4
Desenvolvimento de uma Biblioteca de Redes Neurais de Alto Desempenho – ANNCOM

Nos capítulos anteriores foram apresentados alguns conceitos sobre o desenvolvimento de sistemas considerando questões de qualidade e de desempenho, e a utilização de placas gráficas para computação de propósito geral. Neste capítulo, será apresentada uma biblioteca de redes neurais que, ao levar em consideração as questões acima, alcançou resultados muito significativos com relação às ferramentas e sistemas existentes no mercado.

As redes neurais são muito úteis para a solução de problemas complexos de previsão ou aproximação de funções, e já existem no mercado, há algum tempo, bibliotecas que auxiliam a modelagem, a criação, o treinamento e o teste dos vários tipos de redes conhecidos. Entretanto, tais bibliotecas apresentam limitações quando o tamanho do problema ultrapassa determinado limite de complexidade, por exemplo, devido à alta quantidade de informação (registros ou atributos) contida na base de dados utilizada nas fases de treinamento, validação e teste.

Além do tamanho da base, existe também um equilíbrio entre a complexidade de cada iteração do algoritmo usado no treinamento e a melhora do acerto da rede a cada iteração, pois isto afeta diretamente o tempo total de treinamento. Por exemplo, se uma rede neural é treinada por um algoritmo de retro-propagação com gradiente comum, o custo de cada passo é relativamente pequeno, mas são necessários mais passos para se treinar a rede; por outro lado, um algoritmo quasi-newton requer muito menos iterações, mas com um custo computacional muito maior.

A biblioteca de redes neurais desenvolvida neste trabalho utiliza a capacidade computacional das placas de vídeo e uma estrutura diferenciada com o objetivo de reduzir o tempo de aprendizado, de forma a suportar problemas mais complexos e bases de dados maiores. A maior parte da ANNCOM foi escrita na linguagem C#, sobre a plataforma .NET 3.5, e o restante, em linguagem C estendida para CUDA.

4.1. Levantamento de Requisitos e Arquitetura

Para iniciar o desenvolvimento da biblioteca, analisaram-se as necessidades de aplicações típicas e fez-se um levantamento das ferramentas utilizadas na área de redes neurais artificiais. Das necessidades encontradas, as mais importantes foram:

- Ter um sistema modular para criação de redes neurais que possa ser utilizado diversas vezes, minimizando o retrabalho e facilitando extensões e trabalhos de manutenção;
- Treinar rapidamente muitas redes neurais com bases de dados com muitos registros e atributos.

Essas necessidades transformaram-se em requisitos não funcionais e levaram à escolha da plataforma .NET, que suporta linguagens orientadas a objetos e oferece flexibilidade e robustez, além de ser uma plataforma voltada a alta produtividade. Outro fator que influenciou na escolha da linguagem C# foi o suporte dado pela ferramenta de desenvolvimento da Microsoft, o Visual Studio.

Ao se fazer o levantamento das ferramentas da área de redes neurais, respeitaram-se os seguintes critérios: grau de utilização (quantos laboratórios ou empresas utilizam a ferramenta) e preço. Entre os softwares testados, o Matlab (The MathWorks, Inc., 2010) foi selecionado para testes de validação e de desempenho dos treinamentos das redes, e a MKL (Intel Corporation, 2010) foi utilizada para comparações de desempenho dos cálculos numéricos.

Além dos requisitos principais apresentados anteriormente, outros requisitos levantados foram:

- Requisitos Funcionais
  - Trabalhar com arquivos XML para carregar e salvar configurações de redes e resultados de treinamento;
  - Possibilitar a inserção dos componentes de redes neurais na caixa de ferramentas do Visual Studio, permitindo a criação de redes em tempo de desenvolvimento (*Design Time*), por meio de simples clique e arraste, além da configuração por meio de janelas;

- Suportar conexões com diversos tipos de bancos de dados;
- Possibilitar a compatibilidade com o Matlab por meio de funções que gerem vetores de pesos.

- Requisitos Não Funcionais
  - Apresentação de uma interface gráfica para modelagem, criação, treinamento, validação e teste de redes neurais e comitês de redes neurais (onde, por exemplo, várias redes votam em uma classe).

A partir das bases teóricas sobre arquitetura de software apresentadas no capítulo 2 e dos requisitos supracitados, pode-se obter uma visão geral do sistema, conforme mostra a Figura 18. Nessa figura, os componentes brancos (Elman, RBF e MLP) são instâncias do componente de redes neurais, chamado *NeuralNet;* os cinza-claros são instâncias do treinamento chamado *NetTrain*, e o componente *JacobianCUDA* é a parte do sistema desenvolvida utilizando-se a tecnologia da NVIDIA, com o objetivo de atender à restrição de desempenho do treinamento.

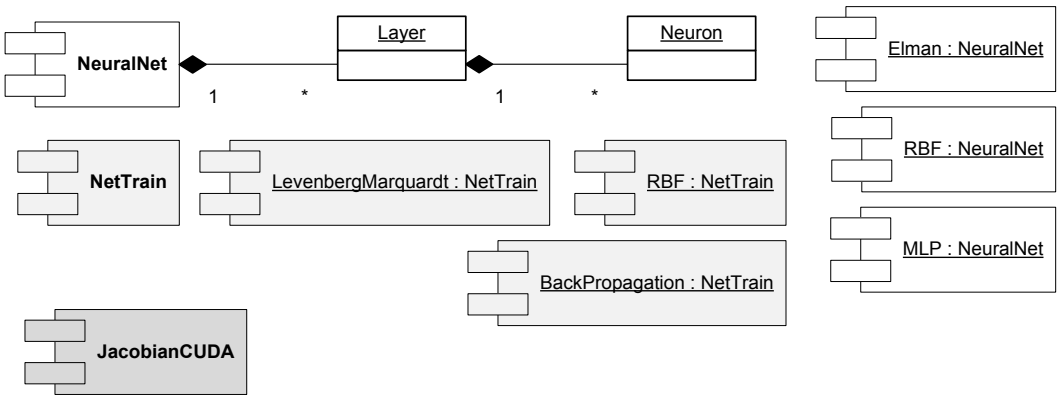

Figura 18 – Visão geral dos componentes principais da ANNCOM.

A arquitetura foi decomposta em componentes orientados a objetos de forma a suportar novas funcionalidades, como um novo algoritmo de treinamento, um novo tipo de rede neural, um comportamento diferente para os neurônios, etc. Essa característica não foi encontrada em nenhuma das ferramentas avaliadas Por exemplo, nelas não é possível propagar um sinal por uma rede MLP utilizando uma função de ativação diferente das já existentes, ou ajustar individualmente todos os parâmetros das gaussianas, inclusive o desvio padrão, dos neurônios no treinamento de redes radiais (o Matlab, por exemplo, utiliza um único valor de

desvio padrão para todos os neurônios da camada escondida). Na ANNCOM, para adicionar alguma funcionalidade nova, basta criar um novo componente derivado de um mais genérico (*NeuralNet*, *NetTrain*, etc.) e implementar a funcionalidade nova. Ao se criar um componente derivado, herda-se todas as funcionalidades e comportamento do componente pai, o que facilita e torna mais produtivo o desenvolvimento, uma vez que só é necessário implementar a parte adicional. No caso do Matlab, vale lembrar que é possível desenvolver funções novas, mas sem a facilidade de se herdar funcionalidades previamente desenvolvidas.

Outro detalhe da arquitetura é o suporte a conexões com vários tipos de gerenciadores de bancos de dados (SGBD – Sistema Gerenciador de Banco de Dados). Essa funcionalidade, apoiada por classes dedicadas à interface com bancos de dados disponíveis no arcabouço da Microsoft, permite que as redes neurais se conectem com diversos tipos de SGBD diferentes, facilitando a portabilidade e a implantação dos sistemas que utilizam a biblioteca, conforme mostra a Figura 19. Além disso, através de uma conexão direta entre a biblioteca e o SGBD, pode-se passar todo o processamento dos dados para o SGBD, de forma a respeitar políticas mais rígidas de manipulação de dados, muitas vezes adotadas pelos setores de TI (Tecnologia da Informação). Nesse caso, a TI fica responsável pela gerência dos procedimentos dentro do SGBD, enquanto a biblioteca apenas faz a leitura, sem ter a necessidade de processos para exportação e importação dos dados.

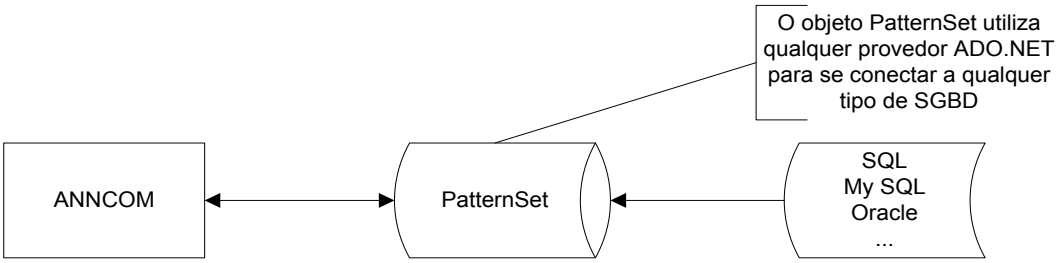

Figura 19 – Suporte a conexão com vários tipos de SGBD.

Para facilitar o desenvolvimento de novos sistemas com a ANNCOM, os componentes principais podem ser adicionados à caixa de ferramentas do Visual Studio (Microsoft Corporation, 2010). Com isso, no desenvolvimento de aplicações novas, o usuário só precisa clicar e arrastar novas redes ou treinamentos para o sistema que está sendo desenvolvido, e editá-los de forma visual e intuitiva. A Figura 20 ilustra a utilização dessa funcionalidade, apresentando, no lado direito,

a janela de ferramentas (*Toolbox*) com três componentes (*NeuralNet*, *GradientDescendent* e *LevenbergMarquardt*). Na parte de baixo da figura, encontram-se duas instâncias de componentes já adicionadas ao projeto (*neuralNet1* e *levenbergMarquardt1*); o componente de treinamento aparece selecionado. Na lateral esquerda da figura, estão listadas as propriedades do componente selecionado. Essas propriedades podem ser alteradas e verificadas em tempo real e, dessa forma, o desenvolvimento de um novo sistema se torna mais intuitivo, já que fica disponível na tela, toda a informação sobre o componente.

Como as redes neurais artificiais possuem uma estrutura com muitas coleções (uma rede possui um coleção de camadas, as camadas possuem uma coleção de neurônios e estes possuem uma coleção de sinapses), a ANNCOM também oferece um editor com janelas para facilitar a manipulação de tais estruturas, como mostra a Figura 21.

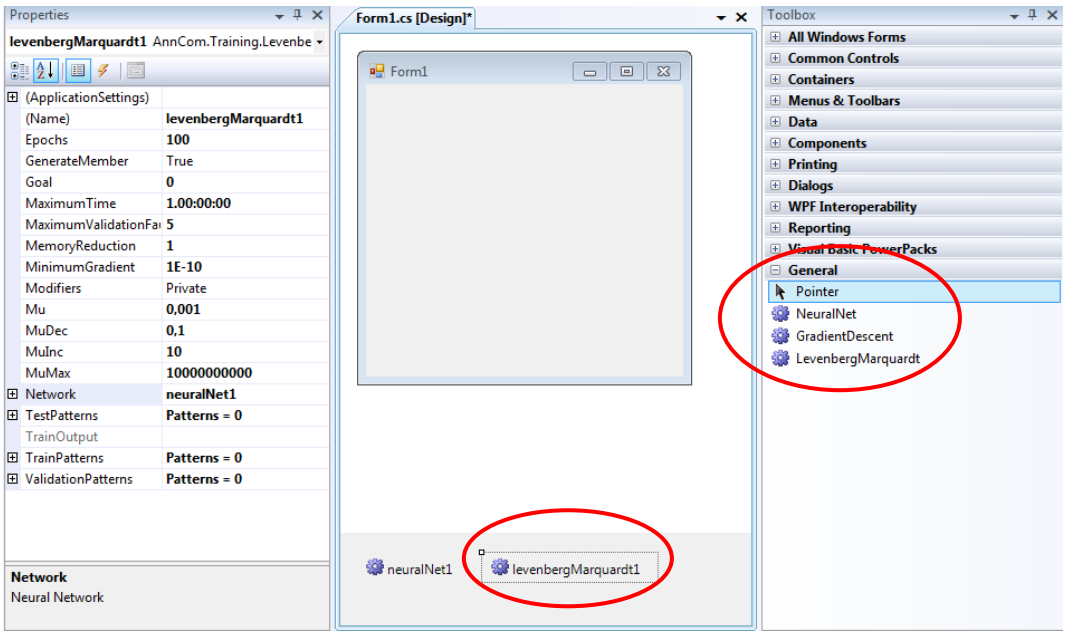

Figura 20 – Suporte da ANNCOM para tempo de execução no Visual Studio.

