

CAPÍTULO 5

PROJETO E PROGRAMAÇÃO DO MAPA AUTO-ORGANIZÁVEL

5.1 Introdução

Para o caso de análise de dados geoespaciais multivariados é necessário que os resultados gerados a partir do Mapa Auto-Organizável possam ser visualizados graficamente por meio de mapas. Para que isto seja possível, sem a necessidade de importação/exportação de arquivos, uma solução possível, e aqui utilizada, é a conexão do algoritmo SOM à biblioteca de acesso ao banco de dados geográficos **TerraLib**, desenvolvida no IN-PE/DPI (Câmara et al., 2002). A TerraLib é uma biblioteca de classes voltada para o desenvolvimento de sistemas de informação geográfica customizados. A **TerraLib** foi desenvolvida na linguagem de programação C++, através da aplicação de modernas técnicas de programação, como padrões de projeto (Gamma et al., 1995), programação genérica (Stroustrup, 2000), STL (Musser e Saini, 1996) e programação multi-paradigma (Coplien, 2000).

Embora o algoritmo padrão de treinamento da rede SOM seja conceitualmente simples, sua implementação requer uma série de cuidados. Kohonen (2001), afirma que a maioria das implementações não se preocupa com os detalhes do processo de construção do algoritmo. Ciente deste problema, a equipe de pesquisas em Mapas Auto-Organizáveis da universidade da Finlândia desenvolveu dois pacotes de software que implementam a rede SOM. O SOM PAK, desenvolvido em C (Kohonen et al., 1995) e o SOM ToolBox, desenvolvido em MatLab (Vesanto et al., 1999). Ambos possuem código fonte aberto, são gratuitos e possuem características importantes para este projeto como confiabilidade, disponibilidade do código fonte e funcionalidade.

Porém, após a análise dos pacotes SOM PAK e SOM ToolBox, verificou-se que ambos demandariam um esforço muito grande de conexão com a biblioteca **TerraLib**, uma vez que estes pacotes foram desenvolvidos em linguagens distintas da C++ e não usam, extensivamente, conceitos de programação moderna, o que acarretaria sérias dificuldades de manutenção. Portanto, apesar das vantagens em termos de confiabilidade e funcionalidade, decidiu-se desenvolver um novo código para o algoritmo SOM. Outros pacotes foram analisados, mas não atendiam simultaneamente os requisitos de disponibilidade do código fonte, confiabilidade e manutenibilidade como o SNNS (Zell et al., 1992) e Nenet (Kohonen, 2001). O pacote SOM ToolBox foi usado neste projeto como mecanismo de comparação e teste do algoritmo SOM desenvolvido.

O desenvolvimento de qualquer simulador neural exige preocupações nas áreas de depuração do código, processamento de alto desempenho, com ou sem paralelização da implementação, e projeto (Lawrence et al., 1996). Este trabalho concentrou-se na elaboração do projeto de implementação baseado no paradigma de Orientação a Objetos (Gamma et al., 1995). Os pacotes SOM PAK e ToolBox auxiliaram na depuração do código projetado e implementado. O projeto consistiu no desenho e implementação de uma biblioteca de classes, SOMLib, que implementa algoritmos e encapsulam dados relativos ao uso da rede SOM para a análise exploratória de dados multivariados, geoespaciais ou não.

5.2 Projeto e programação

Segundo Kohonen (2001), qualquer pacote SOM deve apresentar um conjunto mínimo de características, tais como: permitir que a grade da rede possa ter qualquer dimensão, definição automática das dimensões em função dos auto-valores da matriz de correlação dos padrões de entrada, disposição hexagonal e retangular, aprendizagem em lote e sequencial, função de vizinhança gaussiana e bolha, iniciação linear, tratamento de dados ausentes, algoritmos de visualização e cálculo dos erros de quantização e topológico.

Como observado no Capítulo 2, a rede neural SOM pode variar de diferentes formas. Pode-se ter redes de dimensões variadas, com formatos diferentes da grade de neurônios, funções de vizinhança distintas etc. Representando este conjunto de variações num diagrama de classes (Figura 5.1), pode-se observar a proliferação de classes. Observe-se que todas as características relativas à grade foram encapsuladas nas classes de topologia (2D, 3D ...).

