segundo Lingras et al. (2005), um dos métodos estatísticos mais populares é o *K-means*. Por sua vez, como método neural, a *Rede Neural SOM de Kohonen* (que é objeto de estudo deste trabalho) também é bastante utilizada.

Segundo Malone et al. (2005), o algoritmo SOM é utilizado como técnica de agrupamento por promover um mapeamento do espaço de entrada do domínio para o espaço discreto de saída do arranjo de neurônios, preservando a topologia do espaço de entrada. Dessa forma, é possível interpretar o domínio multidimensional em um mapa geralmente bidimensional. Além disso, o algoritmo contribui para que seja possível a formação de regras (Malone et al., 2005), embora o mapa gerado pelo SOM não garanta uma fácil interpretação do mesmo.

4.2 Segmentando perfis utilizando o algoritmo SOM

Com a utilização do algoritmo SOM, a tarefa de segmentar perfis resume-se nos seguintes passos:

1. **Escolha dos perfis de treinamento:** O algoritmo SOM não necessita que todos os perfis do domínio sejam apresentados durante o seu treinamento. No entanto, a escolha dos perfis com que será treinado o SOM deve ser de tal forma que o conjunto de treinamento *represente bem o domínio*, permitindo que perfis posteriores (e que não foram utilizados no treinamento) possam ser representados no mapa gerado pelo SOM;

2. **Codificação dos atributos:** Por ser o SOM um algoritmo neural, os perfis que servirão para o treinamento devem ser codificados em vetores numéricos (tal como explicitado na Seção 2.1). A sua codificação deve ser feita de tal forma que, para cada atributo do perfil, os seus diferentes valores possíveis (ou faixas de valores) sejam codificados com valores numéricos diferentes. Os vetores já codificados, como característica geral, possuem a mesma dimensão dos perfis que eles representam;

3. **Treinamento do SOM:** Com os vetores de treinamento em mãos, o treinamento do SOM é executado. Para o treinamento, devem ser escolhidos os parâmetros apropriados para que seja formado um mapa que represente bem o domínio. Vale ressaltar que os dados de entrada e os vetores de pesos sinápticos devem sempre ser normalizados;

4. **Análise do mapa gerado pelo SOM:** Após o treinamento, é feita a interpretação do mapa gerado pelo SOM. Para a interpretação, os possíveis agrupamentos devem ser identificados através de métodos de visualização. Como o mapa gerado pelo SOM não possui semântica direta, a total interpretação do mapa não é garantida, sendo necessário algum método adicional para isso (tal como o proposto em Malone et al. (2005)).

4.3 Segmentador

O componente Segmentador, escrito na linguagem de programação `C++` (Stroustrup, 2000), corresponde à implementação do algoritmo SOM, que fará a tarefa de gerar uma mapa onde os diferentes perfis estarão agrupados, e a visualização desse mapa se dá através de um Mapa Contextual, também implementado por esse componente. O componente, contudo, não faz a tarefa de codificar os dados contidos em um banco de dados, devendo essa tarefa ser feita por algum outro componente. O diagrama de classes do Segmentador está na Figura 4.1 e o código-fonte da implementação do componente está no Apêndice A.

Com relação às bibliotecas utilizadas, o foram somente as bibliotecas-padrão do `C++`, o que garante que o componente possa ser compilado em qualquer compilador que implemente o `C++` padrão (Stroustrup, 2000).

As classes integrantes do componente são:

- **Calculos:** Efetua cálculos sobre vetores, necessários para o componente. Os métodos dessa classe são:
  - `calculaNorma:` Calcula a norma de um vetor dado;
  - `normalizaVetor:` Deixa o vetor com norma igual a 1, usando normalização euclidiana;
  - `calculaDistancia:` Dados dois vetores de mesma dimensão, calcula a distância euclidiana entre eles.

  Esses métodos são `static`, o que significa que não é necessário instanciar um objeto dessa classe para usar seus métodos.

- **Dado:** Classe que representa os dados já codificados que servirão de entrada para o SOM. Para essa classe, há um vetor de `doubles`, que guardará os valores codificados do padrão.

- **Neuronio:** É a base do algorimo SOM. Para essa classe, pode ser citado o vetor de pesos sinápticos, representado por um vetor de `doubles`, e os seguintes métodos mais interessantes:
  - `calculaDistanciaEspacial:` Calcula a distância espacial do neurônio em relação a outro neurônio dado;
  - `calculaVizinhanca:` Faz o cálculo do valor de vizinhança, dados o neurônio vencedor e o valor da largura de vizinhança σ, segundo a Equação 3.8;

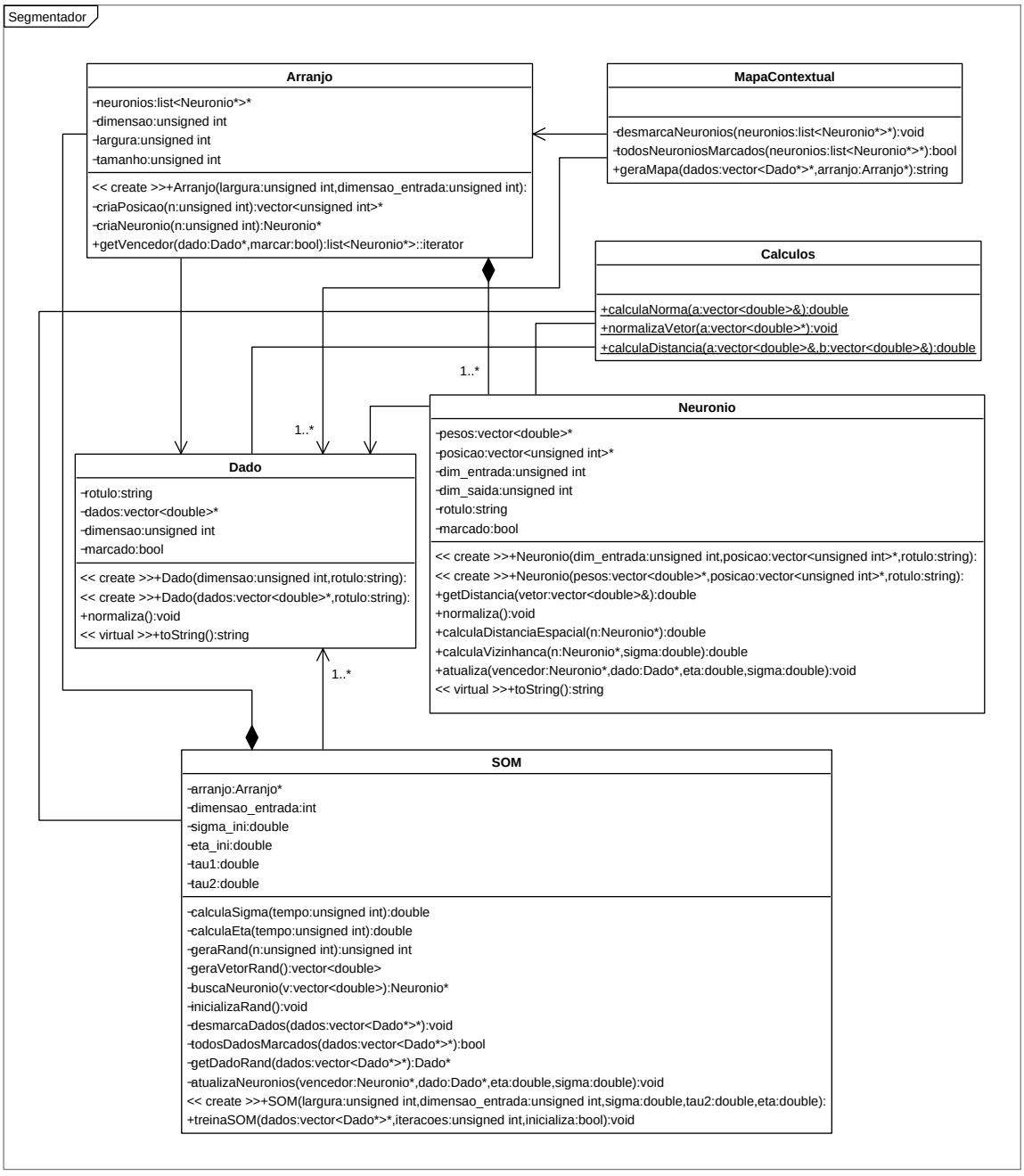

Figura 4.1: Diagrama de classes do Segmentador

  - `atualiza`: Dados o neurônio vencedor, o dado apresentado e os valores da taxa de aprendizado e da largura de vizinhança, atualiza os seus pesos sinápticos segundo a Equação 3.9 e depois normaliza o seu vetor de pesos sinápticos.

