Programação Orientada por Objectos aplicada à simulação de redes neuronais

Pedro Kulzer, António Branco, Ana Tomé, Armando Pinho

Resumo - **Neste artigo descrevemos uma representação orientada ao objecto, destinada à implementação de redes neuronais artificiais. Esta representação é constituída por vários objectos-base que possuem, cada um deles, um nível diferente de representação. A ideia essencial centra-se numa representação *bottom-up*, em que se implementam objectos cada vez mais globais que possuem dentro de si, objectos da posição hierárquica imediatamente inferior. Espera-se assim conseguir uma representação global incremental fácil de compreender, imaginar e programar. Descrevemos um modelo global de um sistema neuronal, focando as três características essenciais: topologia, actualização das saídas e treino.**

Abstract - **In this paper, we describe an object oriented representation that is useful for the programmable implementation of artificial neural networks. This representation is basically composed of several base objects which achieve, every one of them, a different level of representation. The main idea is based on a bottom-up representation, where we implement objects that get more and more global, containing other objects of an immediate lower hierarchical position. With all this, it is expected to achieve an incremental global representation which is easy to understand, imagine and program. We describe a global neural system model, focusing the three essential characteristics: topology, recall and train.**

I. INTRODUÇÃO*

Uma rede neuronal é constituída por neurónios ligados entre si, pressupondo-se assim estruturas de processamento local, distribuído e paralelo. No entanto, o princípio de funcionamento da maioria das estruturas neuronais é suportado pelo conceito de camada, definindo-se camada como um grupo de neurónios (com ou sem as mesmas características funcionais), que recebem sinais com a mesma proveniência e enviam sinais com destino comum. Assim, uma rede neuronal pode definir-se como sendo constituída por uma ou várias camadas. Podemos ainda considerar um nível superior nesta hierarquia, definindo um sistema neuronal como sendo constituído por várias redes. Deste modo, concluímos que um sistema neuronal pode ser descrito como sendo uma estrutura hierárquica, constituída por vários níveis de abstracção diferente (blocos). Aos diversos blocos do sistema neuronal são atribuídas características funcionais diferentes, quer a nível da topologia, cálculo das actuais saídas dos diferentes neurónios e treino.

Posto isto, e passando a falar em termos de programação orientada a objectos, podemos dizer que os sistemas neuronais, são sistemas estrutural e conceptualmente complexos, compostos por subsistemas interrelacionados, que por sua vez possuem os seus próprios subsistemas e assim por diante, até que seja atingido o nível mínimo de componentes elementares. Nestes sistemas, as interacções intracomponentes são geralmente mais fortes do que as interacções intercomponentes. Este facto tem o efeito de separar a dinâmica de alta frequência dos componentes (envolvendo interacções na estrutura interna dos respectivos objectos) da dinâmica de baixa frequência (envolvendo interacções entre aqueles objectos). Assim na programação orientada ao objecto, os dados estão ligados às rotinas de código que executam operações sobre eles, encapsulando código e dados no mesmo objecto.

Os sistemas hierarquizados, que são uma particularidade da programação por objectos, são usualmente compostos por apenas alguns subsistemas diferentes, com várias combinações e arranjos, o que corresponde à estrutura do sistema neuronal [6].

No fundo, para uma implementação correcta duma biblioteca neuronal com base em objectos, é imprescindível ter em conta as seguintes regras de programação orientada a objectos [7]:

i) Cada ideia diferente é uma classe diferente, isto é, devem-se implementar classes diferentes para representar comportamentos diferentes de entidades intrinsecamente diferentes (Ex: Ligação, Neurónio).

ii) Cada entidade separada e do mesmo tipo, é também um objecto separado e do mesmo tipo, isto é, cada entidade é uma realização individual da respectiva classe (Ex: Neurónio1, Neurónio2, Neurónio3 na camada I).

iii) Se duas classes possuem dados e código significativos em comum, faz-se uma classe base com esses dados e código, isto é, os dados e código diferente são colocados em classes descendentes desta classe-base (Ex: A classe-base é o SupervisedNeuron e as classes descendentes são as PerceptronNeuron e AdalineNeuron).