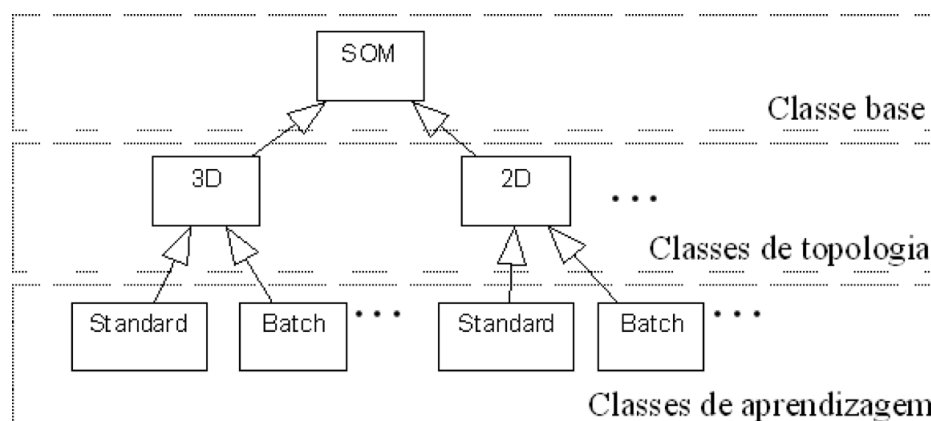


FIGURA 5.1 – Diagrama de Classes para representação das famílias de Mapas Auto-Organizáveis.

As classes foram agrupadas em três categorias: classe base (SOM), classes de topologia (2D, 3D ...) e classes de aprendizagem (Standard, Batch ...). Implementar a biblioteca com base nesta estrutura de classes não configura uma boa idéia, pois, além da duplicação de classes, observa-se um forte acoplamento entre as classes de topologia e de aprendizagem. Para resolver esta questão, dividiu-se o problema em dois: projeto e implementação das classes de aprendizagem e de topologia. Para o problema relativo às classes de aprendizagem tem-se que, a depender do contexto ou necessidade do usuário, deve ser possível variar entre os vários algoritmos de aprendizagem implementados. Pode-se, então, usar o padrão *Strategy* para resolver este problema. O padrão *Strategy* define uma família de algoritmos, encapsula cada um e os faz interoperáveis (Gamma et al., 1995). O diagrama da Figura 5.2 mostra que uma classe abstrata foi criada (*LearningAlgorithm*), pela qual as classes de aprendizagem serão derivadas. O trecho de código em seguida ilustra a implementação da classe base SOM, considerando a estrutura do diagrama de classes (Figura 5.2).

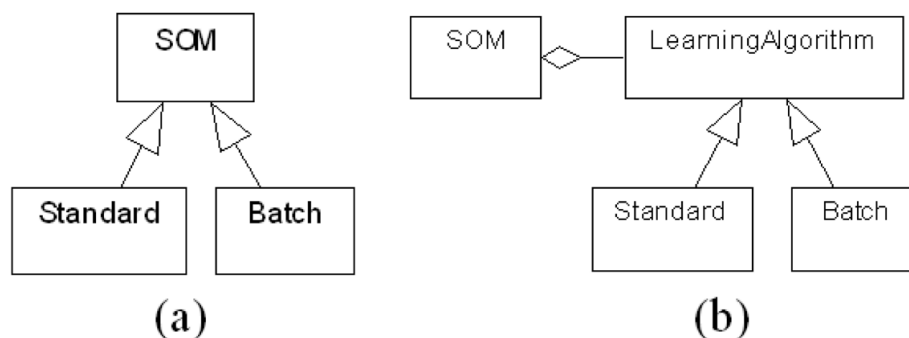


FIGURA 5.2 – Diagrama de Classe: a) Representação da classe base e das classes de aprendizagem; b) Nova estrutura do diagrama -a- baseada no padrão *Strategy*.

Como demonstrado através da Figura 5.3, as questões de topologia e aprendizagem estão “misturadas” de forma que a adição de mais uma classe de topologia implica na reconstrução das classes de aprendizagem relacionadas com a mesma. Este problema foi solucionado com o uso do padrão de projeto *Bridge*. Este padrão desacopla uma abstração de sua implementação, de forma que ambas possam variar independentemente (Gamma et al., 1995). Assim, criou-se mais uma classe abstrata, *TopologyImp*. Foi a partir dessa classe que se originaram as classes concretas de topologia. A Figura 5.4 mostra a nova estrutura de relacionamento entre as classes de topologia e as classes de aprendizagem. Com esta nova estrutura uma mesma implementação de uma classe de topologia pode servir a mais

```

class SOM {
public:
    SOM( Params par ) {          // Constructor
        switch (par.type) {
            case "batch" : _learningAlgorithm = new Batch;
            case "Standard" : _learningAlgorithm = new Standard; }
    }

    ~SOM();          // Destructor
protected:
    LearningAlgorithm * _learningAlgorithm;

    LearningAlgorithm * getLearningAlgorithm() { return _learningAlgorithm;
};
public:
    void Learning() {
        getLearningAlgorithm()->Learning( netParams );
    };
};

```

de uma classe de aprendizagem, sem a necessidade de duplicação de código. Em seguida, tem-se mais um trecho de código da classe abstrata *LearningAlgorithm*.