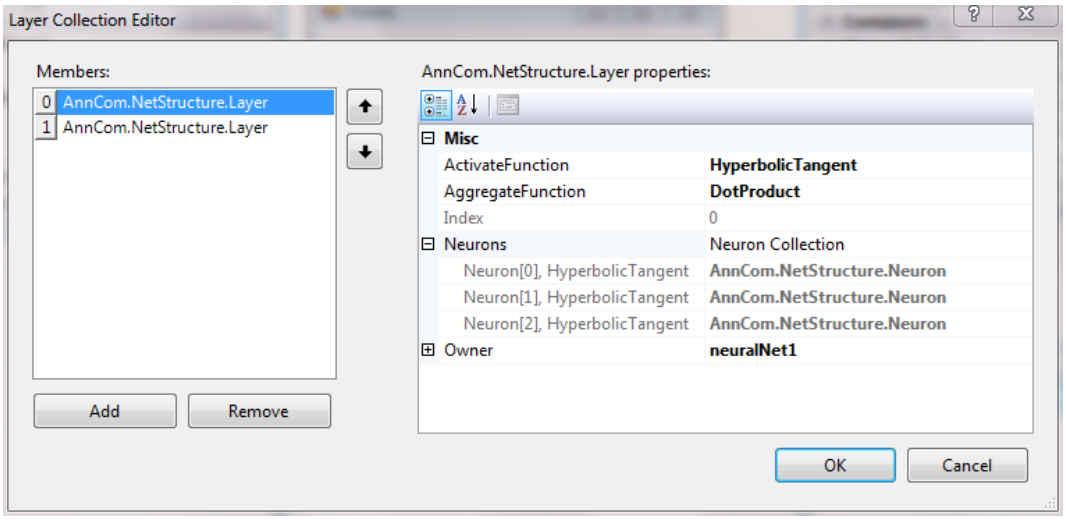

Figura 21 – Adição e edição de coleções na ANNCOM.

O requisito que trata do desempenho da biblioteca exigiu uma completa mudança de paradigma na implementação habitual de treinamento de uma rede neural. Neste trabalho, uma parte do treinamento foi modelada para ser executada em uma GPU com suporte a CUDA (Figura 22), levando-se em consideração a Lei de Amdahl (seção 3.1.1). Outras partes da biblioteca também incorporaram inovações para que o tempo de execução fosse reduzido. Essas inovações incluem o uso de *threads*, sistemas com vários processadores e vários computadores. A seguir, serão descritos em maiores detalhes os dois componentes principais da biblioteca: o *NeuralNet* e o *NetTrain*. Além disso, será apresentada uma ferramenta gráfica para auxiliar a criação de soluções envolvendo redes neurais.

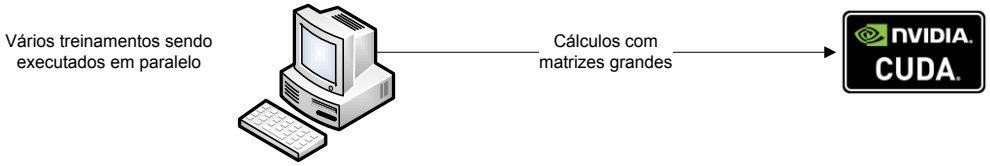

Figura 22 – Visão geral do desvio de parte do treinamento para GPU.

4.2. Estrutura Básica

Para facilitar a leitura e a organização do código, a biblioteca foi dividida em partes que foram definidas como Espaços de Nomes, conforme mostra a Figura 23. Além dos espaços principais, de Redes Neurais (*NetStructure*), de Calculadores de Erros (*NetError*) e de Treinamento (*Training*), que serão descritos nas

seções seguintes, existem o Espaço de Modelos (*Templates*), onde ficam os modelos de redes neurais implementados, o Espaço de Funções de Ativação (*ActivateFunctions*), cujo conteúdo são as funções de ativação das redes neurais e o Espaço de Saídas (*NetOutput*). Esse último também será descrito em uma próxima seção.

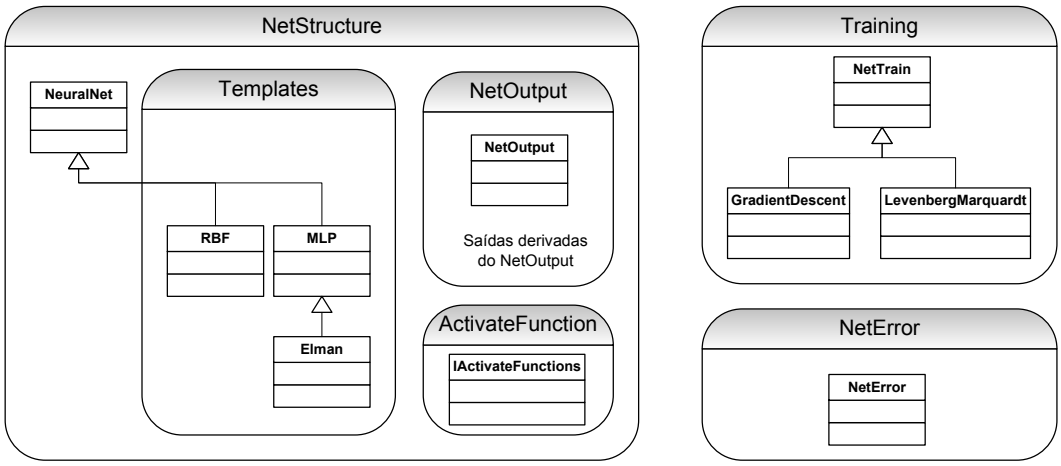

Figura 23 – Esquema com todos os espaços de nomes da biblioteca ANNCOM.

Além dessas partes, foram desenvolvidas outras secundárias que dão suporte às conexões com bancos de dados e cálculos de erro. Para trabalhar com bancos de dados e arquivos de texto ou XML, a ANNCOM possui uma estrutura chamada *PatternSet*, que armazena e manipula as entradas e alvos para treinamento, embaralha-os, transforma-os em matrizes ou divide-os em subgrupos para treinamentos com validação cruzada.

4.2.1. Componente NeuralNet

Com o objetivo de facilitar a explicação sobre o componente *NeuralNet*, a Figura 24 ilustra uma rede neural *Multi Layer Perceptron* (MLP). Nela, as camadas destacadas em azul e cinza possuem coleções de neurônios, os quais, possuem uma coleção de sinapses própria. O processamento dos padrões de entrada, oriundos, no caso da Figura 24, do banco de dados (BD), é feito propagando-os através da rede, que coloca o resultado no neurônio, ou nos neurônios, da última camada.

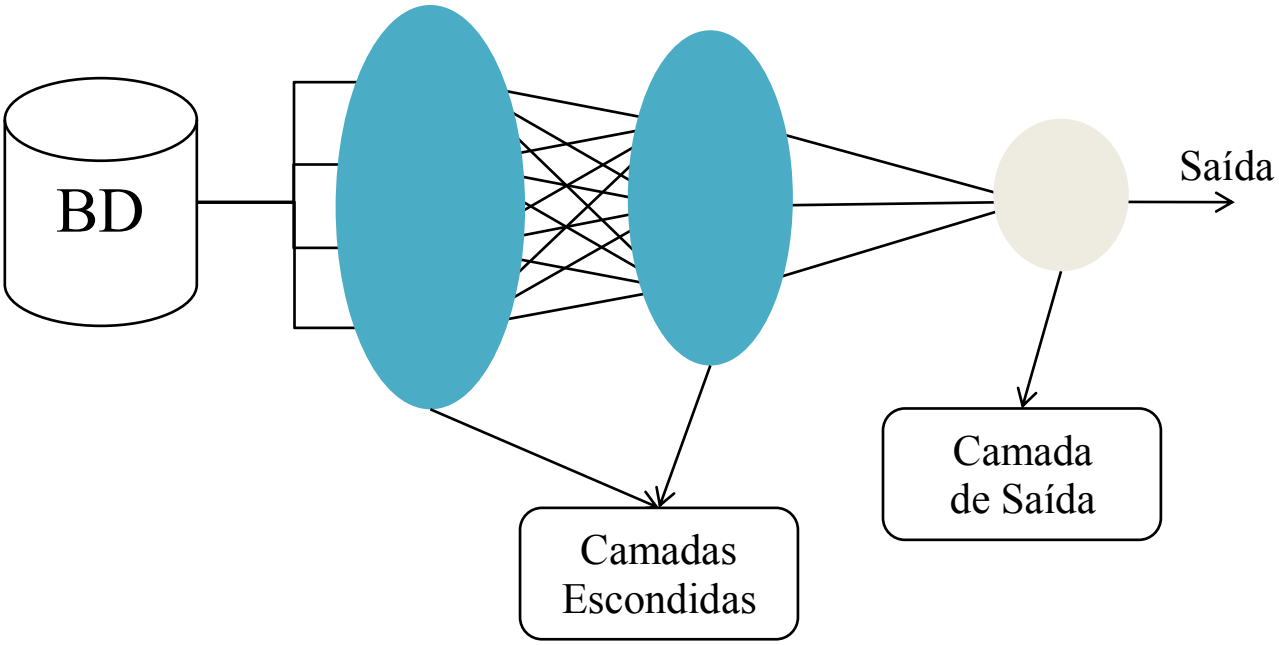

Figura 24 – Representação de uma rede neural MLP.

O componente *NeuralNet* representa uma rede neural artificial completa, incluindo todas as camadas com os neurônios e sendo o responsável pela propagação dos sinais de entrada pela rede. Fica também a cargo desse componente o dever de manter um estado consistente em toda a estrutura da rede neural. Neste trabalho, uma rede neural é considerada consistente quando todos os seus neurônios possuem um valor de saída referente ao último sinal propagado por ele. Como o *NeuralNet* é uma representação genérica de uma rede, todos os modelos desenvolvidos são uma especialização dele e, assim como o *NetError*, que será descrito mais adiante, podem-se criar novos modelos simplesmente herdando do componente pai, nesse caso, o *NeuralNet*.

A primeira inovação da ANNCOM é a estrutura do *NeuralNet* que é composta por vários objetos menores, conforme apresentado no padrão de projeto Composição (Figura 3). Nessa estrutura, a rede neural é formada por uma coleção de camadas que, por sua vez, é formada por uma coleção de neurônios, e estes possuem uma coleção de sinapses. Assim sendo, os componentes da biblioteca se apresentam de forma mais intuitiva, pois as partes do modelo teórico se transformaram em objetos ou componentes independentes, que podem ser especializados, modificados ou atualizados, sem afetar o resto da estrutura.

Além de uma estrutura mais intuitiva, o modelo estrutural da biblioteca permite uma gerência de memória melhor, mesmo custando mais espaço, pois as

informações que a rede possui ficam distribuídas em vários objetos que são gerenciados pelo componente *NeuralNet*. Normalmente, uma rede neural é tratada como uma cadeia de vetores ou várias matrizes, o que pode ser mais simples de implementar, mas que pode gerar blocos contínuos suficientemente grandes de memória a ponto de serem paginados para o disco rígido, implicando em um desempenho menor no treinamento.

4.2.2. Modelos de Redes Neurais Artificiais

Para facilitar o uso da biblioteca, alguns dos modelos mais utilizados de redes neurais foram implementados e são indicados na Figura 25. Esses objetos são especializações do componente mais genérico e possuem algumas restrições de funcionalidades ou comportamentos para que o modelo teórico da rede não seja desrespeitado. Por exemplo, no modelo *Elman*, Figura 26, onde $x(k)$ = entrada da rede, $a_n(k)$ = saída dos neurônios, $z^{-1}(k)$ = vetor de estados e $y(k)$ = saída da rede, os neurônios da camada escondida estão ligados aos neurônios da camada de entrada e, ao se criar uma rede a partir de uma instância do objeto *Elman* da biblioteca ANNCOM, essa ligação será feita de forma automática.

No caso das redes radiais (RBF), ao se criar uma nova rede, deve-se ajustar os parâmetros das funções de ativação (gaussiana) dos neurônios que são criados a partir dos padrões de treinamento. Esse processo também é executado automaticamente ao se criar um objeto RBF da ANNCOM.

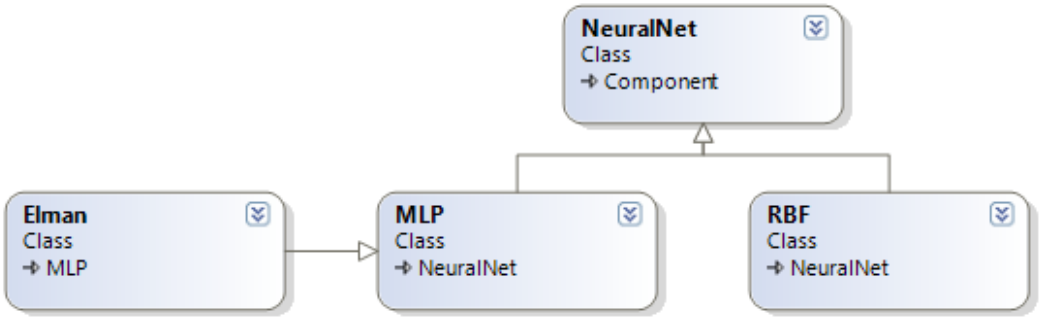

Figura 25 – Diagrama de classes simplificado dos modelos de redes neurais implementados nesse trabalho.