* Trabalho realizado no âmbito da disciplina de projecto.

iv) Se uma classe é um tipo mais específico de uma outra classe mais geral, então a primeira será herdada da segunda (Ex: Um neurónio que é treinado pela regra de Hebb é herdado de um outro que já possui código básico para neurónios treináveis).

v) Se um objecto possui outros objectos, então estes últimos serão membros do primeiro (Ex: Uma camada contém neurónios; um neurónio contém ligações).

As vantagens da programação por objectos são mais notórias quando se tentam implementar sistemas complexos em computador, visto que esta forma de programação se aproxima mais da estrutura real abstraída pela mente humana.

A seguir, vamos proceder ao detalhe deste tipo de programação, em que primeiro falaremos do modelo global de um sistema neuronal, em termos absolutamente gerais, sem nenhuma intenção inicial de descrição do respectivo treino, etc. Depois da descrição da estrutura base deste modelo de programação, procederemos ao detalhe mais específico de cada uma das características desse modelo.

II. MODELO LOBAL DE UM SISTEMA NEURONAL

Um modelo neuronal tem três características principais a serem consideradas [5]:

- Topologia (forma das ligações entre os neurónios).
- Actualização dos valores de saída (forma pela qual são calculadas as saídas dos neurónios).
- Treino (forma de alterar os pesos para se atingir um determinado fim).

Para se obter uma grande flexibilidade de implementação dum modelo neuronal, estes princípios devem atender aos seguintes requisitos básicos:

i) A topologia deve permitir qualquer estrutura de ligações (campos receptivos, realimentações, ligação ponto-a-ponto, etc.).

ii) A actualização dos valores de saída, além do processo de apenas uma propagação dos sinais de entrada para a saída, deve também permitir processos mais complexos (actualização da saída, recursivamente, até atingir um valor estável).

iii) Deve ser possível, além de se usar os treinos previamente implementados, redefinir novos métodos de treino que possam surgir (através da capacidade de *polimorfismo* [1] das classes de treino).

Uma rede neuronal é constituída por neurónios ligados entre si, usualmente ordenados em camadas. A representação desta rede é feita a partir da unidade mais simples: a ligação. Os neurónios possuem ligações, estão dispostos em camadas que os contêm, que por sua vez estão contidos em camadas numa rede neuronal. Por último, a rede pode fazer parte dum conjunto de redes, iguais ou não, contidas num objecto ainda mais global, que é o *sistema* neuronal. Assim obtém-se uma representação hierárquica do tipo *bottom-up*, onde cada objecto é contido no objecto imediatamente superior, tal como se pode ver na Fig.1. Nesta figura visualizam-se os diferentes níveis da hierarquia de um sistema neuronal complexo, sendo cada nível constituído por objectos que contêm uma lista ligada de objectos do nível imediatamente inferior, isto é, cada *Sistema* contém uma lista ligada de *redes*, cada *rede* contém *camadas*, cada *camada* contém *neurónios* e cada *neurónio* contém *ligações*.

Na Fig.2 pode-se observar um exemplo de um sistema neuronal muito simples, constituído por apenas uma rede, duas camadas, três neurónios e sete ligações, onde aparecem as componentes neuronais discretizadas. Na parte superior aparece a topologia da rede, enquanto que na parte inferior se detalhou a representação orientada aos respectivos objectos.

Note-se que existe sempre apenas um objecto do tipo *Sistema*, enquanto que existe um ou mais objectos dos restantes tipos para cada objecto-ascendente. Por exemplo, um neurónio tem de ter, pelo menos, uma ligação.

Ao implementar as várias classes de objectos, teremos encapsulados os dados e o respectivo código que os gere. O exemplo que se segue é específico para o caso do neurónio, que será invariavelmente a estrutura de dados mais complexa de um sistema neuronal. Este exemplo está escrito em C++.