- **Arranjo:** Classe que representa o arranjo de neurônios. Além de armazenar os neurônios, essa classe *determina a localização espacial de cada um deles* e também *determina qual neurônio é o vencedor* para um dado de entrada amostrado, através do método `getVencedor`. A restrição dessa classe é que ela produz apenas arranjos bidimensionais quadrados (com largura e altura iguais).

- **SOM:** Classe que implementa o algoritmo SOM. O algoritmo implementado é o Algoritmo 1, da mesma forma como descrito na Seção 3.2. Para treinar o SOM, o método `treinaSOM` é criado com tal propósito.

- **MapaContextual:** Implementa o Mapa Contextual a partir de um arranjo dado, tal como descrito na Seção 3.5.2.

4.4 Experimentos com o componente Segmentador

Após o componente Segmentador ter sido implementado, foram feitos alguns experimentos para verificar se o algoritmo SOM foi implementado corretamente. Além disso, também foram testadas configurações diferentes dos parâmetros, tais como: número de iterações, largura inicial da função de vizinhança, tamanho do SOM etc. Para essa fase de "testes", um dos primeiros casos experimentados foi o dos animais, realizado por Haykin (1999) e que está explicitado na Seção 3.5.2, obtendo resultado semelhante.

Por conta da dificuldade de se obter dados de ambientes reais, todos os experimentos foram realizados com dados fictícios, fazendo com que os resultados obtidos não se apliquem ao mundo real. Entretanto, esses experimentos evidenciaram que o Segmentador é capaz de agrupar os perfis diferentes, podendo ser utilizado em domínios reais.

Entre os experimentos realizados, estão os descritos nas próximas subseções.

4.4.1 Clientes de um supermercado

Neste experimento, foram considerados os perfis de 20 clientes de um supermercado fictício, onde suas características foram codificadas da seguinte forma:

- **Idade:**
  - 0: $<$ 13 anos;
  - 1: 13-17 anos;
  - 2: 18-24 anos;

  - 3: 25-39 anos;
  - 4: 40-59 anos;
  - 5: $>$ 60 anos;

- **Bairro onde mora:**
  - 0: Farol;
  - 1: Ponta Verde;
  - 2: Pajuçara;
  - 3: Tabuleiro;
  - 4: Jacintinho;
  - 5: Gruta;
  - 6: Serraria;
  - 7: Feitosa;
  - 8: Poço;
  - 9: Barro Duro;

- **Instrução:**
  - 0: Analfabeto;
  - 1: Fundamental incompleto;
  - 2: Fundamental completo;
  - 3: Ensino médio;
  - 4: Ensino superior;
  - 5: Pós-graduação;

- **Renda familiar:**
  - 0: $<$ 1 salário mínimo;
  - 1: 1-2 salários mínimos;
  - 2: 3-5 salários mínimos;
  - 3: 6-10 salários mínimos;
  - 4: $>$ 10 salários mínimos;

- **Sexo:** Masculino (0) ou Feminino (1);
- **Alimentação:** Entre 0 e 1;

- **Bebida:** Entre 0 e 1;
- **Vestuário:** Entre 0 e 1;
- **Eletro/Eletrônicos**: Entre 0 e 1;
- **Livros:** Entre 0 e 1;
- **Produtos diversos:** Entre 0 e 1.

Os itens `Alimentação`, `Bebida`, `Vestuário`, `Eletro/Eletrônicos`, `Livros` e `Produtos diversos` representam a proporção do que a pessoa gastou no mercado nos últimos 6 meses, sendo a soma desses seis atributos sempre igual a 1.

Na Tabela 4.1, há os dados dos clientes já codificados. Esses dados, por sua vez, servirão para alimentar o SOM.

Tabela 4.1: Perfis de clientes do mercado

	Idade	Bairro	Instr.	Renda	S.	Alim.	Beb.	Ves.	E/E	Livros	Div.
Aline	1	0	3	4	1	0,3	0	0,3	0,1	0,25	0,05
Antônio	1	0	3	4	0	0,3	0	0,2	0,3	0,1	0,1
Camila	1	9	3	2	1	0,4	0	0,2	0,1	0,15	0,15
Carla	0	0	1	3	1	0,2	0	0,3	0,1	0,3	0,1
Carlos	3	0	4	3	0	0,4	0,05	0,05	0,1	0,2	0,2
Cauã	0	1	1	4	0	0,35	0	0,3	0,2	0,05	0,1
Creuza	3	4	0	0	1	0,8	0,1	0	0	0	0,1
Fred	3	1	5	4	0	0,4	0,05	0,15	0,05	0,25	0,1
Isabel	4	3	2	2	1	0,4	0	0,2	0,2	0,1	0,1
Joana	1	1	3	4	1	0,2	0	0,4	0,15	0,05	0,2
Joaquim	5	6	3	3	0	0,5	0,1	0,05	0,1	0,1	0,15
José	3	3	1	1	0	0,75	0,1	0,05	0,05	0	0,05
Josefa	5	8	2	1	1	0,7	0,05	0,05	0	0,05	0,15
Júnior	3	2	3	3	0	0,4	0,1	0,1	0,1	0,05	0,25
Marcos	3	2	3	2	0	0,45	0,05	0,1	0,2	0,1	0,1
Maria	3	3	1	0	1	0,7	0,1	0,1	0,05	0	0,05
Teresa	4	6	4	3	1	0,3	0	0,3	0,2	0,05	0,15
Thiago	1	7	3	2	0	0,3	0	0,1	0,25	0,15	0,2
Vanessa	2	7	4	2	1	0,4	0	0,3	0,1	0,1	0,1
Vinícius	3	1	4	3	0	0,3	0,1	0,1	0,1	0,3	0,1

Para o treinamento do SOM, os seguintes parâmetros foram adotados:

- **Número de iterações:** 20000;
- **Formato do arranjo:** quadrado, de tamanho 10x10;
- **Largura inicial do parâmetro de vizinhança σ_0:** $2,5$;

- **Taxa inicial de aprendizado η_0:** $0,1$;

Após o treinamento, o Mapa Contextual resultante é o que está na Tabela 4.2.