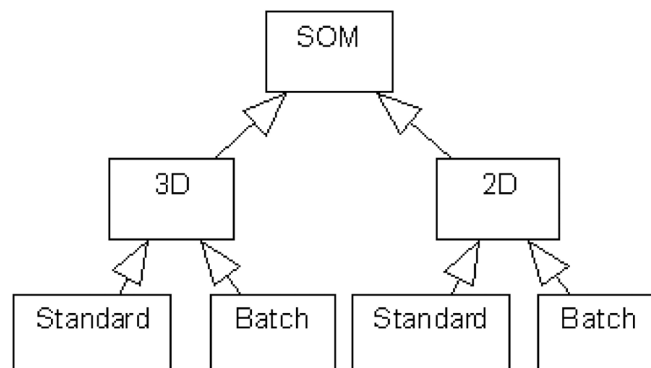


FIGURA 5.3 – Diagrama de Classes. Aqui observa-se o alto acoplamento entre as classes de topologia e de aprendizagem.

Optou-se pela mesma implementação para os dois padrões de projeto usados, mas observe-se que ambas foram motivadas por razões distintas. A Figura 5.5 mostra a configuração final do diagrama de classes após o uso dos padrões. Esta estrutura permitirá uma maior manutenibilidade e possibilidade de reuso de código para a biblioteca SOM-Lib.

Na implementação das classes base SOM e da classe abstrata *LearningAlgorithm* percebe-se que cada uma deve decidir qual objeto criar de acordo com os parâmetros passados no

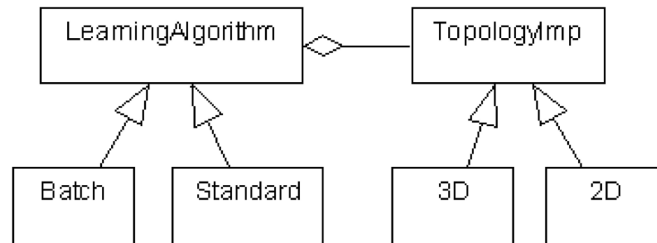


FIGURA 5.4 – Através do padrão *Bridge* separou-se os detalhes de topologia e aprendizagem.

```

class LearningAlgorithm {
public:
    virtual int Learning( Params& par ) = 0;
    TopologyImp * getTopology() { return _topology; };
    ~LearningAlgorithm();

protected:
    LearningAlgorithm(const TopolParams& par) {
        switch( par.type ) {
            case "two" : _topology = new TwoD;
            case "three" : _topology = new ThreeD; }
    };

private:
    TopologyImp * _topology;
};
  
```

construtor de cada classe. Após a passagem de parâmetro, a cláusula *switch* definirá qual objeto construir. Embora seja um método válido, cria a necessidade de se alterar todas as classes que contenham este tipo de cláusula, toda vez que uma nova classe de aprendizagem ou de topologia for implementada. Para este caso, usou-se o padrão de projeto *Abstract Factory*. Este padrão estrutural provê uma interface para a criação de famílias de objetos sem especificar as respectivas classes concretas. Usou-se uma implementação específica deste padrão (Câmara et al., 2001). Nesta implementação, os autores empregaram a programação genérica para definir um *Factory* genérico, cuja função é construir qualquer classe concreta, de um conjunto pré-definido, dispensando o uso de cláusulas do tipo *if..then* e *switch*.

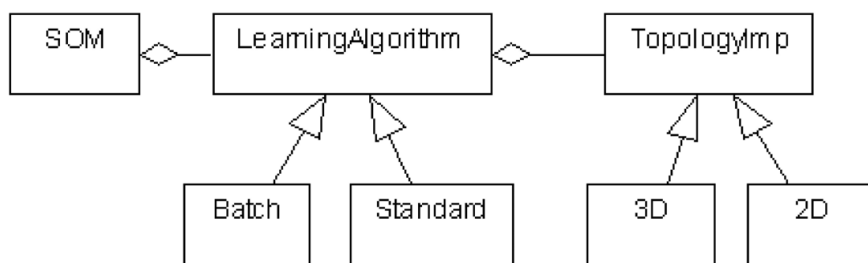


FIGURA 5.5 – Diagrama de Classe final

A Figura 5.6 mostra como ficou a estrutura de classes do diagrama da Figura 5.5, após o uso do padrão *Abstract Factory*. Note-se que, para cada classe de aprendizagem do diagrama da Figura 5.5, foi criada uma classe construtora, *LearningFactory*, *StandardFactory* e *BatchFactory*. A função das classe concretas *StandardFactory* e *BatchFactory* é a de implementar a função *build* da classe *Factory*. O mesmo método foi aplicado no diagrama de classes da Figura 5.4.