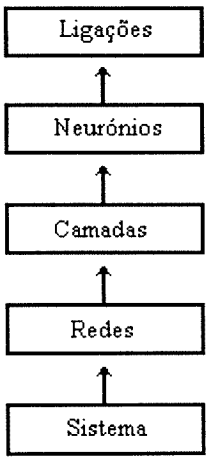

Fig. 1 - Representação hierárquica do tipo *bottom-up*.

```
class TCNeuron:TCStreamObject
{
public:
 TCNeuron();
 virtual ~TCNeuron();
 virtual TCLink  *CreateLink(TCNeuron *Source);
 virtual void    DestroyLink(TCNeuron *Source);
 virtual TWeight     GetBias();
 virtual TLearnRate  GetDecay();
 virtual TLearnRate  GetEpsilon();
 virtual TCLink     *GetLink(TLinkIndex Index);
 virtual TLinkIndex  GetLinkCount();
 virtual TLearnRate  GetMomentum();
 virtual TInOutValue GetOutput();
 virtual TBoolean    HasLink(TCNeuron *Source);
 virtual TBoolean    IsBiasUsed();
 virtual void        Recall;
 virtual void        Save(TCStream *Stream);
 virtual void   SetEpsilon(TLearnRate Epsilon);
 virtual void   SetMomentum(TLearnRate Moment);
 virtual void Train(TInOutValue DesiredOutput);
   . . .
protected:
 TWeight      Bias;
 TBoolean     BiasFrozen;
 TBoolean     BiasUsed;
 TBoolean     Conscious;
 TLeranRate   Decay;
 TLearnRate   Epsilon;
 TLearnRate   Momentum;
   . . .
};
```

O *constructor* TCNeuron() inicializa os dados quando o objecto é criado em memória e o *destructor* ~TCNeuron() destrói-os na altura da sua remoção da mesma.

A palavra-chave *public* permite o acesso aos métodos desta classe, enquanto que o *protected* impede acessos directos às variáveis mas permitindo esse acesso a objectos de classes descendentes desta. Esta última permite que novas classes possam estender as funções dos

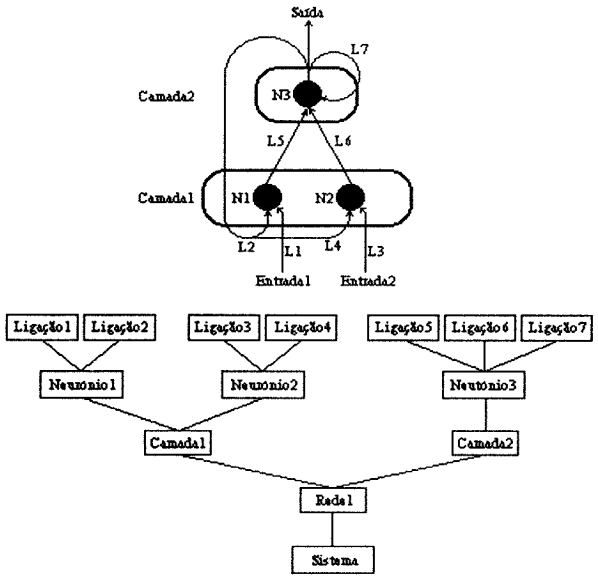

Fig. 2 - Exemplo dum sistema neuronal com apenas uma rede constituída por três neurónios e duas camadas.

métodos de treino e outros.

Apenas se mostraram algumas das muitas funções e campos de dados, que são incluídos neste tipo de estrutura [3]. Note-se que todas as outras estruturas base (ligação, camada, rede e sistema) também serão implementadas de forma semelhante. Cada uma destas classes, tal como esta TCNeuron, implementam funções destinadas às suas várias tarefas (responsabilidades). Para se poder realizar vários tipos de comportamento dos objectos associados a estas classes, temos duas hipóteses de implementação:

i) Existe uma única classe para cada nível hierárquico que possui todas as funções possíveis de uma só vez. Assim, ter-se-iam várias funções que implementariam diferentes comportamentos para uma certa tarefa.

ii) Usa-se a capacidade de polimorfismo através de métodos virtuais em classes derivadas, cada uma com o seu comportamento específico e descendendo de uma classe base abstracta (*abstract base class*).