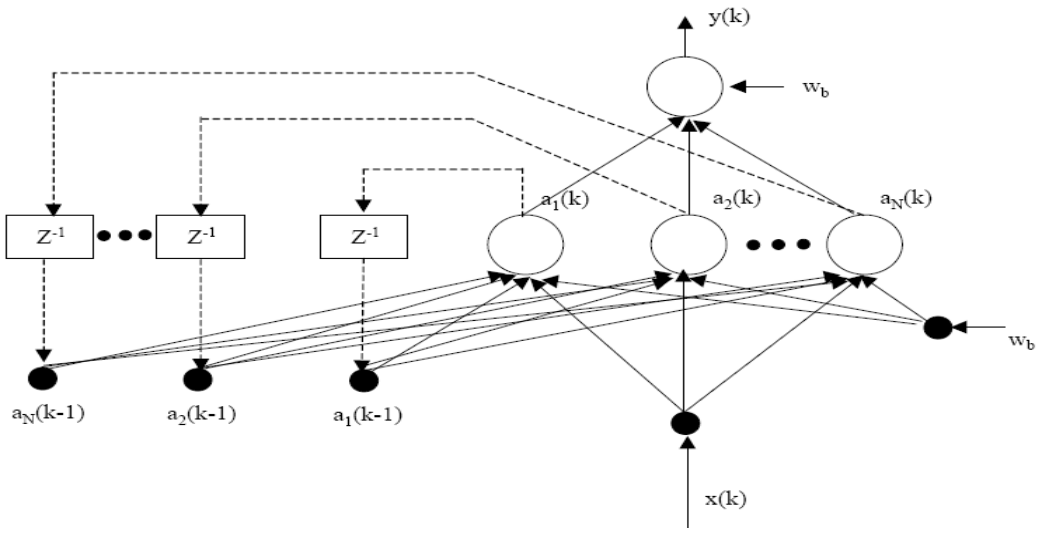

Figura 26 – Modelo de uma rede Elman.

As redes MLP são as mais comuns e sua estrutura já foi apresentada na Figura 24. Para esse tipo de rede, a biblioteca ANNCOM impede o uso de algumas funções de ativação e funções de agregação, e conecta automaticamente os neurônios, de modo que o sinal sempre siga na direção da saída, pois isso é uma característica de tipo de rede, também denominado MLP *Feed-forward Only*.

4.2.3. Estrutura NetOutput

Para decodificação dos resultados, a ANNCOM possui um objeto chamado *NetOutput*. Essa decodificação converte as saídas dos neurônios da última camada da rede para um formato compatível com o problema. Por exemplo, se uma rede neural está sendo usada para avaliar se um cliente pode ou não pegar um empréstimo bancário, o *NetOutput* deve transformar as saídas dos neurônios em "sim" ou "não". Desse objeto derivam vários outros mais específicos, como saídas Numérica, de Classificação e outras, seguindo o padrão de projeto Estratégia. Além disso, como a ANNCOM é orientada a objetos, podem-se criar novos decodificadores herdeiros do *NetOutput*. A Figura 27 apresenta um diagrama de classes resumido com a estrutura já implementada na biblioteca.

O tipo *Approximation* apenas desnormaliza os sinais, caso eles tenham sido normalizados antes de serem apresentados à rede. Esse tipo de saída é normalmente usado em previsões de séries e aproximações de funções. O tipo *Classification* é utilizado em problemas como classificação de clientes, apoio a decisões e etc.

Na ANNCOM, o tipo *Classification* foi derivado em mais três classes, com nomes auto-explicativos para especialistas na área de redes neurais: *One of N*, *Thermometer*, *Binary*. O tipo *Clustering* foi criado para agrupamento de subconjuntos. Esse tipo de saída se adéqua a problemas como mineração de dados ou problemas de classificação onde as classes ainda não são conhecidas.

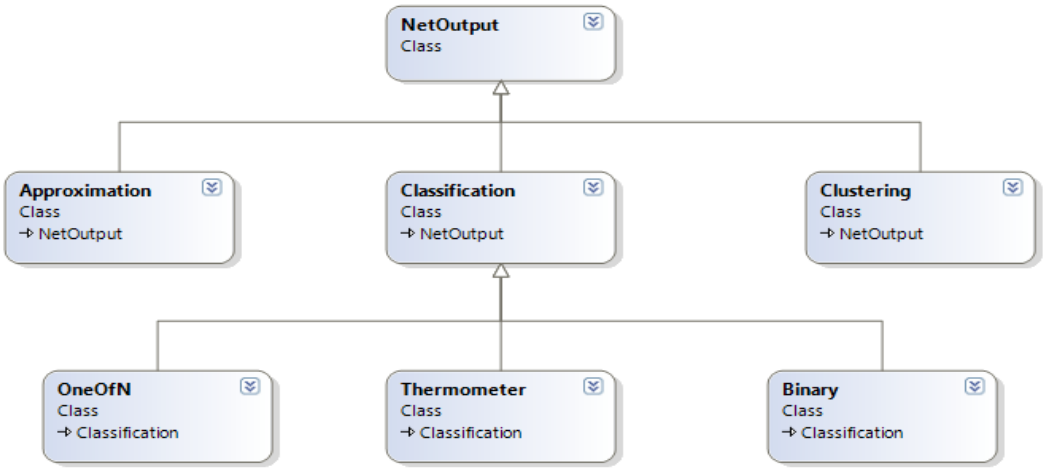

Figura 27 – Diagrama de classes com as estruturas de decodificação das saídas das redes neurais.

4.2.4. Estruturas para Cálculo de Erro

Para fornecer suporte ao cômputo de erros, a biblioteca possui uma estrutura (*NetError*) com diversos tipos de cálculos conhecidos, como Erro Médio Quadrático, Erro Absoluto Médio, U de *Theil*, entre outros. Além disso, é possível criar novos objetos para cálculo de erro, simplesmente herdando do objeto *NetError*. A Figura 28 apresenta um diagrama de classes simplificado, com os objetos para cálculo de erro implementados na ANNCOM.

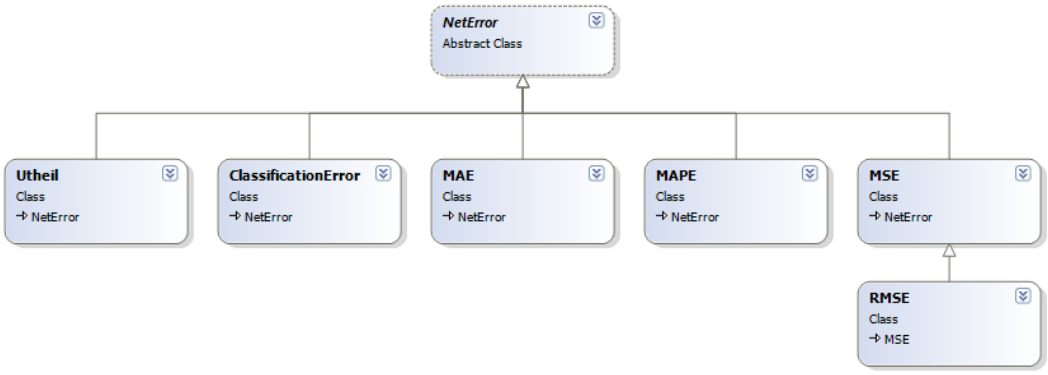

Figura 28 – Diagrama de classes com os objetos para cálculo de erro, implementados na ANNCOM.

A métrica U de Theil serve para determinar a acerácea de uma previsão (P_i) comparando o conjunto previsto com os valores reais (A_i), conforme a Equação 4. Caso a previsão seja perfeita, o resultado será zero, e se um modelo naïve, onde a previsão é igual à última realização da série, for utilizado, o resultado será um. Dessa forma, modelos melhores darão resultados entre zero e um.

$$U = \frac{\sqrt{\sum (P_i - A_i)^2}}{\sqrt{\sum {A_i}^2}} \qquad \text{Equação 4}$$

O objeto chamado *ClassificationError* fornece um resultado percentual dos erros de classificação a partir de uma matriz de confusão como a da Equação 5, onde C_1 é o acerto da rede no grupo um, C_0 é o acerto da rede no grupo zero, F_1 é o erro da rede no grupo um e F_0 é o erro da rede no grupo zero. A partir dessa matriz, pode-se calcular a sensibilidade (S) (Equação 6) e a especificidade (E) (Equação 7).

$$\begin{bmatrix} C_1 & C_0 \\ F_1 & F_0 \end{bmatrix} \qquad \text{Equação 5}$$

$$S = \frac{C_1}{C_0 + F_1} \qquad \text{Equação 6}$$

$$E = \frac{F_1}{C_0 + F_1} \qquad \text{Equação 7}$$

A métrica MAE (*Mean Absolute Error* – Erro Absoluto Médio) é como o nome sugere uma média dos erros absolutos da previsão e é dado pela Equação 8, onde *e* representa o erro da previsão. Essa métrica é usada em previsões de séries e aproximações de funções, assim como a métrica MAPE (*Mean Absolute Percentage Error* – Erro Médio Percentual Absoluto). A MAPE é dada pela Equação 9, onde A é a série real e F é a série prevista. Essa métrica apresenta alguns problemas práticos como, por exemplo, ocorrências de divisões por zero quando existem zeros na série real.

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |e_i| \qquad \text{Equação 8}$$

$$M = \frac{1}{n}\sum_{t=1}^{n}\left|\frac{A_t - F_t}{A_t}\right| \qquad \text{Equação 9}$$

As métricas MSE (*Mean Squared Error* – Erro Médio Quadrático) e sua derivação chamada RMSE (*Root Mean Squared Error* – Raiz do Erro Médio Quadrático) são definidas, respectivamente, na Equação 10 e na Equação 11. Como o MSE é o momento de segunda ordem do erro, incorpora tanto a variância do estimador quanto a sua parcialidade (BERGER e CASELLA, 2001). Para um estimador não viesado, isto é, quando a diferença entre o valor esperado e o valor estimado é zero, o MSE é a variância e ambos têm a mesma unidade de medida. Em uma analogia ao desvio-padrão, tomando a raiz quadrada do MSE produz-se o RMSE, que tem as mesmas unidades que a quantidade estimada por um estimador imparcial. O RMSE é a raiz quadrada da variância, conhecida como o erro padrão (BERGER e CASELLA, 2001).

$$MSE = \frac{1}{n}\sum_{i=1}^{n} {e_i}^2 \qquad \text{Equação 10}$$

$$RMSE = \sqrt{MSE} \qquad \text{Equação 11}$$

4.3. Modelo de Treinamento

Antes de apresentar o modelo de treinamento da biblioteca ANNCOM, serão descritos os dois principais algoritmos de treinamento de redes neurais, ambos envolvendo o cálculo do gradiente. O primeiro deles é chamado Gradiente Decrescente, e é um algoritmo de otimização de primeira ordem, que utiliza o negativo do gradiente para encontrar o mínimo local de uma função. Conforme mostra a Figura 29, cada passo do algoritmo tem a direção do vetor gradiente.

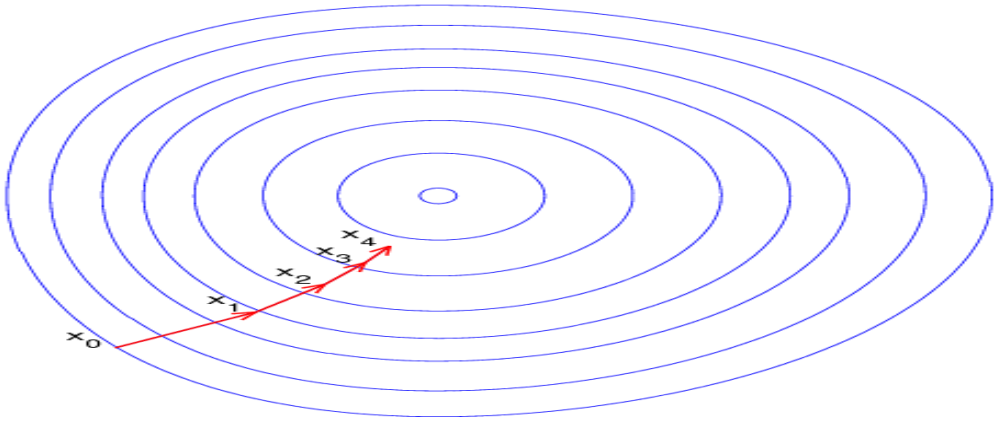

Figura 29 – Ilustração de uma descida por gradiente (WIKIPEDIA, 2010).