A Figura 5.7 mostra a estrutura de classes para a implementação das rotinas de leitura e gravação dos dados, *SOMData*, que alimentaram a rede neural. Optou-se por criar uma classe concreta, *SOMDataCadastro*, para isolar completamente os dados dos detalhes de armazenamento. Assim, a classe *SOMData* transfere todas as responsabilidades de gerenciamento dos dados para a classe *SOMDataCadastro*. Como há várias formas de armazenamento dos dados, usou-se o padrão *Strategy* de forma a facilitar o processo de implementação de novos algoritmos de acesso. Assim, surge a classe abstrata de interface, *ISOMDataRepository*, e as classes concretas derivadas desta e que implementam os métodos de acesso aos dados, *RepositorySOMDataFile*, sistemas de arquivos, e

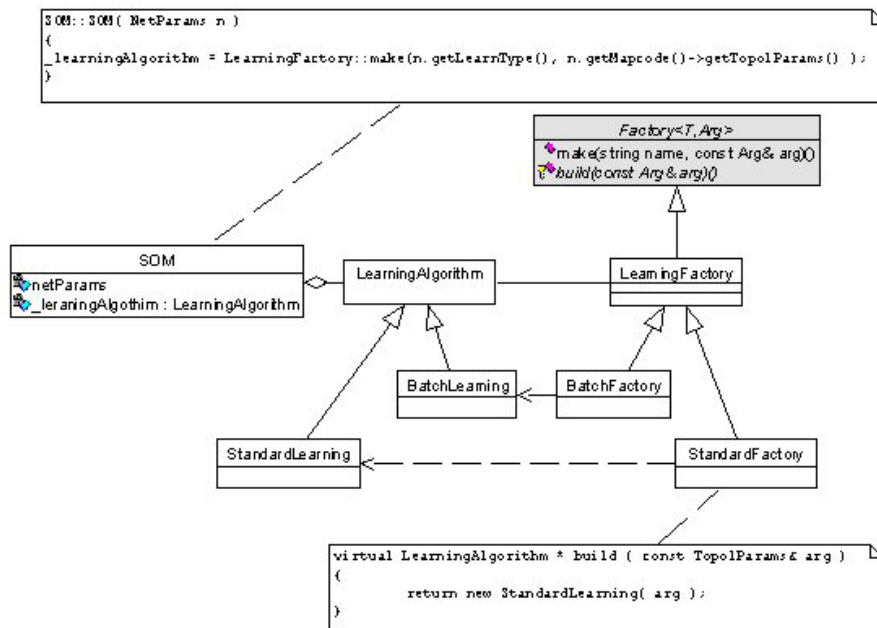


FIGURA 5.6 – Representação do uso do padrão *Abstract Factory* sobre o diagrama de classes da Figura 5.5.

```

class LearningFactory : public Factory<LearningAlgorithm,TopolParams> {
public:
LearningFactory(const string& name): Factory < LearningAlgorithm, TopolParams> (name) {}
}

class BatchLearning : public LearningAlgorithm {
public:
BatchLearning(const TopolParams& s):LearningAlgorithm(s) {};

int Learning ( NetParams& net );
}

class BatchFactory : public LearningFactory {
public:
BatchFactory( const string& name ) : LearningFactory( name ) {};

virtual LearningAlgorithm * build ( const TopolParams& arg )
{ return new BatchLearning( arg ); }
}
  
```

RepositorySOMDataTerralib, banco de dados formato **TerraLib**.