A desvantagem da primeira hipótese, é que torna a estruturação muito deselegante e "amontoada" quando se deseja acrescentar novos comportamentos a uma *class* já existente, e cujo código, porventura, não é acessível ao programador. Enquanto isso, na segunda hipótese, o programador não necessita modificar esta *base class*, tendo apenas de criar classes descendentes daquela. Estas últimas implementariam as suas próprias funções que alteram o comportamento da classe base, funções essas que seriam automaticamente chamadas por esta classe-base através do processo de funções *virtuais* (*runtime late binding*). Uma boa implementação é tornar esta classe-base uma classe *abstracta*, isto é, os métodos de treino, actualização dos valores de saída, e outros que não realizem funções tão básicas como a escolha ou recolha directos de valores de variáveis, são tornados *virtuais puros* (sem código associado nessa classe).

Por exemplo, o programador dispõe de uma biblioteca neuronal, que possui *classes* pré-definidas (todas elas de alguma forma descendentes da *classe-base abstracta* TCNeuron). Estas classes conteriam o código necessário para a simulação de alguns tipos de neurónios mais utilizadas, tendo a possibilidade de criar novos objectos com comportamentos diferentes (*user-defined*). Todos os outros níveis hierárquicos respeitariam uma forma semelhante de implementação. Na Fig.3 apresenta-se um diagrama de caminhos de descendência de possíveis classes pré-definidas.

Para ilustrar esta forma muito conveniente e simples de redefinir um método, alterando assim o comportamento da classe ascendente, apresentamos de seguida um pequeno exemplo de neurónios em que o comportamento relativo ao treino é alterado.

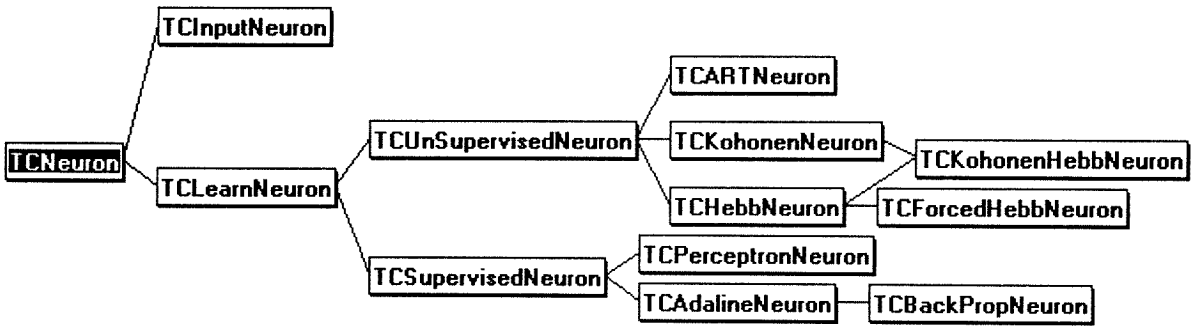

Fig. 3 - Ilustração em forma de diagrama de caminhos de descendência das diferentes classes de neurónios, implementada numa biblioteca neuronal standard. Note-se que se pode conjugar dois tipos de neurónios diferentes numa só classe, usando hereditariedade múltipla, tal como vê no TCKohonenHebbNeuron.

```
class TCHebbNeuron:TCUnSupervisedNeuron
{
. . .
virtual void NormalizeWeights();
virtual void Train();
. . .
};


class TCMyHebbNeuron:TCHebbNeuron
{
. . .
virtual void NormalizeWeights();
. . .
};
```