Tabela 4.2: Mapa Contextual do experimento do supermercado

Carlos	Carlos	Fred	Fred	Antônio	Antônio	Aline	Aline	Carla	Carla
Carlos	Vinícius	Vinícius	Fred	Antônio	Antônio	Aline	Aline	Carla	Carla
Marcos	Vinícius	Carlos	Júnior	Fred	Antônio	Joana	Aline	Carla	Cauã
Marcos	Marcos	Júnior	Júnior	Vinícius	Joana	Joana	Joana	Cauã	Cauã
Isabel	Isabel	Marcos	Júnior	Carlos	Fred	Joana	Vinícius	Cauã	Cauã
Isabel	Isabel	Isabel	Joaquim	Marcos	Teresa	Júnior	Teresa	Thiago	Thiago
Maria	Maria	Joaquim	Joaquim	Maria	Teresa	Teresa	Vanessa	Thiago	Thiago
Maria	Maria	José	Joaquim	Joaquim	Teresa	Vanessa	Vanessa	Vanessa	Thiago
Creuza	Creuza	José	José	Josefa	Josefa	Josefa	Vanessa	Camila	Camila
Creuza	Creuza	José	José	Josefa	Josefa	Creuza	Camila	Camila	Camila

Resultados

Após a análise do Mapa Contextual resultante, é possivel identificar algumas características dos clientes, podendo o leitor verificar se há mais características:

- **Clientes jovens, com boas condições financeiras:** São clientes com idade menor que 18 anos e que têm renda familiar a partir de 6 salários mínimos. Encontram-se nesse grupo os clientes `Aline`, `Antônio`, `Carla`, `Cauã` e `Joana`. Esses clientes moram nos bairros Farol e Ponta Verde e seus gastos estão concentrados entre vestuário, livros ou eletro/eletrônicos;

- **Clientes com baixo nível sócio-econômico:** Encontram-se nessa situação os clientes `Creuza`, `Isabel`, `José`, `Josefa` e `Maria`. Quase todos têm renda familiar abaixo de 3 salários mínimos e seus gastos concentram-se principalmemente em alimentação. Além disso, nenhum deles têm o ensino médio completo e quase todos (com exceção de Josefa) moram nos bairros Jacintinho e Tabuleiro.

- **Clientes integrantes da classe média, do sexo masculino:** Os integrantes desse grupo (`Carlos`, `Fred`, `Júnior`, `Marcos` e `Vinícius`) têm idade a partir dos 25 anos, com renda familiar a partir de 6 salários mínimos e moram nos bairros Farol, Ponta Verde e Pajuçara. A maior parte deles é de graduados e seus gastos são mais distribuídos, com um interesse maior pelos livros (se comparado com os outros grupos).

- **Clientes mais velhos:** os clientes que se encontram nesse grupo (`Joaquim` e `Teresa`) têm idade maior que 40 anos, moram no bairro Serraria, têm pelo menos o ensino médio e renda familiar entre 6 e 10 salários mínimos. O gasto com

produtos diversos é maior devido provavelmente a um gasto maior com remédios, em razão da idade mais avançada de seus integrantes.

- **Clientes jovens:** Os clientes `Camila`, `Thiago` e `Vanessa` são jovens (com idade entre 13 e 24 anos), com renda familiar entre 3 e 5 salários mínimos, pelos menos o nível médio, moram nos bairros Feitosa e Barro Duro e têm maior interesse por roupas, livros ou eletro/eletrônicos.

4.4.2 Análise epidemiológica

Para este experimento, foi considerada uma pesquisa fictícia em alguns bairros de Maceió, cujo objetivo é saber quais as localidades onde há maior incidência das doenças *dengue*, *dengue hemorrágica* e *malária*. Para o treinamento do SOM, a entrada será de 20 dessas pessoas pesquisadas, cuja codificação de seus dados se dá da seguinte forma:

- **Sexo:** Masculino (0) ou Feminino (1);
- **Idade:** Codificado da mesma forma que o experimento da Seção 4.4.1;
- **Bairro:** idem;
- **Dengue:** 0 ou 1;
- **Dengue hemorrágica:** 0 ou 1;
- **Malária:** 0 ou 1.

Os atributos `Dengue`, `Dengue hemorrágica` e `Malária` são valores binários, que indicam se a pessoa teve (com valor 1) ou não teve (valor 0) tal doença no último ano.

Os dados dos habitantes, que com eles será treinado o SOM, é o que está na Tabela 4.3.

Para o treinamento do SOM, os seguintes parâmetros foram adotados:

- **Número de iterações:** 20000;
- **Formato do arranjo:** quadrado, de tamanho 10x10;
- **Largura inicial do parâmetro de vizinhança σ_0:** $2,5$;
- **Taxa inicial de aprendizado η_0:** $0,1$;

Após o treinamento, o Mapa Contextual resultante é o que está na Tabela 4.4.

Tabela 4.3: Perfis dos habitantes

	Sexo	Idade	Bairro	Dengue	Dengue Hemorr.	Malária
Adriana	1	2	5	1	0	0
Alfonso	0	5	3	1	1	0
Aline	1	3	3	1	1	0
Augusto	0	4	6	0	0	0
Bruno	0	0	7	0	0	0
Carlos	0	3	6	0	0	0
Caroline	1	2	4	1	0	0
Francisca	1	3	7	0	0	0
Henrique	0	2	4	0	0	0
José	0	3	4	1	0	0
Juliana	1	1	7	0	0	0
Leila	1	0	5	0	0	1
Maria	1	4	5	0	0	1
Mateus	0	1	5	0	0	1
Miguel	0	4	5	1	0	0
Paulo	0	2	6	0	0	0
Pedro	0	1	7	0	0	0
Sílvia	1	1	4	1	0	0
Tiago	0	0	3	1	0	0
Verônica	1	3	5	0	0	1

Tabela 4.4: Mapa Contextual do experimento dos habitantes

Paulo	Paulo	Leila	Leila	Bruno	Bruno	Tiago	Tiago	Tiago	Tiago
Paulo	Paulo	Leila	Paulo	Bruno	Bruno	Bruno	Tiago	Leila	Sílvia
Juliana	Juliana	Juliana	Pedro	Pedro	Mateus	Leila	Sílvia	Sílvia	Sílvia
Francisca	Juliana	Paulo	Pedro	Pedro	Mateus	Mateus	Juliana	Caroline	Sílvia
Francisca	Francisca	Henrique	Henrique	Carlos	Mateus	Mateus	Caroline	Caroline	Caroline
Francisca	Francisca	Henrique	Henrique	Carlos	Henrique	Verônica	Caroline	Adriana	Adriana
Aline	Aline	Carlos	Carlos	Carlos	Augusto	Augusto	Alfonso	Adriana	Adriana
Aline	Aline	José	José	Augusto	Augusto	Augusto	Verônica	Aline	Adriana
Alfonso	Alfonso	Miguel	Miguel	José	Miguel	Maria	Maria	Maria	Verônica
Alfonso	Alfonso	Miguel	Miguel	José	José	Maria	Maria	Verônica	Verônica

Resultados

Neste experimento, é possível identificar os seguintes agrupamentos após a análise do Mapa Contextual (provavelmente existem mais formas de se agrupar as pessoas, dependendo de como se interpreta o mapa):

- **Pessoas que não contraíram doença alguma:** As pessoas `Augusto`, `Bruno`, `Carlos`, `Francisca`, `Henrique`, `Juliana`, `Paulo` e `Pedro` não tiveram as doenças pesquisadas no período considerado. Eles moram principalmente nos bairros Serraria e Feitosa e a maior parte tem idade menor que 25 anos (exceto `Augusto`, `Carlos` e `Francisca`, que provavelmente formam um grupo de pessoas que não contraíram as doenças e têm maior idade).

- **Pessoas que contraíram dengue hemorrágica:** As pessoas `Alfonso` e `Aline` contraíram essa doença. Ambos moram no bairro Tabuleiro.

- **Pessoas que contraíram apenas dengue, homens:** Nesse grupo, estão `José` e `Miguel`. Eles contraíram dengue, têm idade entre 25 e 59 anos e moram nos bairros Jacintinho e Gruta.

- **Pessoas que contraíram apenas dengue, mulheres:** `Adriana` e `Caroline` fazem parte desse grupo. Elas têm idade entre 18 e 24 anos e moram nos bairros Jacintinho e Gruta.