Para treinar as redes neurais, o objeto *GradientDescent* propaga, a cada iteração ou passo, todos os padrões de entrada da rede (modo *batch*), calculando o erro dos neurônios da última camada para cada padrão (Equação 12, onde t é o valor real, y é a saída da rede, p é o índice do padrão e i é o índice dos neurônios da última camada da rede).

$$E_i^p = t_i^p - y_i^p \qquad \text{Equação 12}$$

Com isso encontra-se o erro total (Equação 13) que será usado para encontrar o MSE da rede neural (Equação 14, onde n é o número de padrões e k é o número de neurônios na última camada da rede).

$$E_{SSE} = \sum_p \sum_i \frac{1}{2}\left(t_i - y_i\right)^2. \qquad \text{Equação 13}$$

$$e_k = \frac{1}{n} E_{SSE} \qquad \text{Equação 14}$$

Esse erro é propagado para as camadas anteriores levando-se em conta os valores dos pesos (w) das sinapses (Equação 15, onde w_{ij} é o peso que liga o neurônio j ao neurônio i).

$$\frac{\partial e}{\partial w_{ij}} = \sum_p \frac{\partial e_p}{\partial w_{ij}} = \sum_k \frac{\partial e_p}{\partial a_k}\frac{\partial a_k}{\partial w_{ij}} \qquad \text{Equação 15}$$

A partir daí, mede-se a contribuição delta (δ_i) de cada padrão ao somatório a_i (Equação 16, onde a_i é a soma ponderada das entradas para o neurônio i).

$$\delta_i = \frac{\partial e_p}{\partial a_i} = \frac{\partial e_p}{\partial y_i}\frac{\partial y_i}{\partial a_i} \qquad \text{Equação 16}$$

O primeiro termo da Equação 16 é obtido da Equação 18 (válido apenas para SSE – *Sum Squared Error*) e o segundo termo da Equação 19, resultando na Equação 17.

$$\delta_{i\_saída} = -(t_{pi} - y_{pi})f' \quad \text{Equação 17}$$

$$\frac{\partial e_p}{\partial y_i} = -(t_{pi} - y_{pi}) \quad \text{Equação 18}$$

$$\frac{\partial y_i}{\partial a_i} = f'(a_i) \quad \text{Equação 19}$$

Na Equação 19, a função *f'* é a derivada da função de ativação de cada neurônio. Considerando que os neurônios influenciam o erro pelo efeito deles sobre os neurônios das camadas subseqüentes, o cálculo do delta (Equação 20, onde *w* é o peso de uma sinapse) para os neurônios nas camadas escondidas é feito de forma indireta (REED e MARKS, 1999).

$$\delta_i = \frac{\partial e_p}{\partial a_i} = \sum_k \frac{\partial e_p}{\partial a_k}\frac{\partial a_k}{\partial a_i} = \sum_k \delta_k \frac{\partial a_k}{\partial a_i} = -f'_i \sum_k \delta_k w_{ki} \quad \text{Equação 20}$$

Após calcular os deltas de todos os neurônios, a atualização dos pesos é feita a partir da Equação 21, não utilizando momento (HAYKIN, 2007), ou da Equação 22, utilizando momento (HAYKIN, 2007). Nessas equações, o η representa a taxa de aprendizado (HAYKIN, 2007).

$$\Delta w_{ij} = -\eta \frac{\partial e_i}{\partial w_{ij}} \quad \text{Equação 21}$$

$$\Delta w_{ij}{}^t = -\eta \frac{\partial e_i}{\partial w_{ij}} + \alpha \Delta w_{ij}{}^{t-1} \quad \text{Equação 22}$$

O segundo algoritmo de treinamento usado na ANNCOM é baseado no algoritmo LMA – *Levenberg-Marquardt Algorithm* (MARQUARDT, 1963) para mínimos quadráticos não lineares, que foi incorporado à retro-propagação para treinamento de redes neurais em (HAGAN e MENHAJ, 1994). O método é uma aproximação do método de Newton, que é descrito na Equação 23 e tem o objetivo de minimizar o erro $E_{SSE} = E(\underline{x})$ da Equação 13, com relação ao vetor $\underline{x}$, que representa o vetor de pesos da rede neural.

$$\Delta x = -[\nabla^2 \mathrm{E}(\underline{\mathrm{x}})]^{-1} \nabla E(\underline{\mathrm{x}}) \quad \text{Equação 23}$$

Na Equação 23, $\nabla^2 \mathrm{E}(\underline{\mathrm{x}})$ é a chamada matriz Hessiana (Equação 24) e $\nabla E(\underline{\mathrm{x}})$ é o gradiente.

$$\mathrm{H}(\underline{x}) = \nabla^2 \mathrm{E}(\underline{x}) = \begin{bmatrix} \frac{\partial^2 E(\underline{x})}{\partial x_1^2} & \cdots & \frac{\partial^2 E(\underline{x})}{\partial x_1 x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 E(\underline{x})}{\partial x_n x_1} & \cdots & \frac{\partial^2 E(\underline{x})}{\partial x_n^2} \end{bmatrix} \quad \text{Equação 24}$$

Entretanto, o custo computacional do método de Newton pode ser muito alto, por causa da complexidade de se calcular a matriz Hessiana e sua inversa, pelo fato que essa matriz pode ser singular. De forma a contornar esses dois problemas, o LMA, mesmo não possuindo uma convergência tão rápida quanto o método de Newton, possui iterações computacionalmente mais baratas e um ajuste para minimizar a ocorrência de matrizes singulares.

Para simplificar o método de Newton, o LMA assume que $E(\underline{x})$ é uma soma de quadrados na forma da Equação 25, o que possibilita utilizar o método de Gauss-Newton (MARQUARDT, 1963) e, com isso, reescrever as duas componentes da Equação 23 na forma da Equação 26, onde *J* é a matriz Jacobiana (Equação 27) e *S* é dado pela Equação 28.

$$E(x) = \sum_{i=1}^{N} e_i^2(\underline{x}) \quad \text{Equação 25}$$

$$\begin{gathered} \nabla E(\underline{x}) = J^T(\underline{x})\underline{e}(\underline{x}) \\ \nabla^2 \mathrm{E}(\underline{x}) = J(\underline{x})J^T(\underline{x}) + S(\underline{x}) \end{gathered} \quad \text{Equação 26}$$

$$J(x) = \begin{bmatrix} \frac{\partial e_1(\underline{x})}{\partial x_1} & \cdots & \frac{\partial e_1(\underline{x})}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial e_N(\underline{x})}{\partial x_1} & \cdots & \frac{\partial e_N(\underline{x})}{\partial x_n} \end{bmatrix} \quad \text{Equação 27}$$

$$S(\underline{x}) = \sum_{i=1}^{N} e_i(\underline{x}) \nabla^2 e_i(\underline{x}) \quad \text{Equação 28}$$

Como o método de Gauss-Newton assume que $S(\underline{x}) \approx 0$, a Equação 23 pode ser reescrita na forma da Equação 29.

$$\Delta x = -\left[J(\underline{x})J^T(\underline{x})\right]^{-1} J^T(\underline{x})\underline{e}(\underline{x}) \quad \text{Equação 29}$$

Para minimizar a ocorrência de matrizes singulares, o algoritmo LM modifica a Equação 29 obtendo a Equação 30 (MARQUARDT, 1963), onde *I* é a matriz identidade e μ é um parâmetro que é multiplicado, a cada iteração, por um fator β enquanto a função $E(\underline{x})$ é minimizada.

$$\Delta x = -\left[J(\underline{x})J^T(\underline{x}) + \mu I\right]^{-1} J^T(\underline{x})\underline{e}(\underline{x}) \quad \text{Equação 30}$$

Caso o valor da função de erro aumente, o parâmetro μ é dividido pelo fator β. É importante perceber que, quando μ é muito grande, tem-se uma maior convergência com um passo de $1/\mu$, e quando μ é muito pequeno, o algoritmo tende ao método de Gauss-Newton.

O ponto mais importante para incorporação desse algoritmo ao treinamento é o cálculo da matriz Jacobiana a partir dos pesos da rede, por meio de pequenas modificações no algoritmo do gradiente decrescente já apresentado. Lembrando que a função de erro é dada pela Equação 31 (onde $\underline{a}_p^n$ é saída da rede neural para o padrão p), no caso da retro-propagação padrão, a derivada é dada pela Equação 32, onde k é o índice dos neurônios da última camada da rede, que possui um total de K neurônios, e l é o índice da camada.

$$E = \frac{1}{2}\sum_{p=1}^{P}\left(\underline{t}_p - \underline{a}_p^n\right)^T\left(\underline{t}_p - \underline{a}_p^n\right) = \frac{1}{2}\sum_{p=1}^{P}\underline{e}_p^T\underline{e}_p \qquad \text{Equação 31}$$

$$\frac{\partial \hat{E}}{\partial w^l(i,j)} = \frac{\partial \sum_{k=1}^{K} e_{p(k)}^2}{\partial w^l(i,j)} \qquad \text{Equação 32}$$

Para os elementos da matriz Jacobiana, é necessário calcular a Equação 33. Lembrando que o cálculo da Equação 33 para a última camada no processo de retro-propagação do gradiente decrescente é dado pela Equação 17, esses elementos são calculados com a modificação dada pela Equação 34 na última camada.

$$\frac{\partial e_p(k)}{\partial w^l(i,j)} \qquad \text{Equação 33}$$

$$\Delta^k = -f^k \qquad \text{Equação 34}$$

Na ANNCOM, utiliza-se uma abordagem inovadora para se treinar redes neurais. Na maioria dos programas avaliados, as redes neurais apontam para o algoritmo de treinamento que será usado para treiná-la. Na biblioteca deste trabalho, concebeu-se um modelo de treinador e aluno. A principal diferença entre as duas abordagens se evidencia ao se modelar um sistema envolvendo muitas redes neurais: a nova abordagem torna a gerência dos treinamentos mais intuitiva, já que podem ser criadas "salas de aula" com redes neurais, como mostra a Figura 30. Nessa figura, as salas de aula representam treinadores com diferentes configurações, que são responsáveis por treinar todos os seus alunos, no caso, as redes neurais. Por exemplo, considere que a sala A representa um treinamento usando o LMA e as redes 1, 2 e 3 são redes MLP com diferentes números de camadas e neurônios. Nesse caso, o treinador A deve treinar, validar e testar as redes 1, 2 e 3

usando suas próprias configurações. O treinador B, por sua vez, treina, valida e testa as redes 4 e 5, usando outras configurações e até outro algoritmo de treinamento.

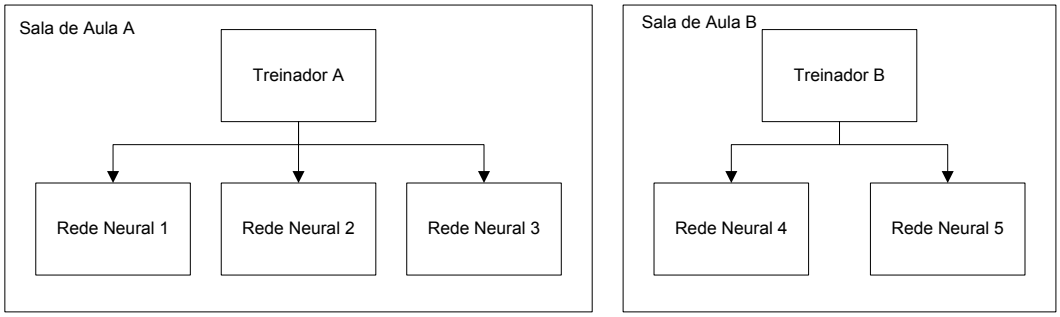

Figura 30 – Exemplo de duas "salas de aula" onde várias redes são treinadas.

Além desse novo modelo para treinamento, foram adicionadas algumas funções para aumentar a produtividade dos programadores que fizerem uso da biblioteca, como por exemplo, o treinamento com validação cruzada automática, que faz várias divisões na base de entrada, embaralhando os padrões e selecionando a rede que melhor atende aos critérios de avaliação do sistema em questão – **Error! Reference source not found.**. Nessa figura, os dados da base de entrada são lidos pelo treinador, que os embaralha de modo que, ao dividir a base misturada, gere novos grupos de treinamento, validação e teste. Depois de embaralhar e dividir a base de entrada, o treinador inicia o treinamento nas suas redes alunas. Esse processo pode ser repetido diversas vezes e serve para maximizar as chances de sucesso no treinamento (HAYKIN, 2007).