Neste exemplo, temos uma classe que implementa as funções dum neurónio que aprende segundo a *regra de Hebb*. O método *Train* é chamado sempre que se queira treinar o neurónio, através da classe base TCNeuron. É este método *train* que chama outro método, *NormalizeWeights*, que ocorre ao nível da classe TCHebbNeuron. Neste primeiro objecto TCHebbNeuron, o método *NormalizeWeights* desempenha alguma função *standard* tal como normalizar todos os pesos do respectivo neurónio. A segunda classe foi uma classe definida pelo programador, que vai permitir alterar o comportamento *standard* da primeira classe, redefinindo o método *NormalizeWeights*, que vai desempenhar algo diferente (por exemplo a normalização de todos os pesos excepto alguns). Desta vez, o *Train* vai chamar o *NormalizeWeights* localizado na nova classe TCMyHebbNeuron, por ser declarado virtual. Estamos aqui a utilizar a capacidade de *polimorfismo* dos objectos, já que uma *class* pode manifestar comportamentos diferentes, através do *overriding* de métodos seus em classes derivadas.

Conclui-se daqui que é aconselhável implementar esta biblioteca neuronal base, com todos os métodos ligados ao treino, etc. com a palavra-chave virtual, de forma a permitir a máxima flexibilidade e simplicidade na alteração dos comportamentos dos objectos dessa biblioteca. Em princípio, a classe base TCNeuron, de onde derivam todas as outras, será uma *abstract base-class* que apenas define os cabeçalhos de todos os métodos virtuais possíveis, mas sem nenhum código associado. Este código será implementado nas *classes* derivadas desta, que farão o *override* destes métodos conforme a necessidade. Por exemplo, a *class* TCInputNeuron apenas pode fazer pouco mais do que o *override* do método de actualização dos valores de saída, enquanto que classes mais sofisticadas poderão fazê-lo a todos.

O mesmo tipo de tratamento que se fez anteriormente, pode ser realizado também para as classes TCLink, TCLayer ,TCNetwork e TCNeuralSystem.

Note-se que este modelo global possui dois tipos de hierarquização:

i) Hierarquização a nível da representação de um sistema neuronal, através da individualização das suas diversas componentes abstractas: ligação, neurónio, camada, rede, sistema. Esta tem a ver com a regra v) de programação orientada ao objecto, anteriormente mencionada.

ii) Hierarquização a nível de cada componente anterior, através da criação de classes com comportamentos diferentes mas tarefas (treino, etc.) rigorosamente iguais: neurónios de entrada, supervisionados, não-supervisionados, de Hebb, etc. Esta tem a ver com a regra iv) de programação orientada ao objecto, anteriormente mencionada.

III. TOPOLOGIA

A estrutura topológica é criada através de comandos que estabelecem um certo padrão de ligações (rede completamente ligada, campos receptivos, etc.), ou através de uma linguagem descritiva (Ficheiros *script*), sendo o primeiro mais fácil de utilizar, mas perdendo bastante flexibilidade.

A estrutura anterior, altamente hierarquizada, possui vantagens concretas na representação topológica, sem levantar grandes encargos adicionais à programação, tais como as que se seguem.

i) Cada ligação é uma entidade perfeitamente independente de tudo o resto, o que permite interligar dois neurónios, independentemente da sua localização ou tipo.

ii) Cada neurónio é uma entidade isolada, permitindo além de estruturas baseadas em camadas convencionais, qualquer outro tipo de distribuição (por exemplo um aglomerado de neurónios).

iii) A estrutura em camadas, permite que se criem padrões de ligações entre estas, facilitando a implementação de estruturas em sistemas complexos. (Varrimentos de campos receptivos, interligações recursivas na mesma camada, etc)

iv) A vantagem de ter o objecto *Rede,* como entidade isolada permite criar módulos que são os componentes básicos de uma estrutura neuronal super-complexa. (Permite criar os módulos do córtex visual; a Retina, as Hiper colunas, o classificador final).

v) O sistema tem como finalidade ter uma entidade que suporta as redes.

O modelo global apresentado inicialmente, em que se faz uma distribuição dos vários objectos de um sistema neuronal com base numa hierarquia de listas ligadas, permite que haja também uma distribuição do código, que faz a actualização dos valores de saída. Por outras palavras, há uma distribuição desta responsabilidade por entre os diversos objectos, o que contribui para:

i) Redução a um mínimo de código necessário em cada objecto.

ii) Tornar o código extremamente simples, porque cada objecto apenas se responsabiliza pelo nível hierárquico imediatamente inferior.

iii) Para se obter um código sem erros, com uma manutenção muito mais simplificada.