- **Pessoas que contraíram apenas dengue, jovens:** `Sílvia` e `Tiago`, integrantes deste grupo, são jovens (com idade menor que 18 anos) e moram nos bairros Tabuleiro e Jacintinho.

- **Pessoas que contraíram malária, jovens:** `Leila` e `Mateus` estão nesse grupo. Ambom moram no bairro Gruta e têm idade menor que 13 anos.

- **Pessoas que contraíram malária, adultos:** `Maria` e `Verônica` fazem parte desse grupo. Elas têm idade entre 25 e 59 anos e moram no bairro Tabuleiro.

Com essa análise, chega-se à conclusão de que é bem provável ser necessário reforçar as medidas profiláticas nos bairros Jacintinho, Gruta e Tabuleiro, a fim de se evitar que essas doenças virem epidemias.

Capítulo 5

Conclusão

Neste trabalho, foi realizado um estudo sobre Redes Neurais, com foco nas redes SOM. Como estudo de caso, foi desenvolvido um sistema de *software* segmentador de perfis, que implementa o algoritmo SOM e cuja segmentação é efetuada através da análise do mapa produzido pelo mesmo.

O objetivo do presente trabalho é, portanto, fornecer um embasamento teórico para o desenvolvimento de um sistema que implemente o algoritmo SOM. Por ser o algoritmo SOM apropriado para ser aplicado em muitos domínios que necessitem de aprendizagem de máquina não-supervisionada, foi escolhido um específico: *a segmentação de perfis*.

Nos experimentos realizados com o sistema desenvolvido, constatou-se a sua capacidade de agrupar os diferentes perfis, propiciando ao usuário do sistema analisar o Mapa Contextual gerado e segmentar os diferentes grupos de perfis. Em alguns experimentos, no entanto, apenas o Mapa Contextual não foi suficiente para uma boa compreensão dos agrupamentos do domínio estudado, evidenciando a necessidade de algum método adicional de visualização, tal como a Matriz-U. Contudo, mesmo com a adoção de tal método, a total interpretação do mapa gerado pelo SOM não é garantida, pois a sua análise é subjetiva e por vezes de difícil interpretação (Zuchini, 2003). Faz-se necessário, portanto, a utilização de algum método adicional para avaliar o mapa, tal como a *extração de regras a partir do SOM*, proposta em Malone et al. (2005).

Outro ponto a considerar é com relação ao tempo de execução dos experimentos, que levou entre 10 e 20 minutos na máquina onde eles foram realizados, no qual o gasto com memória foi pequeno, mas com alta utilização do processador. Desta forma, a utilização de outras estruturas de dados e algoritmos a fim de otimizar a sua execução deve ser considerada. Além disso, também pode vir a ser considerada a *paralelização do algoritmo*, com a utilização dos algoritmos e estruturas de dados apropriadas.

Como trabalho futuro, além dos supramencionados, poderá ser considerada a utilização de algum método derivado do SOM tradicional, tais como muitos encontrados na literatura.

Caso o leitor queira aprofundar-se no assunto, o presente trabalho possui algumas boas referências, o que poderá ajudar em seu trabalho.

Apêndice A

Código-fonte do Segmentador

A.1 Classe Calculos

Cabeçalho

Código A.1: Calculos.h

```
1  /* Classe para efetuar cálculos diversos
2   * entre vetores de doubles */
3
4  #ifndef CALCULOS_H_
5  #define CALCULOS_H_
6
7  #include <vector>
8  #include <cmath>
9
10 using namespace std;
11
12 // Soma de vetores
13 vector<double> operator+ (vector<double> &a, vector<double> &b);
14 // Subtração de vetores
15 vector<double> operator- (vector<double> &a, vector<double> &b);
16 // Divisão por um Real
17 vector<double> operator/ (vector<double> &a, double b);
18 // Multiplicação por um Real
19 vector<double> operator* (vector<double> &a, double b);
20 // Multiplicação por um Real
21 vector<double> operator* (double a, vector<double> &b);
22 // Produto interno
```

```
23 double operator* (vector<double> &a, vector<double> &b);
24
25 class Calculos {
26
27 public:
28   Calculos();
29   virtual ~Calculos();
30
31   // Módulo do vetor
32   static double calculaNorma(const vector<double> &a);
33   // Normaliza o vetor (deixando-o com norma igual a 1)
34   static void normalizaVetor(vector<double> *a);
35   // Distância euclidiana
36   static double calculaDistancia(const vector<double> &a,
37     const vector<double> &b);
38 };
39
40 #endif /*CALCULOS_H_*/
```

Implementação

Código A.2: Calculos.cpp

```
1  #include "Calculos.h"
2
3  Calculos::Calculos() {}
4
5  Calculos::~Calculos() {}
6
7  // Módulo do vetor
8  double Calculos::calculaNorma(const vector<double> &a) {
9    double distancia = 0.0;
10
11   for(unsigned int i = 0 ; i < a.size(); i++)
12     distancia += pow(a.at(i), 2.0);
13
14   return sqrt(distancia);
15 }
16
```

```
17 void Calculos::normalizaVetor(vector<double> *a) {
18   // Calcula a norma do vetor
19   double norma = Calculos::calculaNorma(*a);
20
21   // Normaliza o vetor
22   for(unsigned int i = 0; i < a->size(); i++)
23     a->at(i) /= norma;
24 }
25
26 // Distância euclidiana
27 double Calculos::calculaDistancia(const vector<double> &a,
28   const vector<double> &b) {
29   double dif, distancia = 0.0;
30
31   for(unsigned int i = 0 ; i < a.size(); i++) {
32     dif = a.at(i) - b.at(i);
33     distancia += pow(dif, 2.0);
34   }
35
36   return sqrt(distancia);
37 }
38
39 // ************************************************************
40 // Sobrecarga de operadores
41 // ************************************************************
42
43 // Soma de dois vetores
44 vector<double> operator+ (vector<double> &a, vector<double> &b){
45   vector<double> vetorSoma(a.size());
46
47   for(unsigned int i = 0; i < a.size(); i++) {
48     vetorSoma.at(i) = a.at(i) + b.at(i);
49   }
50
51   return vetorSoma;
52 }
53
54 // Subtração de dois vetores
55 vector<double> operator- (vector<double> &a, vector<double> &b){
```

```
56    vector<double> vetorSub(a.size());
57
58    for(unsigned int i = 0; i < a.size(); i++) {
59      vetorSub.at(i) = a.at(i) - b.at(i);
60    }
61
62    return vetorSub;
63  }
64
65  // Divisão por um Real
66  vector<double> operator/ (vector<double> &a, double b) {
67    vector<double> vetorDiv(a.size());
68
69    for(unsigned int i = 0; i < a.size(); i++) {
70      vetorDiv.at(i) = a.at(i)/b;
71    }
72
73    return vetorDiv;
74  }
75
76  // Multiplicação por um Real
77  vector<double> operator* (vector<double> &a, double b) {
78    vector<double> vetorMult(a.size());
79
80    for(unsigned int i = 0; i < a.size(); i++) {
81      vetorMult.at(i) = a.at(i)*b;
82    }
83
84    return vetorMult;
85  }
86
87  // Multiplicação por um Real
88  vector<double> operator* (double a, vector<double> &b) {
89    return b*a;
90  }
91
92  // Produto interno
93  double operator* (vector<double> &a, vector<double> &b) {
94    double prodInter = 0.0;
```

```
95
96    for(unsigned int i = 0; i < a.size(); i++) {
97      prodInter += a.at(i)*b.at(i);
98    }
99
100   return prodInter;
101 }
```