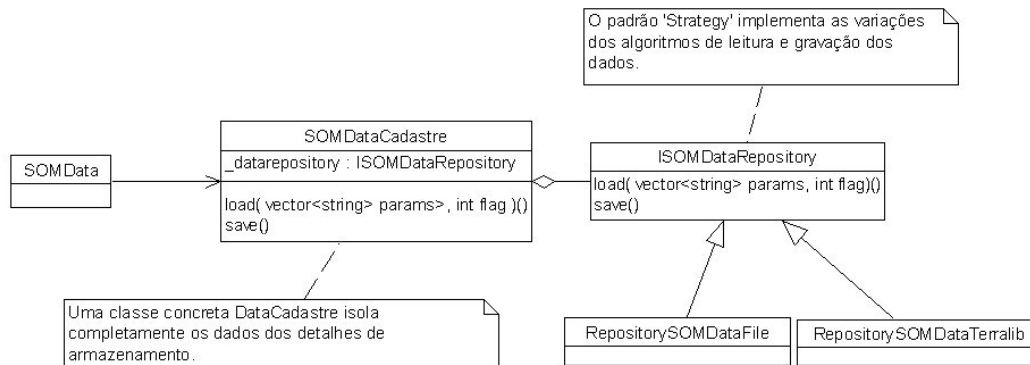


FIGURA 5.7 – Representação da estrutura de classes relativas aos dados e algoritmo de leitura e gravação dos dados de entrada da rede neural.

5.3 Características

No projeto SOMLib implementou-se os algoritmos de aprendizagem em lote e sequencial; as funções de vizinhança gaussiana, bolha e gaussiana cortada; a grade com arranjo hexagonal e retangular; o cálculo do erro de quantização e topológico; a iniciação por interpolação simples e linear; a grade bidimensional.

5.4 Avaliação da biblioteca

Para a avaliação da SOMLib usou-se dois conjuntos de dados da base *UCI Repository of machine learning databases* (Blake e Merz, 1998): Íris e Wine. As análises de separabilidade das classes e comparação com os resultados gerados pelo SOM Toolbox validaram a biblioteca para estes casos.

5.5 Uso da biblioteca SOMLib

A seguir, tem-se um exemplo, em C++, do uso da SOMLib. Neste exemplo, os padrões são lidos a partir de um arquivo de dados, 'dados.pat'. Após a leitura, um SOM com valores *default* é criado, bidimensional, com aprendizagem em lote. Em seguida, os parâmetros da rede são ajustados: dimensão 20x20, disposição hexagonal da grade de neurônios, função de vizinhança gaussiana, raio inicial igual a 15, iniciação linear, 2000 épocas de treinamento. As funções de iniciação, *InitMapcode()*, e de aprendizagem, *Learning()*, são então chamadas. Finalmente, os vetores de código da rede treinada serão gravados no arquivo "mapa_treinado.cod".

```

#include "..\Som.h"
#include "..\SOMDataCadastre.h"
#include "..\RepositorySOMDataFile.h"
#include "..\MapcodeCadastre.h"
#include "..\RepositoryMapcodeFile.h"

void
main() {

    vector<string> params;

    RepositorySOMDataFile repD;
    SOMDataCadastre cadD( repD );
    SOMData * data = new SOMData;
    params.push_back( "dados.pat" ); // Lê os dados do arquivo dados.pat

    SOM * mysom = new SOM; // Cria rede bidimensional com algoritmo
                          // de aprendizagem em lote

    mysom->setData( data );

    mysom->getMapcode()->setNumVar( mysom->getData()->getDimension() );
    mysom->getMapcode()->setDimensions( 0, 20 ); // Rede bidimensional 20x20
    mysom->getMapcode()->setDimensions( 1, 20 );
    mysom->getMapcode()->setLattice( "hexa" ); // Grade hexagonal
    mysom->getMapcode()->setNeighborType( "gaussian" ); // Função gaussiana
    mysom->getMapcode()->CreateCodebook( 20*20, mysom->getData()->getDimension() );

    mysom->setInitNeighbor( 15 ); // Vizinhaça inicial
    mysom->setInitType( LINEAR ); // Iniciação linear
    mysom->setNumIterations( 2000 ); // 2000 épocas

    mysom->InitMapcode(); // Inicia os vetores de código
    mysom->Learning(); // Executa algoritmo de aprendizagem

    RepositoryMapcodeFile repM;
    MapcodeCadastre cadM( repM ); // Salva mapa treinado em arquivo
    cadM.save( mysom->netParams.getMapcode(), "mapa_treinado.cod" );
};

```

5.6 Descrição do sistema *CASA*

A fim de tornar possível a observação visual dos resultados obtidos pelo SOM quanto ao processamento de dados geográficos, foi desenvolvido o sistema *CASA* (*Connectionist Approach for Spatial Analysis of Areal Data*). O sistema *CASA* foi construído sobre as bibliotecas *SOMLib* e **TerraLib**. O sistema é um simulador neural que possibilita a avaliação de Mapas Auto-Organizáveis bidimensionais e implementa um conjunto de ferramentas de apoio à análise exploratória de dados geoespaciais armazenados em bancos de dados geográficos acessíveis via biblioteca **TerraLib**.