IV. ACTUALIZAÇÃO DOS VALORES DE SAÍDA

As características funcionais de base, a nível da actualização dos valores de saída dos neurónios, são:

i) Para calcular a saída do neurónio, não interessa a proveniência da ligação, visto que são todas tratadas da mesma forma.

ii) Os neurónios calculam localmente o valor da saída, conforme as respectivas funções de entrada e transferência e tempo de estabilização em caso de recursividade, através de uma mensagem de uma entidade superior.

iii) A camada limita-se a emitir mensagens de actualização dos valores de saída para os neurónios desta.

iv) A rede estabelece a ordem pela qual as camadas de neurónios são actualizadas.

v) O sistema permite que blocos separados tenham valores de saída que podem ser combinados, mas calculados em alturas diferentes.

V. TREINO

As características funcionais de base, a nível do treino dos neurónios, são:

i) O valor das ligações é alterado durante o treino, independentemente umas das outras, permitindo implementar regras que criem e destruam ligações.

ii) Cada neurónio possui uma regra de treino que lhe confere um comportamento próprio, completamente autónomo de todo o resto; o próprio neurónio é que possui toda a responsabilidade de ir utilizar os parâmetros necessários à sua regra de treino (valores de entrada, propagação de erros, etc.) .

iii) A camada envia mensagens a cada neurónio para se treinar (por definição uma camada deve ter a mesma regra de treino para todos os neurónios).

iv) A rede envia mensagens às camadas, que podem ter regras de treino diferentes.

v) O sistema permite englobar vários módulos com regras de treino diferentes.

Uma ilustração simples do código necessário para um ciclo de treino a partir da execução do método TCNetwork::Train() do objecto de rede, para a regra de Hebb simples, é o seguinte:

```
TCNetwork::Train()
{
  for (TWord L=0; L<GetLayerCount(); L++)
    GetLayer(L)->Train();
};


TCLayer::Train()
{
  for (TWord N=0; N<GetNeuronCount(); N++)
    GetNeuron(N)->Train();
};
```

```
TCNeuron::Train()
{
  for (TWord L=0; L<GetLayerCount(); L++)
  {
    TCLink *Link=GetLink(L);
    Link->SetWeight(Link->GetWeight()
                    +(GetEpsilon()*Link-
>GetInput())
  };
  NormalizeWeights();
};
```

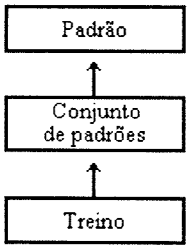

Fig. 4- Hierarquia de classes das estruturas de treino de um sistema neuronal.

O objecto *Sistema* está encarregue de gerar conjuntos de padrões de treino, teste, etc. intermédios (entre redes), conforme o necessário. Assim, cada rede é treinada individualmente, o que reduz significativamente o tempo de espera de resultados de redes intermédias, além de organizar melhor todo o complexo processo de treino de um sistema neuronal de tamanho significativo.

VI. PADRÕES DE TREINO E TESTE

Além da já referida hierarquização das componentes de construção das redes neuronais propriamente ditas, também se pode fazer uma hierarquização semelhante das componentes do respectivo treino. Estas serão subdivididas em Padrão, Conjunto de treino e Objecto de treino, tal como se mostra na Fig.4. Cada objecto do tipo padrão, conterá os dados referentes a um dado padrão. Cada conjunto de treino conterá uma lista ligada de padrões. Cada objecto de treino conterá uma lista ligada de conjuntos de treino, bem como dados e os respectivos métodos para a implementação de uma ou de mais estratégias de treino. Serão estes métodos que vão chamar os métodos de treino das respectivas redes, de acordo com aquelas estratégias.