A.2 Classe Dado

Cabeçalho

Código A.3: Dado.h

```
1  // Classe que representa os dados que alimentarão o SOM
2
3  #ifndef DADO_H_
4  #define DADO_H_
5
6  #include <vector>
7  #include <cmath>
8  #include <string>
9  #include <sstream>
10
11 #include "Calculos.h"
12
13 using namespace std;
14
15 class Dado {
16   /* Rótulo que marca o dado (serve para a formação de um
17    * Mapa Contextual) */
18   string rotulo;
19
20   vector<double>* dados; // Vetor que representa o dado
21   unsigned int dimensao; // Dimensão do dado
22   // Serve para os algoritmos SOM e do Mapa Contextual
23   bool marcado;
24
25 public:
```

```
26    Dado(unsigned int dimensao, string rotulo = "");
27    Dado(vector<double>* dados , string rotulo = "");
28    virtual ~Dado();
29
30    // Gets e sets
31    string getRotulo();
32    vector<double>* getDados();
33    bool getMarcado();
34
35    void setRotulo(string rotulo);
36    void setDados(vector<double>* dados);
37    void setMarcado(bool marcado);
38
39    void normaliza(); // Normaliza o vetor de dados
40
41    virtual string toString();
42 };
43
44 #endif /*DADO_H_*/
```

Implementação

Código A.4: Dado.cpp

```
1  #include "Dado.h"
2
3  Dado::Dado(unsigned int dimensao, string rotulo) {
4     this->dimensao = dimensao;
5     this->rotulo = rotulo;
6     this->marcado = false;
7     this->dados = new vector<double>(this->dimensao);
8  }
9
10 Dado::Dado(vector<double>* dados , string rotulo) {
11    this->dados = dados;
12    this->dimensao = this->dados->size();
13    this->rotulo = rotulo;
14    this->marcado = false;
15 }
```

```
16
17 Dado::~Dado() {
18   delete this->dados;
19 }
20
21 string Dado::getRotulo() {
22   return this->rotulo;
23 }
24
25 vector<double>* Dado::getDados() {
26   return this->dados;
27 }
28
29 bool Dado::getMarcado() {
30   return this->marcado;
31 }
32
33 void Dado::setRotulo(string rotulo) {
34   this->rotulo = rotulo;
35 }
36
37 void Dado::setDados(vector<double>* dados) {
38   this->dados = dados;
39   this->dimensao = this->dados->size();
40 }
41
42 void Dado::setMarcado(bool marcado) {
43   this->marcado = marcado;
44 }
45
46 void Dado::normaliza() {
47   Calculos::normalizaVetor(this->dados);
48 }
49
50 string Dado::toString() {
51   ostringstream str("");
52
53   str << "Dado(" << this->rotulo << "): ";
54   for(unsigned int i = 0; i < this->dimensao; i++) {
```

```
55      str << this->dados->at(i) << " ";
56    }
57
58    return str.str();
59  }
```

A.3 Classe Neuronio

Cabeçalho

Código A.5: Neuronio.h

```
1  // Classe Neurônio, que é a base do algoritmo SOM de Kohonen
2
3  #ifndef NEURONIO_H_
4  #define NEURONIO_H_
5
6  #include <vector>
7  #include <cmath>
8  #include <string>
9  #include <sstream>
10
11 #include "Calculos.h"
12 #include "Dado.h"
13
14 using namespace std;
15
16 class Neuronio {
17
18   // Vetor de pesos sinápticos
19   vector<double>* pesos;
20   // Vetor da posição do neurônio no arranjo
21   vector<unsigned int>* posicao;
22
23   unsigned int dim_entrada; // Dimensão de entrada
24   unsigned int dim_saida; // Dimensão de saída
25
26   string rotulo; // Servem para o Mapa Contextual
27   bool marcado;
```

```
28
29 public:
30
31   Neuronio(unsigned int dim_entrada,
32     vector<unsigned int>* posicao, string rotulo = "");
33   Neuronio(vector<double> *pesos,
34     vector<unsigned int>* posicao, string rotulo = "");
35   virtual ~Neuronio();
36
37   // Gets e sets
38   unsigned int getDim(); // Dimensão de entrada
39   vector<double>* getPesos();
40   // Posição do neurônio no arranjo
41   vector<unsigned int>* getPosicao();
42   string getRotulo();
43   bool getMarcado();
44
45   void setPesos(vector<double>* pesos);
46   void setRotulo(string rotulo);
47   void setMarcado(bool marcado);
48
49   // Distância euclidiana
50   double getDistancia(const vector<double> &vetor);
51   void normaliza(); // Normaliza seus pesos sinápticos
52
53   // Retorna a distância espacial entre dois neurônios
54   double calculaDistanciaEspacial(Neuronio* n);
55
56   /* Calcula a função de vizinhança do neurônio
57    *dado um vencedor */
58   double calculaVizinhanca(Neuronio* n, double sigma);
59
60   /* Atualiza o neurônio, segundo o algoritmo da Seção 3.2.3
61    * E depois normaliza o vetor de pesos */
62   void atualiza(Neuronio* vencedor, Dado* dado, double eta,
63     double sigma);
64
65   virtual string toString();
66 };
```

```
67
68 #endif /*NEURONIO_H_*/
```

Implementação

Código A.6: Neuronio.cpp

```
1  #include "Neuronio.h"
2
3  Neuronio::Neuronio(unsigned int dim_entrada,
4    vector<unsigned int>* posicao, string rotulo) {
5    this->posicao = posicao;
6    this->pesos = new vector<double>(dim_entrada);
7
8    this->dim_entrada = dim_entrada;
9    this->dim_saida = this->posicao->size();
10
11   this->rotulo = rotulo;
12   this->marcado = false;
13 }
14
15 Neuronio::Neuronio(vector<double> *pesos,
16   vector<unsigned int>* posicao, string rotulo) {
17   this->pesos = pesos;
18   this->posicao = posicao;
19
20   this->dim_entrada = this->pesos->size();
21   this->dim_saida = this->posicao->size();
22
23   this->rotulo = rotulo;
24   this->marcado = false;
25 }
26
27 Neuronio::~Neuronio() {
28   delete this->pesos;
29   delete this->posicao;
30 }
31
32 unsigned int Neuronio::getDim() {
```

```
33    return this->dim_entrada;
34  }
35
36  vector<double>* Neuronio::getPesos() {
37    return this->pesos;
38  }
39
40  vector<unsigned int>* Neuronio::getPosicao(){
41    return this->posicao;
42  }
43
44  string Neuronio::getRotulo() {
45    return this->rotulo;
46  }
47
48  bool Neuronio::getMarcado() {
49    return this->marcado;
50  }
51
52  void Neuronio::setPesos(vector<double>* pesos) {
53    this->pesos = pesos;
54    this->dim_entrada = this->pesos->size();
55  }
56
57  void Neuronio::setRotulo(string rotulo) {
58    this->rotulo = rotulo;
59  }
60
61  void Neuronio::setMarcado(bool marcado) {
62    this->marcado = marcado;
63  }
64
65  double Neuronio::getDistancia(const vector<double> &vetor) {
66    return Calculos::calculaDistancia(vetor, *this->pesos);
67  }
68
69  void Neuronio::normaliza() {
70    Calculos::normalizaVetor(this->pesos);
71  }
```

```
72
73 double Neuronio::calculaDistanciaEspacial(Neuronio* n) {
74   // Converte os vetores de inteiro para double
75   vector<double> this_pos(this->dim_saida);
76   for(unsigned int i = 0; i < this->posicao->size(); i++) {
77     this_pos.at(i) = double(this->posicao->at(i));
78   }
79
80   vector<double> n_pos(this->dim_saida);;
81   for(unsigned int i = 0; i < n->getPosicao()->size(); i++)
82     n_pos.at(i) = (double)n->getPosicao()->at(i);
83
84   return Calculos::calculaDistancia(n_pos,this_pos);
85 }
86
87 double Neuronio::calculaVizinhanca(Neuronio* n, double sigma) {
88   return exp(-pow((this)->
89     calculaDistanciaEspacial(n),2)/(2*pow(sigma,2)));
90 }
91
92 /* Faz a equação:
93  * w(n+1) = w(n) + eta(n)*h(i(x),n)*[x - w(n)]
94  * E depois normaliza o vetor de pesos sinápticos */
95 void Neuronio::atualiza(Neuronio* vencedor, Dado* dado,
96   double eta, double sigma) {
97   // Acha o nível de vizinhança
98   double h = this->calculaVizinhanca(vencedor,sigma);
99   // Define o quão será atualizado o neurônio
100  double plasticidade = eta*h;
101
102  // Acha a subtração x - w
103  vector<double> dif = *(dado->getDados()) - *(this->pesos);
104
105  // Multiplica pela plasticidade
106  dif = plasticidade * dif;
107
108  // Altera os pesos do neurônio
109  *(this->pesos) =  *(this->pesos) + dif;
110
```

```
111     // Normaliza o vetor de pesos do neurônio
112     this->normaliza();
113 }
114
115 string Neuronio::toString() {
116     ostringstream str("");
117
118     str << "Neuronio pos(" << this->posicao->at(0) << "," <<
119       this->posicao->at(1) << "): ";
120     for(unsigned int i = 0; i < this->dim_entrada; i++) {
121       str << this->pesos->at(i) << " ";
122     }
123
124     return str.str();
125 }
```