Na Figura 5.8, tem-se a tela principal do sistema. Por meio desta é possível fazer toda a parametrização do simulador. São elementos configuráveis a partir desta tela: os parâmetros de estrutura da rede e de aprendizagem, a análise de agrupamentos, a matriz de distância unificada e a conexão com banco de dados geográfico.

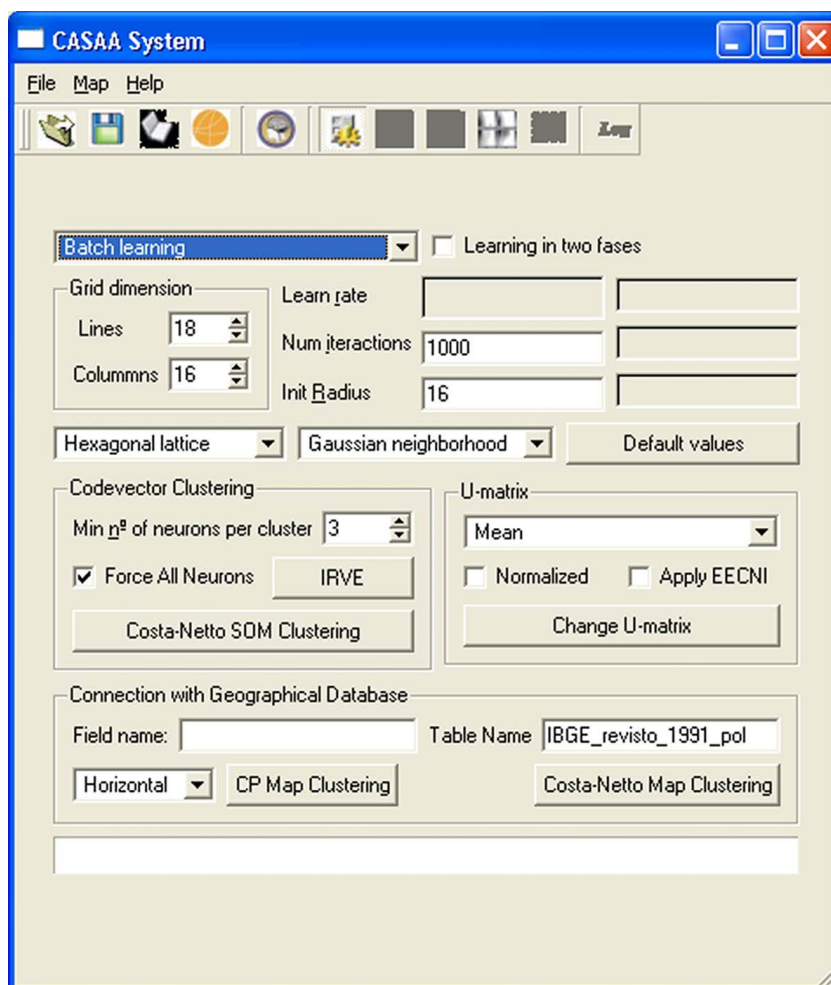


FIGURA 5.8 – Tela inicial do sistema *CASA*.

Para a definição da estrutura da rede o sistema permite a configuração das dimensões (*Grid dimension*) e formato da grade (*lattice*). São parâmetros de aprendizagem configuráveis a taxa de aprendizagem (*learning rate*), o algoritmo de aprendizagem, a função de vizinhança, o número de épocas de aprendizagem (*Num iterations*) e o número de fases de aprendizagem (uma ou duas). Ainda é possível optar por valores de parâmetros *default* (*Default values*) para as dimensões, a organização da grade, a função de vizinhança e o algoritmo de aprendizagem.

A análise de agrupamentos (*Codevector clustering*) está baseada no algoritmo Costa-Netto. Para esta análise, pode-se: optar pelo número de neurônios mínimos por agrupamento (*Min n° of neurons per cluster*), forçar que todos os neurônios especializados sejam rotulados segundo o critério do vizinho mais próximo e, através do botão IRVE, calcular este índice de avaliação da dependência espacial.

As modificações na matriz de distância unificada podem ser feitas através das opções do grupo '*U-matrix*'. Pode-se calcular a U-matriz pela média, mediana, valor máximo e valor mínimo; pode-se, ainda, normalizar os valores e aplicar o algoritmo de Eliminação do Efeito da Cadeia dos Neurônios Inativos - *Apply EECNI* (Costa, 1999).