De forma semelhante ao que acontece nas componentes do sistema neuronal, o Objecto de treino contém uma lista ligada de conjuntos de treino (treino, validação, teste e eventuais outros) e cada Conjunto de treino contém Padrões.

Como cada rede treinável tem de possuir uma estrutura de treino, o objecto Sistema possui ainda uma lista ligada de objectos Treino, associados às suas respectivas redes. Isto é visualizado na Fig.5 como exemplo de um sistema neuronal real.

Neste sistema neuronal temos quatro redes que formam um sistema artificial de visão, para reconhecimento de padrões. As redes da retina e do pós-processador não necessitam de treino, pelo que não têm nenhum objecto de treino associado. Apenas o classificador vai ser alvo de teste com um conjunto de padrões de teste. Como já foi dito anteriormente, estes conjuntos de treino intermédios são gerados pelo objecto do *Sistema* conforme a necessidade. Assim, neste caso específico da Fig. 5, o conjunto de teste será formado pelos padrões de saída do pós-processador, que vão servir para testar o classificador.

Resta dizer que tudo o que foi dito sobre *polimorfismo* e redefinição de métodos, também é aplicável a este conjunto de classes para treino. Pode-se, assim, redefinir o comportamento destas classes para conseguir alterar as estratégias de treino pré-definidas na biblioteca *standard* (tais como o critério de paragem, a sequência de apresentação de padrões, etc.).

Um exemplo de hierarquização de diferentes classes descendentes da classe-base abstracta TCPattern, é o que pode ver na Fig.6. Aqui, temos um objecto especializado para cada tipo de dados.

VII. CONCLUSÕES

A representação hierarquizada por classes de um sistema neuronal complexo apresentada possui todas as vantagens da programação orientada a objectos (polimorfismo, hereditariedade, encapsulamento, abstracção, reuso de código, modularidade) [1]. Estas características são amplamente usadas na vertente da topologia, actualização de saídas e treino da rede neuronal. Conseguiu-se, deste modo, atingir os requisitos a que um simulador de redes neuronais deve obedecer: a topologia deve permitir qualquer estrutura de ligações; a actualização dos valores de saída, além do processo de apenas permitir uma propagação dos sinais de entrada para a saída, deve também permitir processos mais complexos; deve ser possível, além de se usar os treinos previamente implementados, redefinir novos métodos de treino que possam surgir.

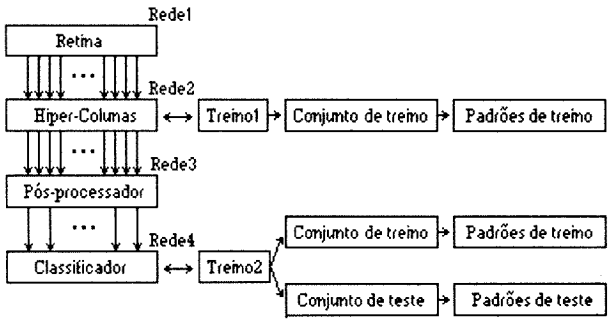

Fig. 5- Ilustração da relação estreita rede-treino.

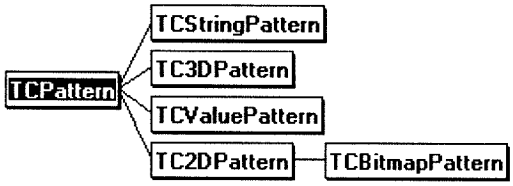

Fig. 6 - Ilustração esquemática dum exemplo de hierarquização das classes para os padrões de treino/teste.

Para finalizar, vamos resumir as principais consequências vantajosas oferecidas pela programação orientada a objectos, para este caso específico da implementação de sistemas neuronais complexos:

- **Modularidade**

Objectos bem concebidos são extremamente auto-contidos (não dependem de outros para funcionarem; são independentes), o que aumenta o nível de modularidade e de robustez, podendo estes ser reusados e estendidos na medida do necessário. Um objecto interage com o resto do programa duma forma muito restrita, isto é, o programa não deveria ter acesso aos dados dum objecto, a não ser através seus métodos. Além disso, os objectos devem-se apenas preocupar com os seus dados, sem ter de modificar variáveis globais exteriores.