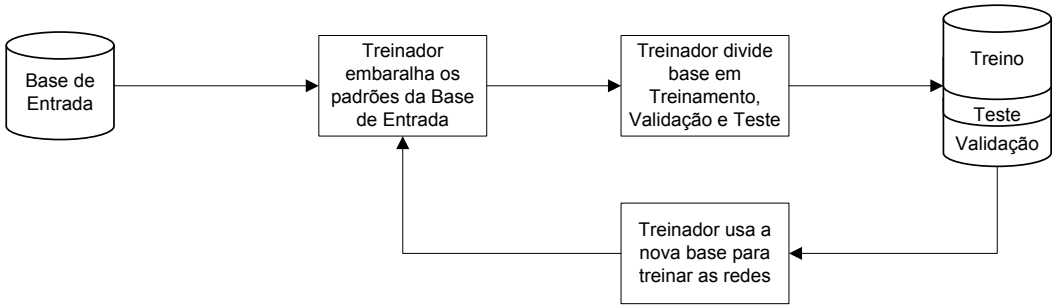

Figura 31 – Validação cruzada automática da ANNCOM.

Para avaliar os resultados com diferentes critérios de erro, o *NetTrain* possui um gerenciador para cálculos de erro, que é responsável por calcular os erros refe-

rentes aos padrões da base de validação ou teste – Figura 32. Nessa figura, o retângulo cinza é o início do processo com os padrões sendo propagados através da rede neural. Após a propagação, o treinador invoca o gerenciador de erro para executar os cálculos necessários. Por sua vez, o gerenciador de erro, executa os cálculos, salva os resultados e devolve o controle ao treinador. Caso não tenha que executar nenhum cálculo, informa ao treinador para continuar. Por exemplo, se um sistema treina uma rede neural utilizando, para teste, as métricas Erro Percentual Absoluto Quadrático e U de Theil, a saída desse treinamento terá dois vetores com os resultados dos cálculos de erro de teste de cada época, além das informações padrões, como tempo decorrido, erro do treinamento e status final.

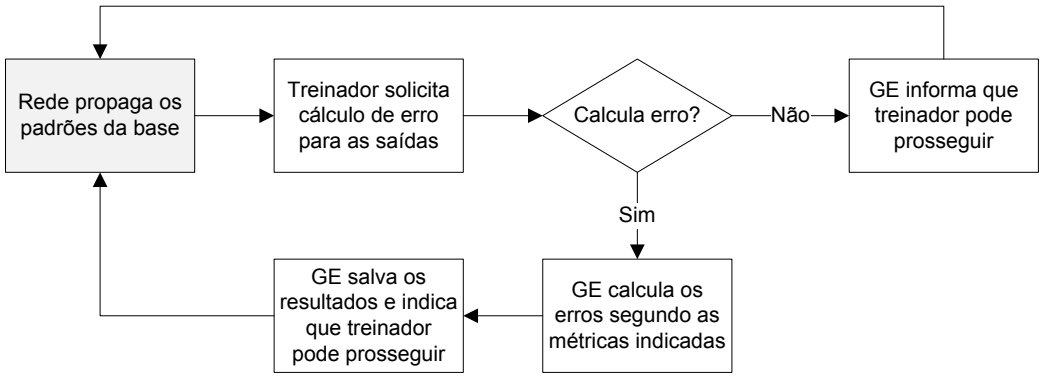

Figura 32 – Exemplo de funcionamento do Gerenciador de Erro (GE) para cálculo de erros de validação e teste durante o treinamento.

Além das funcionalidades já apresentadas, o *NetTrain* oferece dois tipos de inicialização de pesos para as redes neurais: uma aleatória e baseada em (REED e MARKS, 1999) e outra baseada em (NGUYEN e WIDROW, 1990) usada no Matlab. Com o objetivo de melhorar o desempenho da biblioteca, o treinamento pode ser feito utilizando vários processadores, através do recurso de *threads*. A parte do código paralelizada é responsável pelo cálculo da matriz Jacobiana quadrada e da variação dos pesos a partir dessa matriz, conforme Equação 30 (HAGAN e MENHAJ, 1994). Tal cálculo representa mais de 70% do tempo de processamento para uma base de tamanho pequeno e pode chegar a quase 90% em bases maiores.

4.3.1.
O Treinamento em GPGPU

Considerando que a Equação 30 é executada uma vez a cada época do treinamento e que nela a matriz *J* possui tamanho (w, p), onde *w* é o número de pesos da rede e *p* é o número de padrões, mesmo utilizando vários núcleos, o tempo total de um treinamento pode não ser satisfatório, uma vez que a matriz, pode conter milhões de elementos. Na verdade, esse tempo pode chegar a 90% do tempo total, conforme apresentado anteriormente.

A utilização de placas gráficas se mostrou promissora, pois o número de núcleos de processamento em uma única placa pode chegar a centenas. Além disso, o tempo para se gerenciar *threads* na parte C# da ANNCOM pode custar mais do que 10% do tempo total de treinamento, enquanto que, conforme indicado no capítulo 3, a arquitetura CUDA pode tratar *threads* sem custo adicional. Outro ponto positivo, percebido ao se utilizar a tecnologia da NVIDIA, foi a possibilidade de se trabalhar com várias placas gráficas, permitindo a execução de vários treinamentos simultâneos.

A Figura 33 apresenta uma visão geral do treinamento e o fluxo do programa através do *host* e do *device*. Nessa figura, a Equação 30 é enviada para a GPU, de forma que o vetor com as variações dos pesos da rede fosse obtido diretamente na placa gráfica e a partir da matriz Jacobiana, dessa forma, a placa de vídeo é responsável por calcular toda a Equação 30, transferindo para a memória RAM da CPU apenas o resultado de Δx..

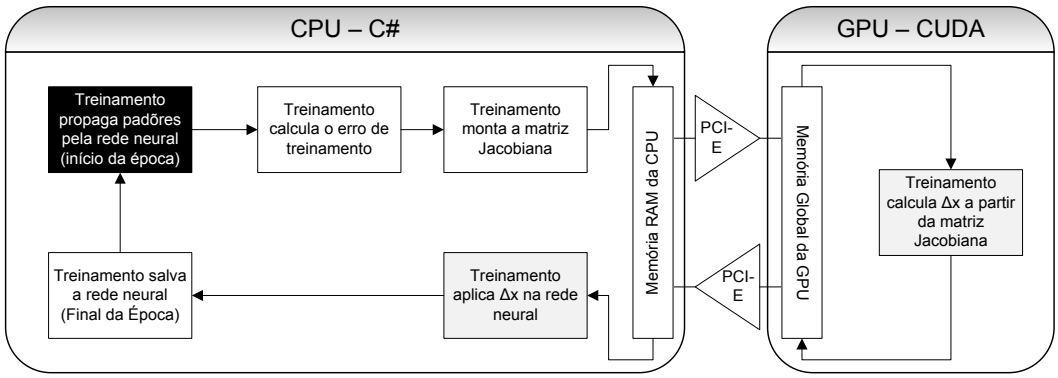

Figura 33 – Segundo modelo (resumido) proposto, que executa uma porção maior do código na placa gráfica.

4.3.2.
Implementação na GPU

A implementação da Equação 30 na GPU foi dividida em duas partes (*kernels*) principais: a primeira parte é responsável pela multiplicação matricial das matrizes e a segunda responsável por calcular a inversão matricial de um dos resultados da primeira parte. A Figura 34 mostra como foi dividido o processamento em *kernels*.

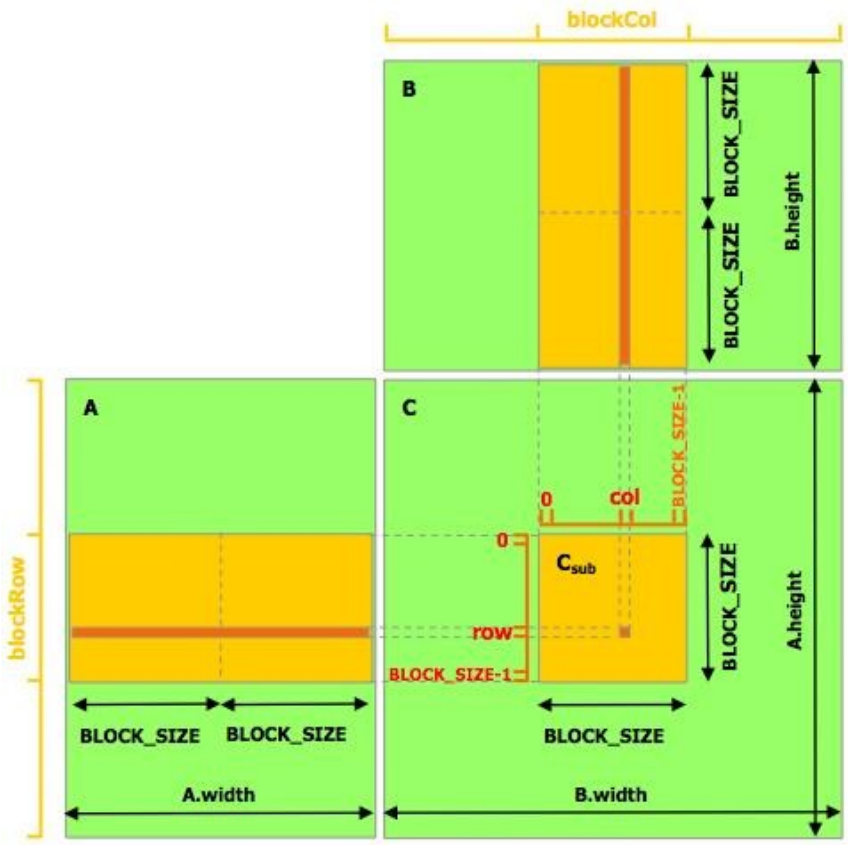

Figura 34 - Divisão do processamento em kernels.

4.3.3.
Segunda Parte: Inversão Matricial

O cálculo da matriz inversa, apesar de não representar a parte mais custosa da Equação 30, é a parte mais complexa do desenvolvimento. Para aproveitar velocidade da memória compartilhada da placa gráfica, cujos acesos pode chegar a ser 300 vezes mais rápido do que o acesso à memória global, o primeiro passo foi dividir a matriz em blocos conforme a Figura 35, da mesma forma como foi feito na multiplicação matricial. A vantagem do tratamento baseado em blocos é

dividir a matriz em submatrizes e transferi-las para memória compartilhada. Dessa forma, os acessos à memória global são limitados a uma única leitura e escrita, melhorando o desempenho. A partir daí, aplica-se a eliminação gaussiana no bloco, onde os pivôs são calculados.

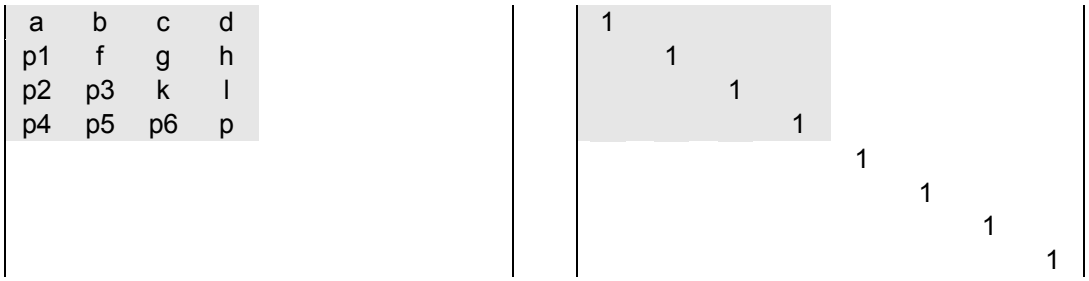

Figura 35 – Divisão da matriz em blocos de tamanho *n* (nesse caso, *n* é 4).

A partir dos pivôs calculados, atualizam-se as n linhas adjacentes das matrizes, conforme a Figura 36. Essa operação é feita nos diversos blocos que, nesse exemplo, estão marcados com cores diferentes. Esta operação é executada linha a linha de forma independente, mas os pivôs são modificados antes da atualização da próxima coluna, o que exige uma sincronização entre os blocos.

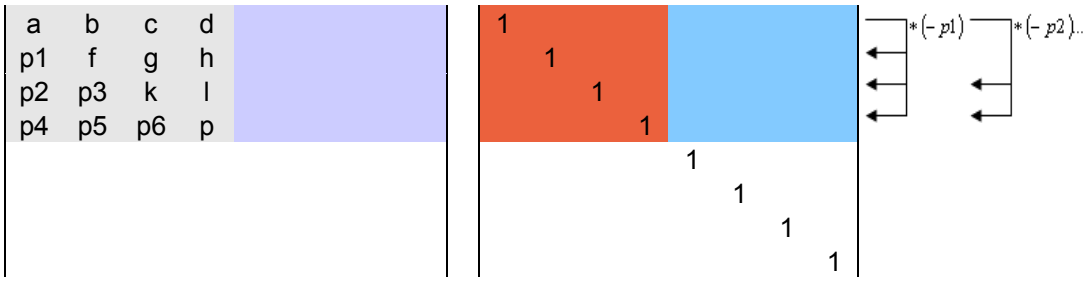

Figura 36 – Atualização das linhas adjacentes.