A conexão com o banco de dados geográfico é feita na leitura e gravação dos dados. Para a leitura dos dados tem-se a tela representada pela Figura 5.9. Através desta tela é feita a conexão com o banco e a leitura das variáveis contidas numa tabela específica e sobre uma determinada restrição da cláusula *WHERE*. Também é neste momento que é lida a matriz de proximidade entre os objetos. Após a leitura dos dados e processamento (aprendizagem) da rede neural, os resultados podem ser gravados na base, através das opções do grupo *Connection with Geographical Database*. Esses dados de gravação estão relacionados com a análise de agrupamentos baseada nos Planos de Componentes (*CP Map Clustering*) ou no algoritmo Costa-Netto (*Costa-Netto Map Clustering*). A visualização destes resultados pode ser efetuada pelo sistema TerraView.

Após a fase de treinamento, o sistema gera uma tela (Figura 5.10) contendo informações sobre as opções do treinamento e resultados. São informações contidas na tela de informações (*Log Info*): arquivo de dados (*Data File*), tipo de aprendizagem (*Learning type*), número de épocas de treinamento (*Training epochs*), raio inicial (*Initial radius*), estrutura da grade de neurônios (*Lattice*), função de vizinhança (*Neighbourhood*), dimensões (*Dimensions*), erros de quantização (*Quantization error*) e topológico (*Topological error*), arquivo de dados do mapa neural (*Mapcode File*), número de agrupamentos encontrados pelo algoritmo Costa-Netto (*Number Cluster*) e dos índices de validação do particionamento dos dados Davies-Bouldin e CDbw.

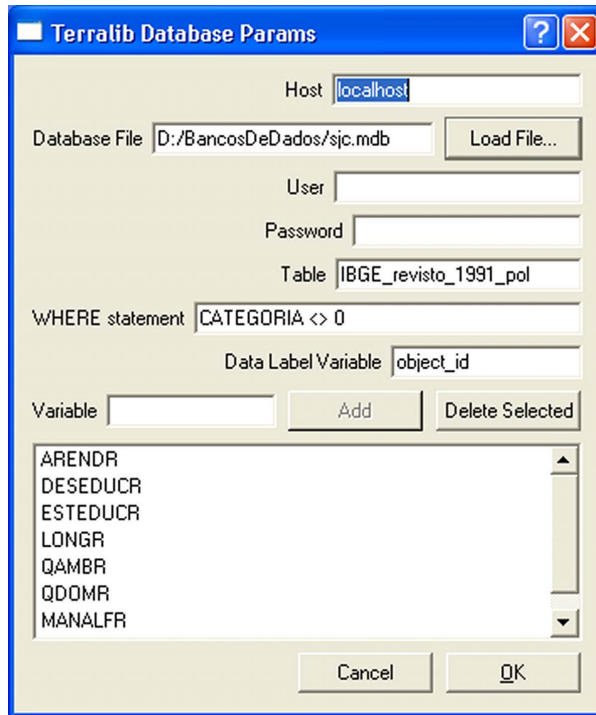


FIGURA 5.9 – Formulário de acesso ao banco de dados geográfico.

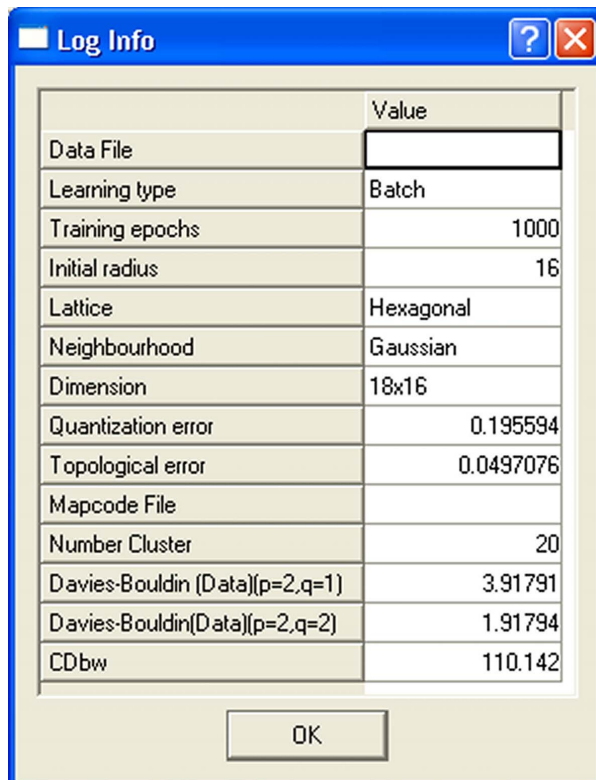


FIGURA 5.10 – Formulário com informações sobre o processo de aprendizagem da rede, número de agrupamentos encontrados pelo algoritmo Costa-Netto e índices de validação deste particionamento.