A.4 Classe Arranjo

Cabeçalho

Código A.7: Arranjo.h

```
1  /* Arranjo de neurônios bidimensional e quadrado.
2   * Ou seja, com largura e altura idênticas (L*L) */
3
4  #ifndef ARRANJO_H_
5  #define ARRANJO_H_
6
7  #include <list>
8  #include <vector>
9
10 #include "Neuronio.h"
11 #include "Dado.h"
12
13 using namespace std;
14
15 class Arranjo {
16     // Contêiner onde estarão os neurônios
17     list<Neuronio*>* neuronios;
```

```
18    // Dimensão dos vetores de pesos sinápticos do neurônios
19    unsigned int dimensao;
20
21    // Largura (e altura também) do arranjo
22    unsigned int largura;
23    unsigned int tamanho; // Tamanho do arranjo
24
25    // Cria uma posição 2D dado um inteiro
26    vector<unsigned int>* criaPosicao(unsigned int n);
27    // Cria um neurônio na posição correta no arranjo
28    Neuronio* criaNeuronio(unsigned int n);
29
30  public:
31    Arranjo(unsigned int largura, unsigned int dimensao_entrada);
32    virtual ~Arranjo();
33
34    // Gets e sets
35    list<Neuronio*>* getNeuronios();
36    unsigned int getDimensao();
37    unsigned int getLargura();
38    unsigned int getTamanho();
39    void setNeuronios(list<Neuronio*>* neuronios);
40
41    // Retorna o iterador para o neurônio vencedor
42    list<Neuronio*>::iterator getVencedor(Dado* dado,
43      bool marcar = false);
44  };
45
46  #endif /*ARRANJO_H_*/
```

Implementação

Código A.8: Arranjo.cpp

```
1  #include "Arranjo.h"
2
3  vector<unsigned int>* Arranjo::criaPosicao(unsigned int n) {
4    // Cálculo das posições
5    unsigned int pos_x = n % this->largura;
```

```
6     unsigned int pos_y = n / this->largura;
7
8     vector<unsigned int>* pos = new vector<unsigned int>(2);
9     pos->at(0) = pos_x; pos->at(1) = pos_y;
10
11    return pos;
12  }
13
14  Neuronio* Arranjo::criaNeuronio(unsigned int n) {
15    return new Neuronio(this->dimensao,this->criaPosicao(n));
16  }
17
18  Arranjo::Arranjo(unsigned int largura,
19    unsigned int dimensao_entrada) {
20    this->largura = largura;
21    this->tamanho = largura*largura; // Pois é um arranjo quadrado
22    this->dimensao = dimensao_entrada;
23
24    // Cria a lista de neurônios
25    this->neuronios = new list<Neuronio*>();
26
27    // Criação dos neurônios
28    for(unsigned int i = 0; i < this->tamanho; i++) {
29      this->neuronios->push_back(this->criaNeuronio(i));
30    }
31  }
32
33  Arranjo::~Arranjo() {
34    delete this->neuronios;
35  }
36
37  list<Neuronio*>* Arranjo::getNeuronios() {
38    return this->neuronios;
39  }
40
41  unsigned int Arranjo::getDimensao() {
42    return this->dimensao;
43  }
44
```

```
45 unsigned int Arranjo::getLargura() {
46   return this->largura;
47 }
48
49 unsigned int Arranjo::getTamanho() {
50   return this->tamanho;
51 }
52
53 void Arranjo::setNeuronios(list<Neuronio*>* neuronios) {
54   this->neuronios = neuronios;
55 }
56
57 /* Faz a competição entre os neurônios para descobrir
58  * quem é o vencedor:
59  * O vencedor é o neurônio cujo vetor de pesos sinápticos tem
60  * a menor distância em relação ao vetor de dados apresentado */
61 list<Neuronio*>::iterator Arranjo::getVencedor(Dado* dado,
62   bool marcar) {
63   // Obtém o primeiro neurônio
64   list<Neuronio*>::iterator aux = this->neuronios->begin();
65   // Guarda em vencedor e atualiza aux
66   list<Neuronio*>::iterator vencedor = aux++;
67
68   double menor_distancia = (*vencedor)->
69     getDistancia(*(dado->getDados()));
70
71   /* Percorre os neurônios para descobrir qual tem
72    * menor distância */
73   for(; aux != this->neuronios->end(); aux++) {
74     if((*aux)->getDistancia(*(dado)->getDados()) <
75       menor_distancia && !(*aux)->getMarcado() ||
76       (*vencedor)->getMarcado()) {
77       vencedor = aux;
78       menor_distancia = (*vencedor)->getDistancia(*(dado)->
79         getDados());
80     }
81   }
82
83   if(marcar) { // Tem utilidade no Mapa Contextual
```

```
84      (*vencedor)->setMarcado(true);
85    }
86
87    return vencedor;
88  }
```