Os elementos abaixo do pivô do bloco da etapa 1 são necessário para produzir o bloco seguinte. Depois disso, as colunas são calculadas novamente a partir dos pivôs gerados nos passos anteriores, e, em seguida, coluna por coluna do bloco, propaga o resultado para as colunas do bloco 0 salvando os pivôs gerados em cada linha. A Figura 37 mostra como fica o bloco depois dos passos já apresentados.

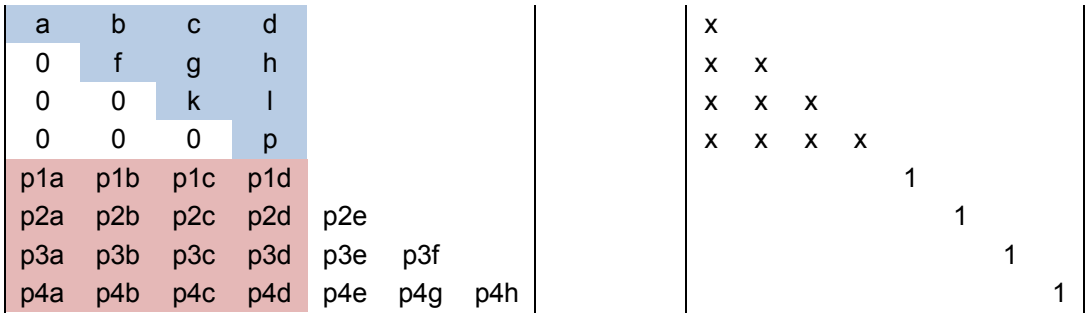

Figura 37 – Blocos atualizados.

A partir dos pivôs calculados na etapa 3, são calculadas as linhas remanescentes da matriz e da unidade matriz original novamente. Esta é a parte mais custosa do algoritmo e que requer para os cálculos os seguintes elementos:

- Pivôs do respectivo bloco (bloco azul escuro na Figura 38);
- Linhas de fatores para a multiplicação dos pivôs (destacado em amarelo na Figura 38);
- O bloco a ser ajustado (de cinza, vermelha, azul claro na Figura 38)

Os cálculos dentro de um bloco são independentes uns dos outros, mas as operações nas coluna devem ser sincronizadas.

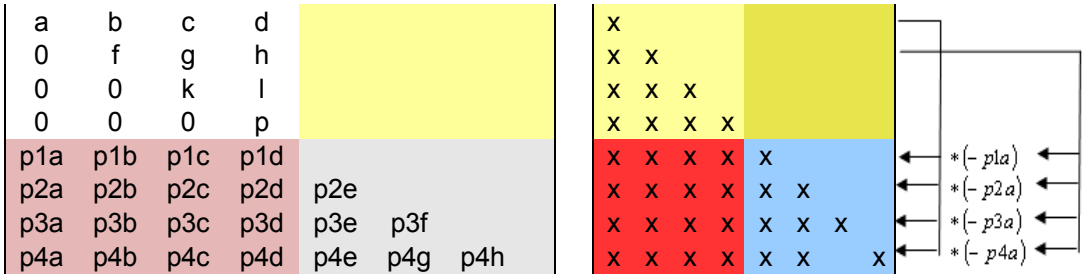

Figura 38 – Linha de fatores para a multiplicação dos pivôs.

Finalmente, todos os elementos abaixo da diagonal principal são iguais a zero. A partir desse ponto, basta repetir os passo 1 ao 4 usando o sub-bloco do passo 1 ao longo da diagonal. Para o restante do processamento, a transformação do resultado a ser observado que a matriz triangular inferior não é preenchido com 0, mas com valores numéricos.

4.4. Ferramenta Gráfica para Criação de Soluções Utilizando Redes Neurais – Clinn

Nesta seção é apresentada a ferramenta denominada Clinn, que significa *Client of Neural Network*, que utilizou, como biblioteca de redes neurais, a ANN-COM, já apresentada anteriormente neste capítulo. O Clinn, cuja tela inicial é apresentada pela Figura 39, também foi desenvolvidos sobre a plataforma .NET da Microsoft, pelos motivos já citados anteriormente.

O Clinn oferece diversos modos de trabalho e visualização para projetos envolvendo uma ou várias redes neurais. Além disso, existem vários modelos "passo-a-passo" que auxiliam na construção das soluções ou projetos e em estratégias para criar e treinar redes genéricas ou baseadas em algum modelo (ex: MLP, Elman).

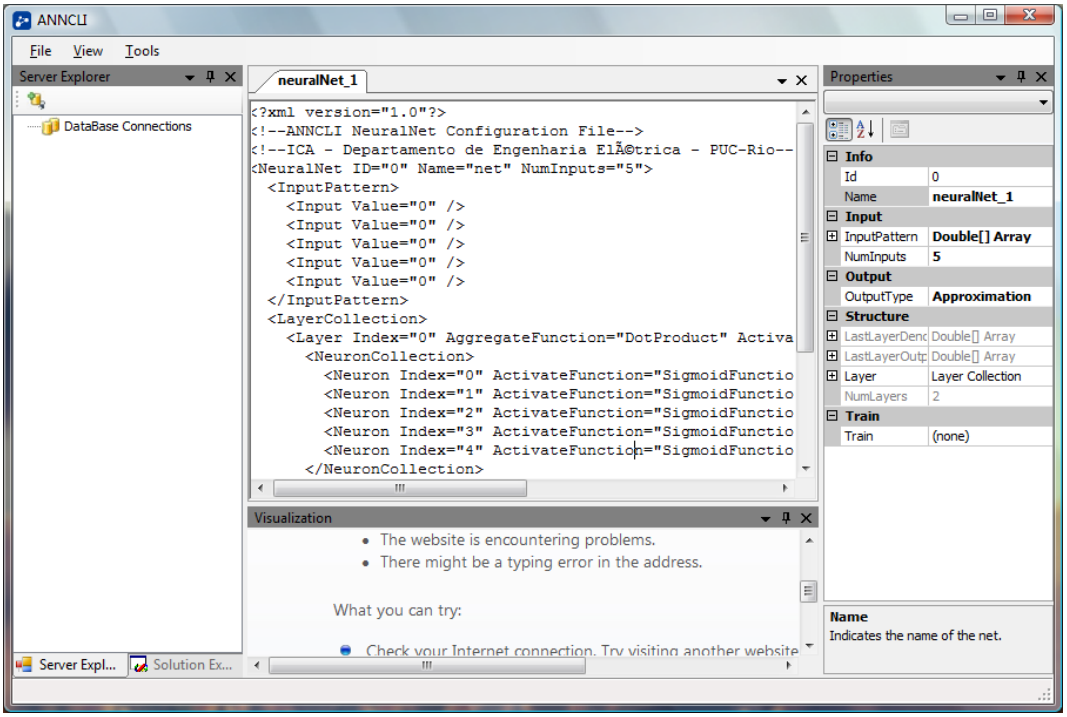

Figura 39 – Tela inicial do Clinn.

4.4.1. Interface utilizando Docas Flutuantes

As interfaces gráficas que utilizam docas flutuantes são muito comuns nos programas atuais, pois apresentaram diversas informações na tela de forma organizada. Na Figura 39 é possível verificar diversas funcionalidades disponíveis em

forma de docas flutuantes. Essas “janelas internas” possuem diversas funcionalidades que facilitam o trabalho do cliente. A Figura 40 mostra a possibilidade de se criar um ambiente completamente personalizado e que pode ser salvo, tornando mais fácil a adaptação ao programa.

Além das docas de funcionalidades (propriedades, explorador de solução e explorador de banco de dados), existe a doca de documentos, onde são editadas as redes neurais por meio de arquivos em formato XML. No futuro, o editor de texto será melhorado de forma a interpretar em tempo real o texto digitado, além de colori-lo, a fim de auxiliar na edição dos arquivos XML. Entretanto, já existe suporte para validação, que é feita cada vez que os documentos são salvos. Nos parágrafos seguintes, cada uma das docas de funcionalidades disponíveis no Clinn será descrita com maiores detalhes.

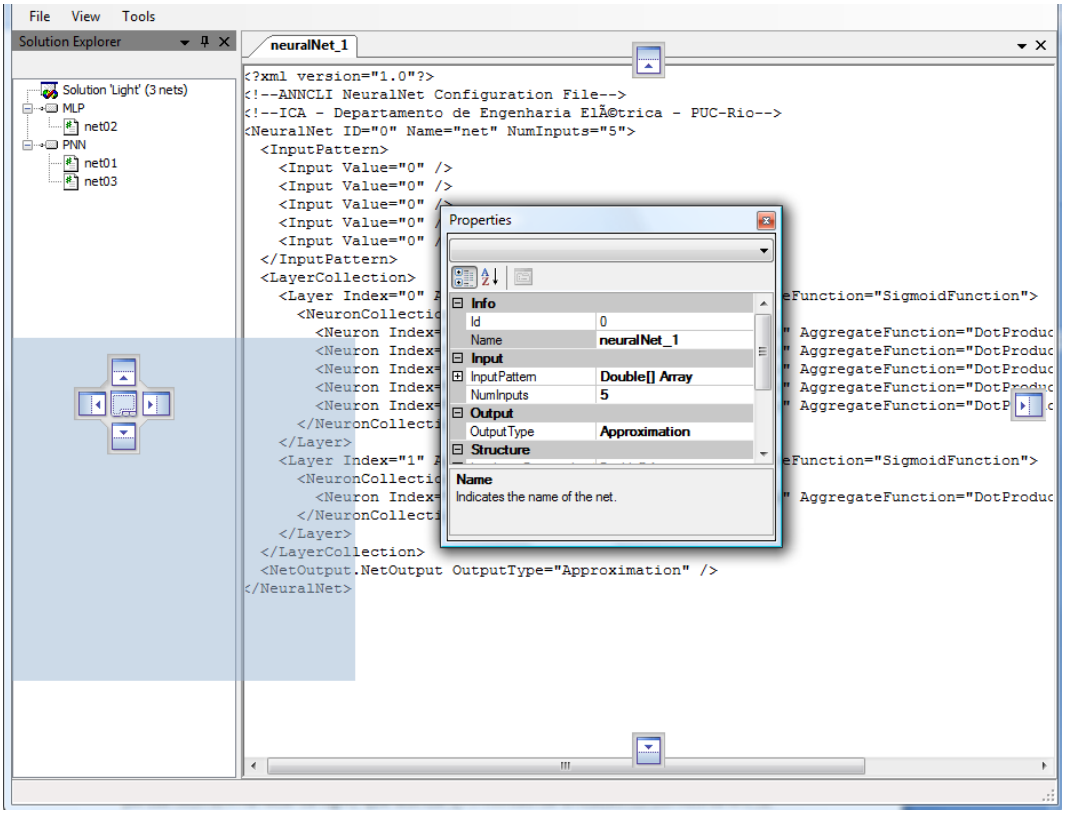

Figura 40 – Movimentação das docas pelo programa.

A doca Propriedades auxilia na edição das redes de uma forma simples e rápida. A Figura 41 mostra maiores detalhes da doca e como é simples ler qualquer informação contida no componente. Por essa doca é possível inserir um padrão

manualmente na camada de entrada da rede neural e propagá-lo através dessa rede, através da propriedade *InputPattern*.

As Figura 42, Figura 43 e Figura 44 mostram a possibilidade de editar as estruturas internas da rede como, respectivamente: camadas, neurônios e sinapses. Na Figura 42, são mostradas as camadas contidas nesta coleção (à direita). Nela, as camadas podem ser removidas e adicionadas, além de poderem ter sua ordem alterada. Neste modo é possível alterar qualquer propriedade das camadas individualmente. Na Figura 43, todas as propriedades dos neurônios são mostradas. Nessa figura, é possível observar que algumas propriedades são somente-leitura para preservar a consistência da rede neural. Na Figura 44, destaca-se o vetor de pesos, que permite visualizar ou editar os mesmos. Nesta tela, todas as propriedades das sinapses podem visualizadas, e algumas, alteradas pelo mesmo motivo indicado anteriormente.