O resultado do processo de segmentação do Mapa neural, através do algoritmo Costa-Netto, é ilustrado através da coloração do Mapa neural (Figura 5.11). Cada cor representa um agrupamento. Ao clicar num neurônio (círculo) uma nova tela aparece, contendo informações sobre quais dados de entrada estão relacionados com este neurônio (*Label*), qual sua posição (*Neuron number*) e qual o seu agrupamento (*Cluster ID*). O sistema ainda gera os Planos de Componentes (Figura 5.12) e a U-matriz (Figura 5.13).

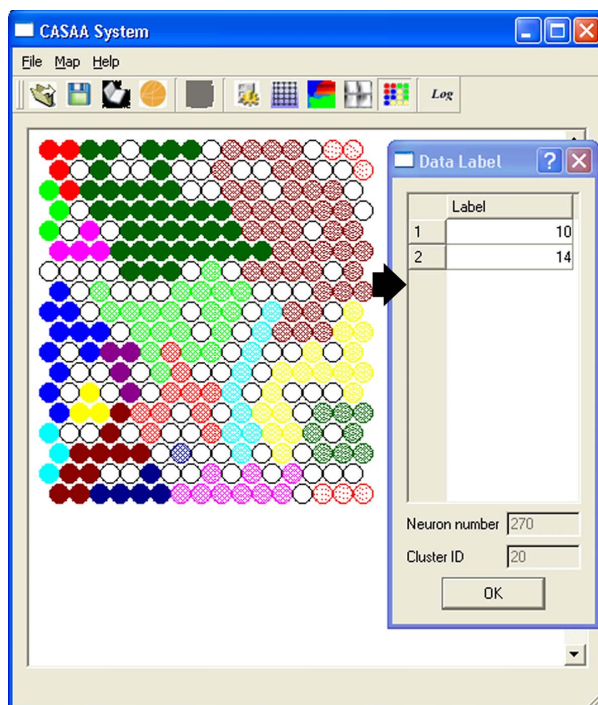


FIGURA 5.11 – Resultado do processo de segmentação do Mapa neural através do algoritmo Costa-Netto. O formulário *Data Label* informa, para cada neurônio, quais padrões de entrada estão relacionados com o mesmo, sua posição (*Neuron number*) e a qual agrupamento pertence (*cluster ID*).

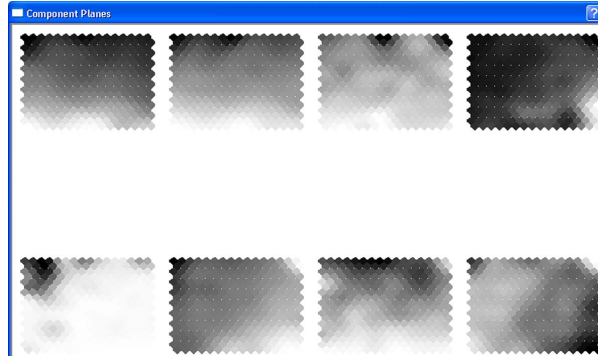


FIGURA 5.12 – Planos de Componentes gerados pelo sistema.



FIGURA 5.13 – U-matriz pelo sistema.

5.7 Sumário

Em função da necessidade de se integrar os algoritmos do SOM com a biblioteca Terralib foi necessário o projeto e programação do Mapa Auto-Organizável. Pacotes disponíveis e de código aberto como o SOM PAK e o SOM ToolBox atendem as necessidades de adaptações no SOM mas apresentam dificuldades de integração com a biblioteca TerraLib e de escalabilidade.

O projeto SOMLib baseou-se no paradigma Orientado a Objetos e em técnicas de programação como padrões de projeto, STL e programação genérica. O objetivo desse projeto foi construir uma biblioteca com alto nível de escalabilidade, facilidade de manutenção e de fácil integração com a TerraLib.

A partir das bibliotecas QT e SOMLib foi desenvolvido o sistema *CASA* - (*Connectionist Approach for Spatial Analysis of Areal Data*), ambiente gráfico que facilita o processo de configuração e uso dos algoritmo do SOM. Este sistema foi usado para a tarefa de treinamento da rede, análise de agrupamentos e comunicação com o banco de dados geográficos. Para visualização da U-matriz e dos Planos de Componentes foi usado o pacote SOM ToolBox.