A.5 Classe SOM

Cabeçalho

Código A.9: SOM.h

```
1  /* Classe que implementa o algoritmo SOM de Kohonen.
2   * O algoritmo utilizado é o incremental */
3
4  #ifndef SOM_H_
5  #define SOM_H_
6
7  #include <list>
8  #include <vector>
9  #include <cmath>
10 #include <cstdlib>
11 #include <ctime>
12
13 #include "Calculos.h"
14 #include "Neuronio.h"
15 #include "Arranjo.h"
16 #include "Dado.h"
17
18 using namespace std;
19
20 class SOM {
21   Arranjo* arranjo; // Arranjo de neurônios
22
23   unsigned int dimensao_entrada; // Dimensão de entrada
24
25   // Valores iniciais dos parâmetros sigma e eta
26   double sigma_ini, eta_ini;
27   double tau1, tau2; // Valores das constantes temporais
```

```
28
29    // Calcula a largura da vizinhança
30    double calculaSigma(unsigned int tempo);
31    // Calcula a taxa de aprendizado
32    double calculaEta(unsigned int tempo);
33
34    // Gera um inteiro aleatório
35    unsigned int geraRand(unsigned int n);
36    // Gera um vetor de pesos aleatório
37    vector<double> geraVetorRand();
38
39    /* Busca se há um neurônio com vetor de pesos
40     * com determinado valor */
41    Neuronio* buscaNeuronio(vector<double> v);
42    // Inicializa os neurônios do arranjo com valores aleatórios
43    void inicializaRand();
44
45    // Desmarca todos os dados
46    void desmarcaDados(vector<Dado*>* dados);
47
48    // Verifica se todos os dados foram marcados
49    bool todosDadosMarcados(vector<Dado*>* dados);
50    // Obtém um dado ainda não marcado
51    Dado* getDadoRand(vector<Dado*>* dados);
52
53    // Atualiza todos os neurônios
54    void atualizaNeuronios(Neuronio* vencedor, Dado* dado,
55      double eta, double sigma);
56
57  public:
58    SOM(unsigned int largura, unsigned int dimensao_entrada,
59      double sigma, double tau2 = 1000, double eta = 0.1);
60    virtual ~SOM();
61
62    // Treina o SOM
63    void treinaSOM(vector<Dado*>* dados, unsigned int iteracoes,
64      bool inicializa = true);
65
66    // Gets e sets
```

```
67    double getSigmaIni();
68    double getEtaIni();
69    double getTau1();
70    double getTau2();
71    Arranjo* getArranjo();
72
73    void setSigmaIni(double sigma);
74    void setEtaIni(double eta);
75    void setTau1(double tau1);
76    void setTau2(double tau2);
77 };
78
79 #endif /*SOM_H_*/
```

Implementação

Código A.10: SOM.cpp

```
1  #include "SOM.h"
2
3  double SOM::calculaSigma(unsigned int tempo) {
4    return this->sigma_ini * exp(-(double(tempo))/this->tau1);
5  }
6
7  double SOM::calculaEta(unsigned int tempo) {
8    return this->eta_ini * exp(-(double(tempo))/this->tau2);
9  }
10
11 // Gera um inteiro aleatório
12 unsigned int SOM::geraRand(unsigned int n) {
13   // Seguindo dica do Stroustrup :P !
14   return int(double(rand())) % n;
15 }
16
17 // Gera um vetor de pesos aleatório e normalizado
18 vector<double> SOM::geraVetorRand() {
19   unsigned int sinal;
20   double valor;
21   vector<double> vetor(this->dimensao_entrada);
```

```
22
23    // Gera o vetor
24    for(unsigned int i = 0; i < this->dimensao_entrada; i++) {
25      // Para ficar entre 0 e 1
26      valor = double(this->geraRand(RAND_MAX))/RAND_MAX;
27      sinal = this->geraRand(2); // Gera 0 ou 1
28      if(sinal == 1)
29        vetor.at(i) = valor; // Número positivo
30      else
31        vetor.at(i) = -valor; // Negativo
32    }
33
34    Calculos::normalizaVetor(&vetor);
35    return vetor;
36  }
37
38  /* Busca se há um neurônio com vetor de pesos com
39   * determinado valor */
40  Neuronio* SOM::buscaNeuronio(vector<double> v) {
41    // Percorre os neurônios do arranjo
42    for(list<Neuronio*>::iterator i = this->arranjo->
43      getNeuronios()->begin();
44      i != this->arranjo->getNeuronios()->end(); i++) {
45      if(*((*i)->getPesos()) == v)
46        return *i;
47    }
48
49    return NULL; // Caso não encontre
50  }
51
52  // Inicializa os neurônios do arranjo com valores aleatórios
53  void SOM::inicializaRand() {
54    vector<double> aux;
55    vector<double>* aux2;
56
57    /* Percorre os neurônios do arranjo, criando valores
58     * aleatórios para seus pesos */
59    for(list<Neuronio*>::iterator i = this->arranjo->
60      getNeuronios()->begin();
```

```
61      i != this->arranjo->getNeuronios()->end(); i++) {
62      do {
63        aux = this->geraVetorRand();
64        // Para que não haja vetores iguais
65      } while(this->buscaNeuronio(aux) != NULL);
66
67      aux2 = (*i)->getPesos();
68      *aux2 = aux;
69    }
70  }
71
72  // Desmarca todos os dados
73  void SOM::desmarcaDados(vector<Dado*>* dados) {
74    for(vector<Dado*>::iterator i = dados->begin();
75      i != dados->end(); i++) {
76      (*i)->setMarcado(false);
77    }
78  }
79
80  // Retorna true quando todos estiverem marcados
81  bool SOM::todosDadosMarcados(vector<Dado*>* dados) {
82    for(vector<Dado*>::iterator i = dados->begin();
83      i != dados->end(); i++) {
84      if((*i)->getMarcado() == false)
85        return false;
86    }
87    return true;
88  }
89
90  // Obtém um dado ainda não marcado de forma aleatória
91  Dado* SOM::getDadoRand(vector<Dado*>* dados) {
92    unsigned int rnd;
93    Dado* d;
94
95    do {
96      // Gera um número aleatório
97      rnd = this->geraRand(dados->size());
98      d = dados->at(rnd); // Obtém um dado
99    } while(d->getMarcado() &&
```

```
100       !this->todosDadosMarcados(dados));
101
102     if(d != NULL)
103       d->setMarcado(true); // Marca o dado
104     return d;
105   }
106
107   // Atualiza todos os neurônios
108   void SOM::atualizaNeuronios(Neuronio* vencedor, Dado* dado,
109     double eta, double sigma) {
110     // Percorre todos os neurônios e atualiza-os
111     for(list<Neuronio*>::iterator i = this->arranjo->
112       getNeuronios()->begin();
113       i != this->arranjo->getNeuronios()->end(); i++) {
114       (*i)->atualiza(vencedor,dado,eta,sigma);
115     }
116   }
117
118   SOM::SOM(unsigned int largura, unsigned int dimensao_entrada,
119     double sigma, double tau2, double eta) {
120
121     this->sigma_ini = sigma;
122     this->tau2 = tau2;
123     this->eta_ini = eta;
124
125     this->tau1 = this->tau2/log(this->sigma_ini);
126
127     this->arranjo = new Arranjo(largura,dimensao_entrada);
128     this->dimensao_entrada = dimensao_entrada;
129
130     srand(1000); // Inicializa o gerador de números aleatórios
131   }
132
133   SOM::~SOM() {
134     delete this->arranjo;
135   }
136
137   // Faz o treinamento segundo o algoritmo incremental
138   void SOM::treinaSOM(vector<Dado*>* dados,
```

```
139     unsigned int iteracoes, bool inicializa) {
140     unsigned int n_it = 0;
141     double sigma, eta; // Parâmetros para plasticidade
142     // Iterador para o neurônio vencedor
143     list<Neuronio*>::iterator vencedor;
144     Dado* dado; // Ponteiro para um objeto Dado
145
146     if(inicializa)
147         // Inicializa os neurônios do arranjo de forma aleatória
148       this->inicializaRand();
149
150     this->desmarcaDados(dados); // Desmarca todos os dados
151
152     // Repete até o número máximo de iterações
153     while(n_it < iteracoes) {
154       // Calcula a largura da vizinhança
155       sigma = this->calculaSigma(n_it);
156       // Calcula a taxa de aprendizado
157       eta = this->calculaEta(n_it);
158
159       /* Apresenta todos os dados de forma aleatória e atualiza
160        *  os pesos sinápticos */
161       while(!this->todosDadosMarcados(dados)) {
162         // Obtém um dado de forma aleatória
163         dado = this->getDadoRand(dados);
164         // Faz a competição
165         vencedor = this->arranjo->getVencedor(dado);
166         // Atualiza os neurônios
167         this->atualizaNeuronios(*vencedor,dado,eta,sigma);
168       }
169
170       this->desmarcaDados(dados);
171       n_it++;
172     }
173 }
174
175 // Gets e sets
176 double SOM::getSigmaIni() {
177     return this->sigma_ini;
```

```
178 }
179
180 double SOM::getEtaIni(){
181   return this->eta_ini;
182 }
183
184 double SOM::getTau1(){
185   return this->tau1;
186 }
187
188 double SOM::getTau2(){
189   return this->tau2;
190 }
191
192 Arranjo* SOM::getArranjo(){
193   return this->arranjo;
194 }
195
196 void SOM::setSigmaIni(double sigma){
197   this->sigma_ini = sigma;
198 }
199
200 void SOM::setEtaIni(double eta){
201   this->eta_ini = eta;
202 }
203
204 void SOM::setTau1(double tau1){
205   this->tau1 = tau1;
206 }
207
208 void SOM::setTau2(double tau2) {
209   this->tau2 = tau2;
210 }
```