Um objecto é como um circuito integrado (*chip*). Quando se quer construir uma aplicação, basta ligar alguns objectos diferentes de forma conveniente para obter um resultado altamente modular e estruturado. Isto leva a um conceito de controlo de avarias muito mais localizado e eficiente. Enquanto que em programas normais, o conserto dum erro de programação (*bug*) gerava mais dois ou três, na programação por objectos um erro não se propagará tão facilmente já que está confinado ao objecto em questão. Claro que isto só será verdade se os objectos comunicarem entre si duma forma estritamente necessária. Cada objecto até pode ser testado individualmente, sendo depois inserido no programa principal sem originar erros noutros objectos.

- **Segurança de utilização**

As mesmas características que tornam os objectos tão modulares, são responsáveis pelo seu elevado grau de confiança de utilização. Os objectos estão debilmente ligados uns aos outros, comunicando duma forma estritamente necessária. Portanto, o efeitos secundários de alterações a variáveis globais são minimizados, eliminando muitas oportunidades de aparecerem erros. Por causa dessa mínima comunicação, objectos que funcionam bem individualmente, têm elevadas probabilidades de funcionarem bem quando integrados no programa com outros objectos.

- **Reuso de código**

Durante a programação avançada dum projecto, poucas vezes se escreve uma rotina totalmente nova, limitando-se a fazer cópias modificadas para se adaptarem a cada situação (polimorfismo). Antes da programação por objectos, o reuso de código já feito era difícil e pouco modular. Mas com os objectos, este trabalho é grandemente facilitado. Para obter um comportamento duma rotina ligeiramente diferente para cada situação específica, basta criar um objecto descendente que implemente apenas uma actualização da parte do código que deverá ser diferente. Tudo o resto ficará igual.

- **Gravação e leitura através de *streams***

A hierarquização do sistema neuronal em objectos permite que se faça a transferência (leitura/escrita) de dados entre disco e memória, com base em *streams*, simplificando todo o código envolvido nesta operação.

Para permitir esta vantagem e simplicidade de se usarem *streams* para transferir um sistema neuronal entre disco e memória, na programação por objectos, cada classe envolvida tem de herdar a classe *TCStreamObject*. Conforme o compilador e a *runtime class library* usada, este nome pode variar.

Estas características são amplamente usadas em pacotes de ferramentas de programação que já incluem objectos-base, tais como a *ObjectWindows Library* [2] que contém objectos pré-fabricados para permitirem construir aplicações para o *Windows*. O que se pretende neste artigo é justamente dar os conceitos básicos de como se pode implementar um pacote de objectos-base (Ex: *ObjectNeural Library*), destinado à construção de sistemas neuronais de uma forma rápida, eficiente e segura, tal como no *ObjectWindows*.

Estes conceitos apresentados até aqui, foram resultado do estudo e construção de um simulador de redes neuronais, chamado de *NeuroCAD - Neural Network CAD-Simulator*, utilizado num projecto de reconhecimento automático de caracteres manuscritos[4], onde estavam envolvidas grandes quantidades de neurónios num sistema neuronal de elevada complexidade.

REFERÊNCIAS

[1] A. Porter, "C++ Programming for Windows", Osborne, McGraw Hill, 1993.

[2] Borland International, "ObjectWindows 2.0 programmer's guide", 1994.

[3] E. Schöneburg, N. Hansen, A. Gawelczyk, "Neuronale Netzwerke - Einführung, überblick und anwendungsmöglichkeiten", Markt & Technik Verlag AG, 1990.

[4] P. Kulzer, A. Branco, "Reconhecimento automático de caracteres", Publicação interna do DETUA, 1994.

[5] P.K. Simpson, "Foundations of Neural Networks", General Dynamics Electronics Division, .

[6] C.W. Lefebvre, J.C. Príncipe, "Object oriented artificial neural network implementation", Publicação interna.

[7] B. Stroustrup, "The C++ programming language", Addison-Wesley, 1985.