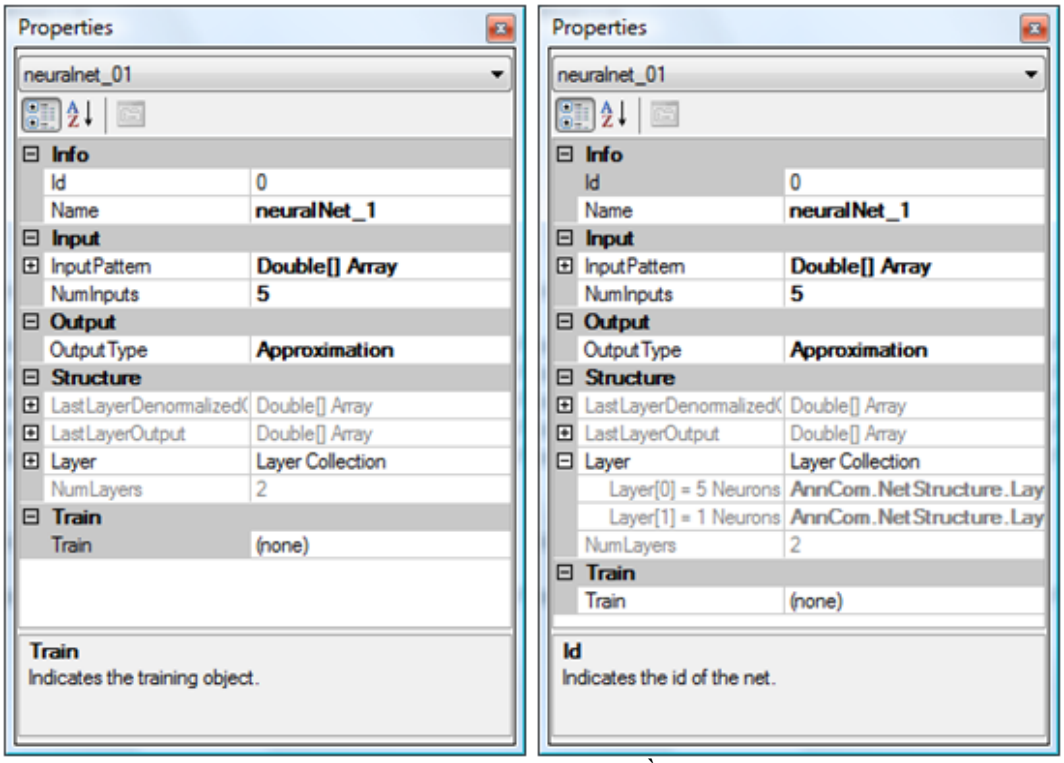

Figura 41 – A doca de propriedades em detalhes. À direita, a figura mostra a facilidade de navegação pelo componente.

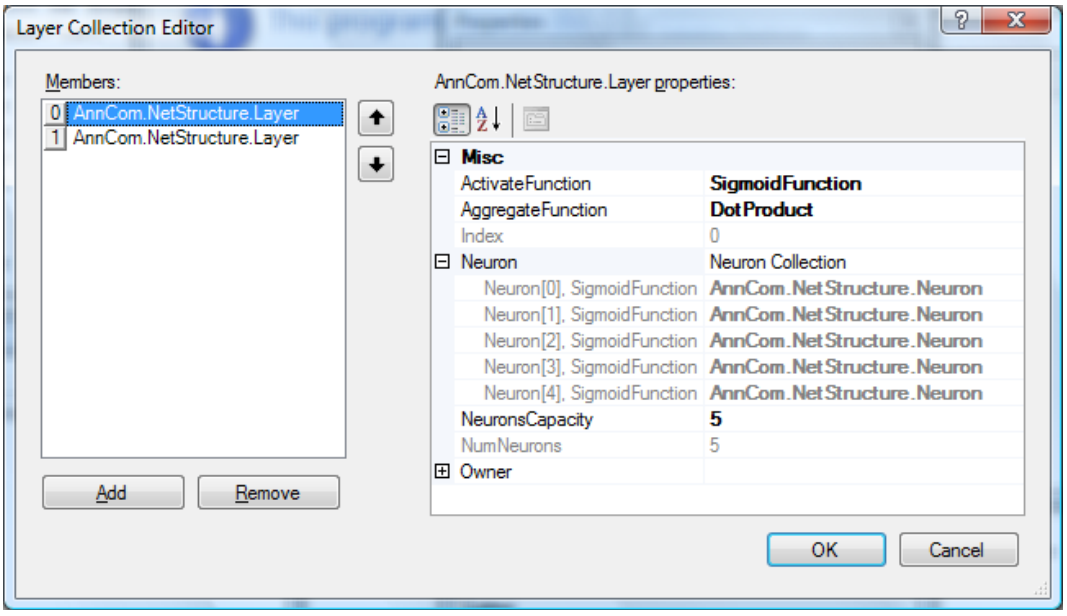

Figura 42 – Editor de coleção de camadas.

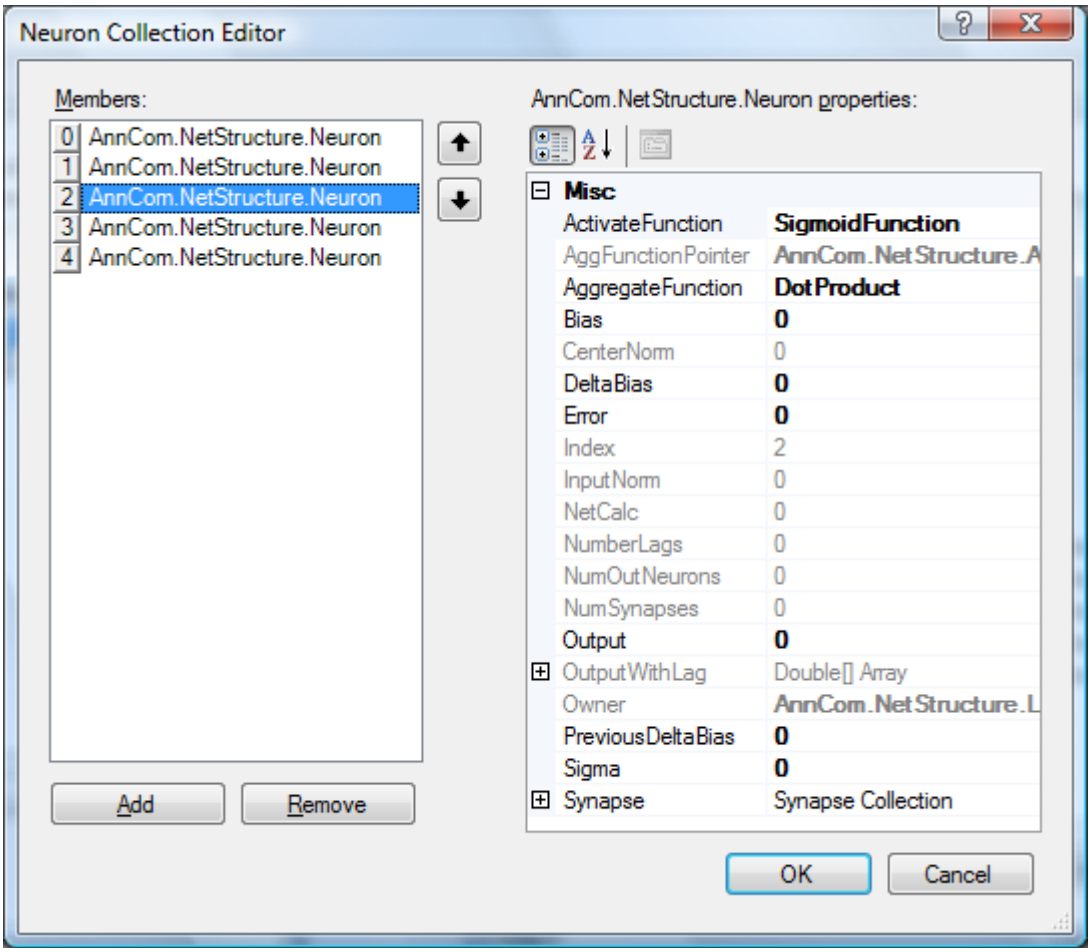

Figura 43 – Editor de coleção de neurônios.

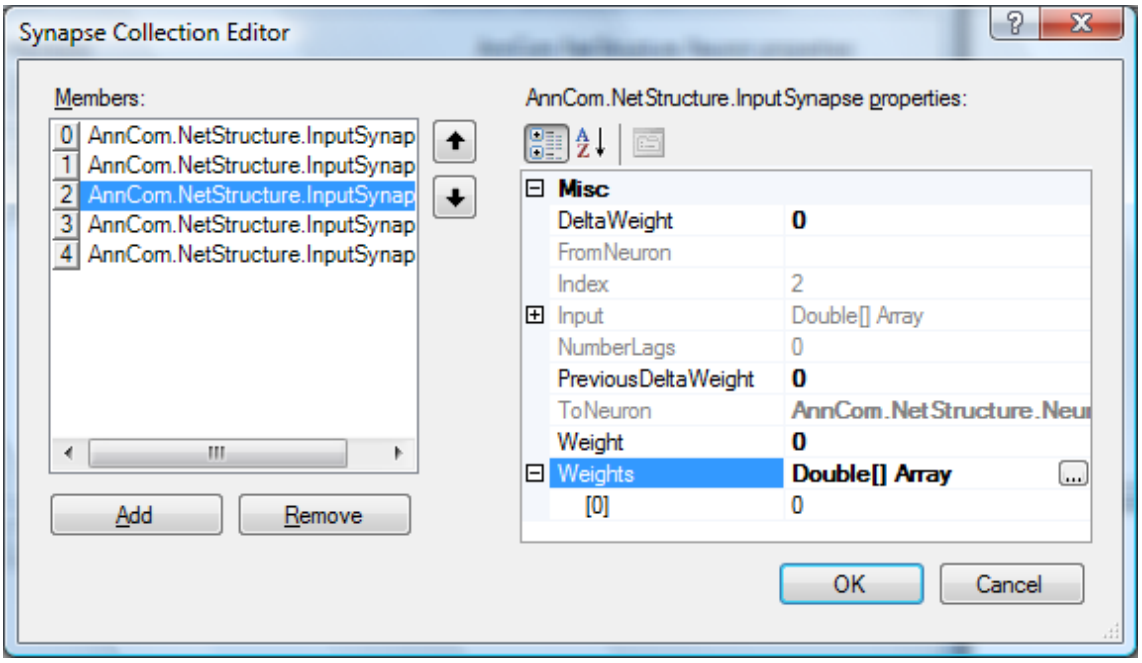

Figura 44 – Editor de coleção de sinapses.

A doca “Explorador de Solução” organiza as redes já criadas na solução corrente, uma vez que, para se encontrar uma boa rede para um problema, é necessário se testar várias redes com diferentes arquiteturas. Por exemplo, se uma estratégia de treinamento é configurada para que se crie redes MLP variando a quantidade de neurônios em uma determinada camada, podem ser geradas muitas redes. Com o objetivo de organizar essas redes, o Explorador de Solução as agrupa como mostra a Figura 45. Nessa figura, pode-se ver uma solução já criada com algumas redes neurais. Para editar qualquer rede contida na árvore, basta clicar duas vezes sobre a mesma e esta será aberta pelo editor de documentos que será apresentado em breve. Além disso, esta doca permite apagar os arquivos, adicionar novos e alterar seus nomes.

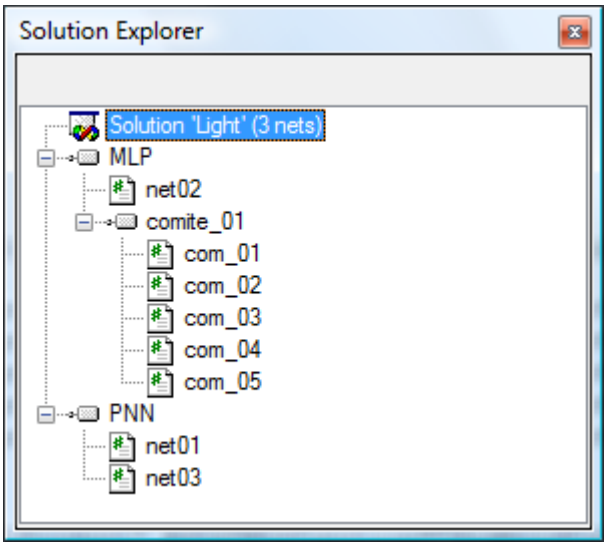

Figura 45 – Doca exploradora de solução.

A doca "Explorador de Bases de Dados" (Figura 46) permite a conexão com bases simultâneas e a visualização de tabelas ou visualizações dessas tabelas de forma simples e rápida. Na Figura 46, o explorador está conectado a uma base de dados SQL e mostra o conteúdo da mesma. Essa doca foi criada, pois a manipulação dos dados é uma parte fundamental na solução de um problema utilizando redes neurais. A rápida visualização e manipulação desses dados podem economizar algumas horas no final de um projeto. A Figura 47 mostra a visualização e a edição de tabelas ou visualizações dessas tabelas.

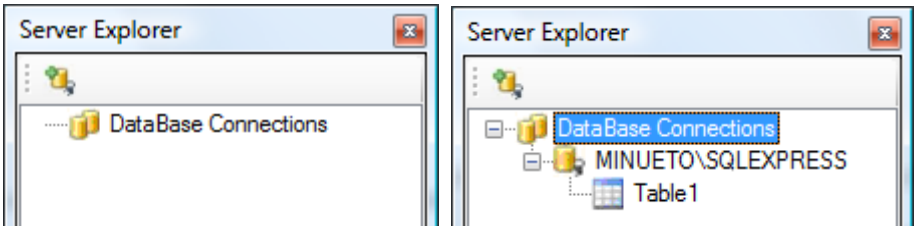

Figura 46 – Explorador de bases de dados.