A.6 Classe MapaContextual

Cabeçalho

Código A.11: MapaContextual.h

```
1  /* Mapa Contextual, visto na Seção 3.4.2
2   * Serve como ferramementa para auxiliar a leitura do SOM */
3
4  #ifndef MAPACONTEXTUAL_H_
5  #define MAPACONTEXTUAL_H_
6
7  #include <list>
8  #include <vector>
9  #include <string>
10 #include <sstream>
11
12 #include "Neuronio.h"
13 #include "Arranjo.h"
14 #include "Dado.h"
15
16 using namespace std;
17
18 class MapaContextual {
19   // Desmarca todos os neurônios
20   void desmarcaNeuronios(list<Neuronio*>* neuronios);
21   // Verifica se todos os neurônios estão marcados
22   bool todosNeuroniosMarcados(list<Neuronio*>* neuronios);
23
24 public:
25   MapaContextual();
26   virtual ~MapaContextual();
27
28   // Gera um Mapa Contextual
29   string geraMapa(vector<Dado*>* dados, Arranjo* arranjo);
30 };
31
32 #endif /*MAPACONTEXTUAL_H_*/
```

Implementação

Código A.12: MapaContextual.cpp

```
1  #include "MapaContextual.h"
```

```
2
3  MapaContextual::MapaContextual() {
4  }
5
6  MapaContextual::~MapaContextual() {
7  }
8
9  // Desmarca todos os neurônios
10 void MapaContextual::desmarcaNeuronios(
11   list<Neuronio*>* neuronios) {
12   for(list<Neuronio*>::iterator i = neuronios->begin();
13     i != neuronios->end(); i++) {
14     (*i)->setMarcado(false);
15   }
16 }
17
18 // Verifica se todos os neurônios estão marcados
19 bool MapaContextual::todosNeuroniosMarcados(
20   list<Neuronio*>* neuronios) {
21   for(list<Neuronio*>::iterator i = neuronios->begin();
22     i != neuronios->end(); i++) {
23     if((*i)->getMarcado() == false)
24       return false;
25   }
26   return true;
27 }
28
29 // Gera um Mapa Contextual
30 string MapaContextual::geraMapa(vector<Dado*>* dados,
31   Arranjo* arranjo) {
32   Dado* d;
33   list<Neuronio*>::iterator n; // Guardará o vencedor
34
35   // Desmarca todos os neurônios
36   this->desmarcaNeuronios(arranjo->getNeuronios());
37
38   // Cria o Mapa Contextual
39   unsigned int i = 0;
40   while(!this->todosNeuroniosMarcados(arranjo->
```

```
41      getNeuronios())) {
42      while((i < dados->size()) &&
43        !this->todosNeuroniosMarcados(arranjo->getNeuronios())) {
44        d = dados->at(i);
45        // Acha o vencedor e marca-o
46        n = arranjo->getVencedor(d,true);
47        (*n)->setRotulo(d->getRotulo()); // Rotula o neurônio
48        i++;
49      }
50      i = 0;
51    }
52
53    this->desmarcaNeuronios(arranjo->getNeuronios());
54
55    // Constrói a string com o Mapa Contextual
56    ostringstream mapa("") ;
57    i = 0;
58    unsigned int largura = arranjo->getLargura();
59
60    /* Percorre todos os neurônios e escreve os seus
61     * rótulos em mapa */
62    for(n = arranjo->getNeuronios()->begin(); n != arranjo->
63      getNeuronios()->end();
64      n++, i++) {
65      if((i % largura == 0) && (i / largura != 0))
66        mapa << endl;
67
68      mapa << (*n)->getRotulo() << "␣␣";
69    }
70
71    return mapa.str();
72  }
```

Referências Bibliográficas

Barto, A. G., Bradtke, S. J. & Singh, S. P. (1995), 'Learning to act using real-time dynamic programming', *Artificial Intelligence* **73(1)**, 81–138.

de Oliveira, R. N. (2006), Aprendizagem de máquina em um ambiente para negociações automatizadas, (dissertação de mestrado), Universidade Federal de Campina Grande, Campina Grande, PB.

Haykin, S. (1999), *Neural Networks - A Comprehensive Foundation*, 2 ed., Prentice-Hall, New Jersey, USA.

Honkela, T., Kaski, S., Lagus, K. & Kohonen, T. (1998), 'Websom–self-organizing maps of document collections', *Neurocomputing* **21**, 101–117.

Kaski, S., Kangas, J. & Kohonen, T. (1998), 'Bibliography of the self-organizing map (som) papers: 1981-1997', *Neural Computing Surveys* **1**, 1–176.

Kohonen, T. (1982), 'Self-organized formation of topologically correct feature maps', *Biological Cybernetics* **43**, 59–69.

Kohonen, T. (2001), *Self-Organizing Maps*, Springer, Berlin.

Kohonen, T. (2006), 'Self-organizing neural projections', *Neural Networks* **19**, 723–733.

Lingras, P., Hogo, M., Snorek, M. & West, C. (2005), 'Temporal analysis of clusters of supermarket customers: conventional versus interval set approach', *Information Sciences* **172**, 215–240.

Luger, G. F. (2004), *Inteligência Artificial - Estruturas e Estratégias para a Solução de Problemas Complexos*, 4 ed., Bookman, Porto Alegre, RS.

Malone, J., McGarry, K., Wermter, S. & Bowerman, C. (2005), 'Data mining using rule extraction from kohonen self-organising maps', *Neural Computing & Applications* **15**, 9–17.

McCulloch, W. S. & Pitts, W. (1943), 'A logical calculus of the ideas immanent in nervous activity', *Bulletin of Mathematical Biophysics* **5**, 115–137.

Oja, M., Kaski, S. & Kohonen, T. (2002), 'Bibliography of the self-organizing map (som) papers: Addendum 1998-2001', *Neural Computing Surveys* **3**, 1–156.

Russell, S. & Norvig, P. (2004), *Inteligência Artificial*, 2 ed., Elsevier, Rio de Janeiro.

Stroustrup, B. (2000), *A Linguagem de Programação C++*, 3 ed., Bookman, Porto Alegre.

Ultsch, A. (2003), U*-matrix: a tool to visualize clusters in high dimensional data, Technical Report 36, University of Marburg, Department of Computer Science.

Ultsch, A., Mantyk, R. & Halmans, G. (1993), Connectionist knowledge acquisition tool: Conkat, *in* 'Hand J (ed) Artificial intelligence frontiers in statistics: AI and statistics III', Chapman and Hall, pp. 256–263.

Yeo, N., Lee, K., Venkatesh, Y. & Ong, S. (2005), 'Colour image segmentation using the self-organizing map and adaptive resonance theory', *Image and Vision Computing* **23**, 1060–1079.

Zuchini, M. H. (2003), Aplicações de mapas auto-organizáveis em mineração de dados e recuperação de informação, (dissertação de mestrado), Faculdade de Engenharia Elétrica e de Computação (FEEC - UNICAMP), Campinas, SP.