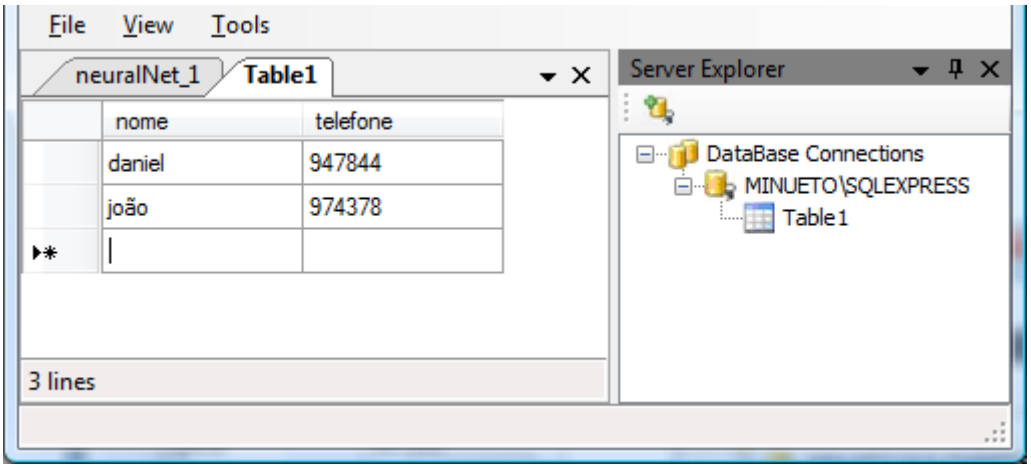

Figura 47 – Possibilidade de visualizar e editar as informações das tabelas.

A doca de treinamento oferece algumas funções que automatizam o processo de criação de uma rede. Nela é possível realizar operações para adicionar treinamento, interconectar a rede, desconectar a rede e inicializar os pesos. A operação de adicionar treinamento cria um componente de treinamento e o liga à rede neural. Interconectar a rede significa interligar todos os neurônios da rede através de sinapses. Os neurônios da camada n são ligados aos da camada n-1 até a primeira camada escondida, pois esta é ligada ao vetor de entrada através da *InputSynapses*. A operação de desconectar toda a rede remove todas as conexões da rede neural em questão. Quando esse comando é acionado, uma janela de confirmação é exibida, pois todo um treinamento pode ser perdido com este comando. A inicialização dos pesos da rede neural pode ser feita de três formas: (1) o usuário escolhe um valor fixo que será copiado para todos os pesos. Isto pode ser útil quando se quer experimentar diferentes tipos de treinamento; (2) o usuário fornece uma semente inicial para geração de pesos pseudo-randômicos. Este método gera pesos próximos de zero para evitar o problema de paralisia da rede; e (3) o usuário deixa o programa gerar pesos pseudo-aleatórios por conta própria. Isto pode ser útil quando se quer experimentar diversas vezes um determinado treinamento.

Futuramente, esta doca suportará mais opções além de uma integração maior com a doca exploradora de base de dados a fim de tornar mais dinâmico o processo.

As docas apresentadas acima são utilizadas para manipular a solução e, de forma gráfica, as propriedades dos componentes utilizados na solução. Para se editar as tabelas, visualizações e redes neurais, a doca de documentos. Ela tem um comportamento de lista, onde as abas representam os documentos abertos que

podem ser fechados ou reordenados. Isto é bastante útil quando se trabalha com muitos documentos abertos ao mesmo tempo. A Figura 48 mostra alguns documentos abertos nessa doca. É importante salientar que os arquivos de configuração de rede neural são validados cada vez que os mesmos são salvos, com o objetivo de mantê-los sempre válidos. Outra característica interessante é o back-up automático dos arquivos, que salva de tempos em tempos os arquivos para protegê-los de falhas no sistema. Mesmo se um arquivo nunca foi salvo, uma cópia temporária deste fica salva no disco rígido. Além das funcionalidades apresentadas, a doca de documentos fornece suporte para o passo a passo que será detalhado a seguir.

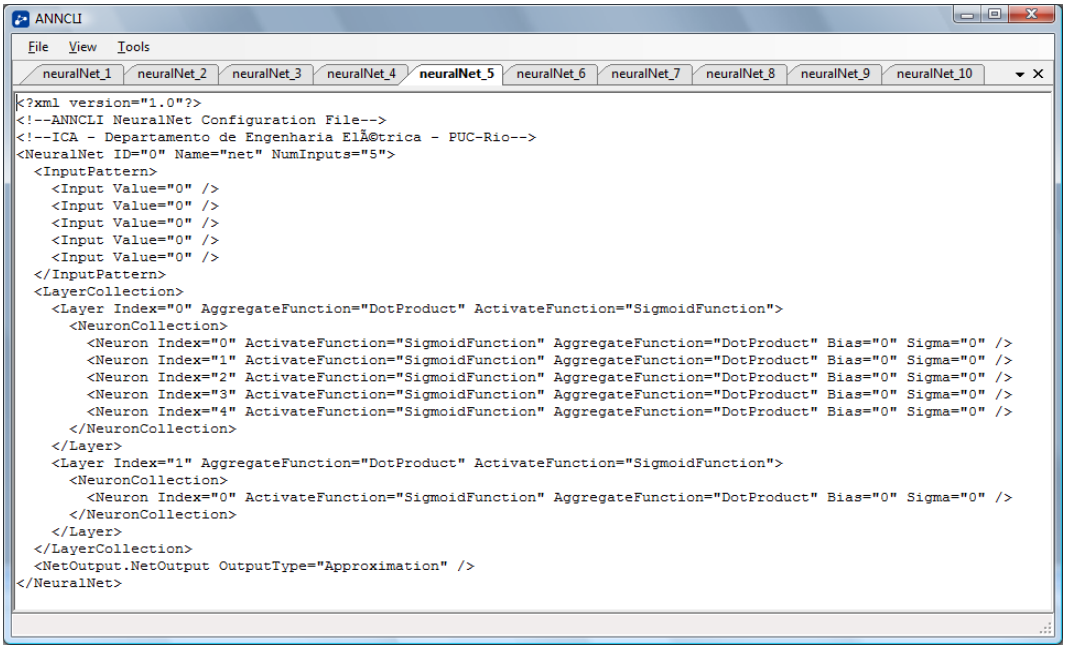

Figura 48 – Doca de documentos com uma lista de arquivos abertos.

4.4.2. Processo de Criação Automatizada

Uma maneira mais fácil de criar novas redes é usando o modelo passo a passo que pode ser visto na Figura 49. Neste modo, as configurações das redes são selecionadas de maneira intuitiva e fácil. Além disso, todos os detalhes da rede vão sendo aprofundados a cada passo. Na primeira parte, características como nome, tipo de estrutura, número de entradas, tipo de normalização, funções de treinamento, tipo de cálculo de erro são apresentados apenas como uma seleção simples.

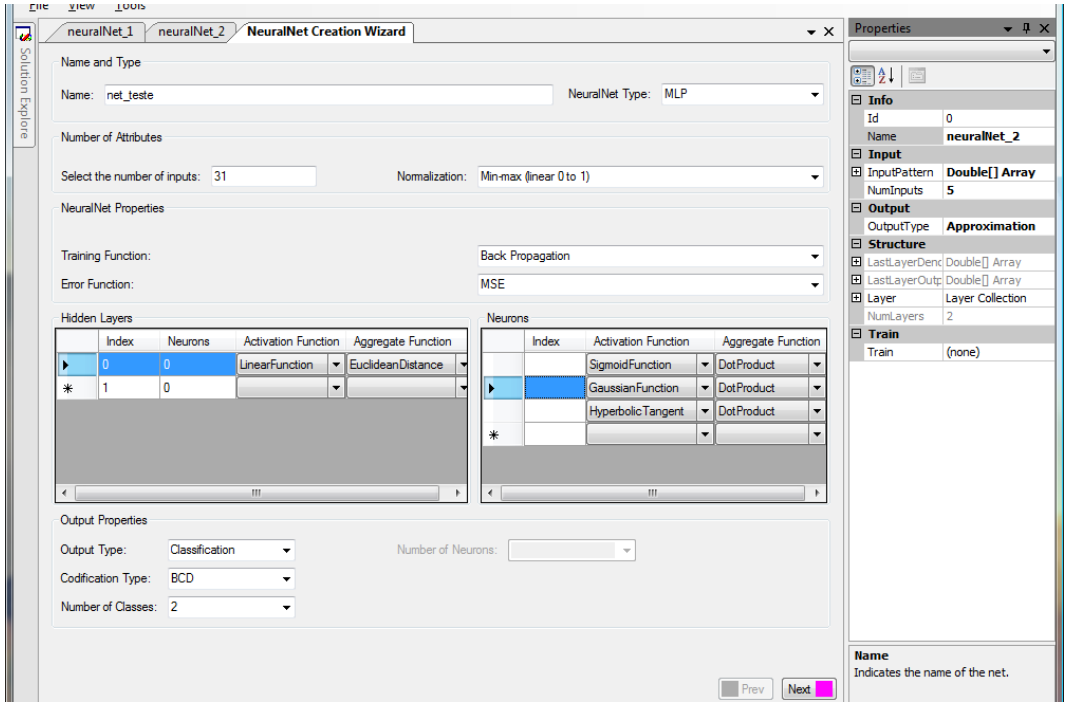

Figura 49 – Modelo passo a passo para se criar uma nova rede neural.

Depois de escolher essas primeiras características, o usuário pode montar a estrutura de camadas das redes adicionando os neurônios e selecionando, para cada neurônio, as funções de ativação e agregação, o que possibilita montar estruturas com várias funções distintas. Na terceira e última parte desta primeira tela é possível escolher o formato de saída da rede entre classificação e aproximação (para as redes de agrupamento, essas opções ficam desabilitadas). Como são muitas telas para descrever os diferentes caminhos possíveis no modo passo a passo, apenas a primeira tela é aqui apresentada.

4.4.3. Processo de Treinamento

Para treinar as redes, o Clinn oferece outro modelo passo a passo que cria estratégias de treinamento. Nessas estratégias, é possível escolher limites para variar o número de neurônios nas camadas, programar alterações nas funções de ativação e agregação, o número de épocas, o valor do momento e até ativar uma "sintonização" automática para os limites superiores e limites inferiores nas saídas dos neurônios da última camada em rede neurais de classificação. Por exemplo, o gráfico da Figura 50 mostra um limite superior (*upper threshold*) em 0,8 e um

limite inferior (*lower threshold*) em 0,1. Isto quer dizer que todos os valores maiores ou iguais a 0,8 serão interpretados como um e todos os valores menores que 0,1, como zero. A região entre esses dois valores é chamada de região de dúvida. Neste caso a rede neural não sabe informar a que grupo ou classe o padrão apresentado pertence. Nesta última opção, futuramente será possível executar um algoritmo genético para mapear os melhores valores.

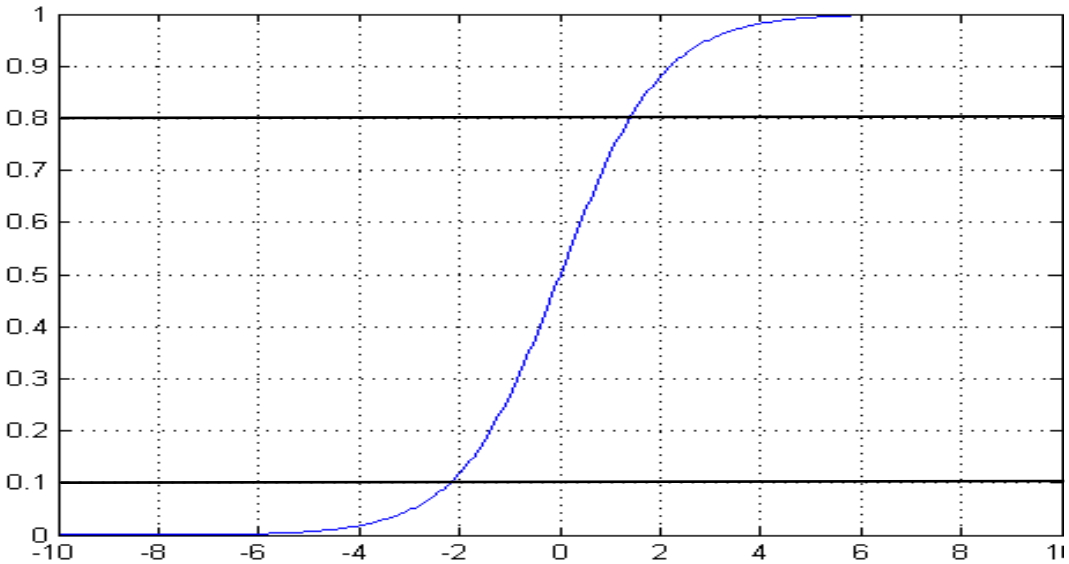

Figura 50 – Função sigmoid de um neurônio da última camada da rede neural.