UNIVERSIDADE FEDERAL DE SANTA CATARINA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Matheus Braun Magrin

EXPANSÃO DO MÓDULOS DE REDES NEURAIS DO GNU OCTAVE

Monografia submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Luiz Angelo Daros de Luca

Florianópolis, março de 2013

EXPANSÃO DO MÓDULO DE REDES NEURAIS DO GNU OCTAVE

Matheus Braun Magrin

Esta monografia foi julgada adequada para a obtenção do título de Bacharel em Ciência da Computação e aprovada em sua forma final pelo Curso de Bacharelado em Ciência da Computação.

_____

Vitório Bruno Mazzola, Dr.

(Coordenador do Curso)

Banca Examinadora:

_____

Luiz Angelo Daros de Luca, Me. (orientador)

_____

Antonio Carlos Mariani, Me. (coorientador)

_____

Mauro Roisenberg, Dr.

Agradecimentos

Agradeço à minha família, pelo apoio incondicional, mesmo que distante.

Agradeço à Laís Souza, pela parceria na vida e especialmente nas noites de trabalho mal dormidas.

Agradeço aos amigos, pela *fraternagem*.

Agradeço ao meu orientador, Luiz, pela sugestão de um tema que além de interessante, vai se reverter em benefício à comunidade do software-livre e à sociedade e também pelo apoio durante todo o processo.

Agradeço ao Laboratório de Sistemas de Conhecimento - LSC da Universidade Federal de Santa Catarina - UFSC pelo apoio no desenvolvimento deste trabalho.

Este trabalho foi desenvolvido no âmbito do Programa de Pesquisa e Desenvolvimento Tecnológico do Setor de Energia Elétrica da CELESC Distribuição S.A., regulado pela Agência Nacional de Energia Elétrica - ANEEL, dentro do escopo do projeto de P&D 5697-0210/2011 – Parâmetros Preditivos para a Energia e a Carga no Mercado de Energia Elétrica da CELESC.

I expect to pass through this world but once. Any good, therefore, that I can do or any kindness I can show to any fellow creature, let me do it now. Let me not defer or neglect it for I shall not pass this way again.

Stephen Grellet

Resumo

Redes neurais artificiais são um campo da Inteligência Artificial. Surgiram como classificadores mas hoje em dia podem ser aplicadas para aproximação de funções, sistemas de controle em robótica, filtros adaptativos e também reconhecimento de padrões. Este trabalho descreve a implementação de um módulo de redes neurais artificiais para o GNU Octave. A implementação teve por objetivo manter a compatibilidade com o módulo existente do MATLAB. O módulo permite a criação de redes feed-forward e permite que as redes sejam treinadas utilizando o algoritmo de retro-propagação de erro por descida em gradiente e Levenberg-Marquardt.

Palavras-chave: redes neurais artificiais, GNU Octave, MATLAB.

Abstract

Artificial neural networks are a field of Artificial Intelligence. They were created as classifiers, but today can be applied in the function approximation, robotics control systems, adaptative filters and also in pattern recognition. This works describes the implementation of a artificial neural networks module for GNU Octave. One of this implementation's main goals were to maintain compatibility with the existent MATLAB toolbox. This module allows the creation of feed-forward networks e allows them to be trained with the gradient descent error back-propagation and Levenberg-Marquardt algorithms.

Keywords: artificial neural networks, GNU Octave, MATLAB.

Índice de Figuras

Sumário

1 INTRODUÇÃO

A Inteligênca Artificial (IA) centra-se na criação de modelos para a inteligência e a construção de sistemas computacionais baseados nesses modelos (BITTENCOURT, 1998). Pode-se dizer que dentro da IA existem basicamente duas vertentes de pesquisa: a IA simbólica e a IA conexionista. A vertente simbólica busca, através da manipulação de conceitos e símbolos, simular comportamentos inteligentes. A abordagem conexionista busca estudar e simular os comportamentos inteligentes existentes na natureza e os mecanismos responsáveis por eles.

Uma das áreas pertencentes à IA conexionista são as Redes Neurais Artificiais (RNA). Segundo (HAYKIN, 2001), RNAs podem ser vistas como processadores paralelamente distribuídos, constituídos de unidades de processamento simples. Estas unidades de processamento são estruturas capazes de armazenar conhecimento experimental e torná-lo disponível para o uso. Essa capacidade confere à RNA a possibilidade de aprender a partir de conjuntos de dados. Elas tem aplicação em diversas áreas, como na análise de séries temporais, reconhecimento de padrões e processamento de sinais.

Estão disponíveis diversos softwares que possibilitam ao usuário trabalhar com RNAs de maneira facilitada. Um destes softwares é o GNU Octave, que é um ambiente de programação de alto nível, focado na computação numérica e científica, desenvolvido sobre uma licença de software livre. Ele foi inspirado pelo Matlab, um software anterior, que já era utilizado para essa finalidade. O Matlab possui uma extensa gama de ferramentas, incluindo um módulo para trabalhar com RNAs. Porém é um software proprietário e de custo elevado, o que impossibilita sua utilização à um grande público. Embora ambos tenham módulos que permitem a criação e utilização de RNAs, é reconhecido que o módulo do Matlab apresenta uma gama maior de arquiteturas e algoritmos disponíveis. No intuito de ampliar o acesso à essa tecnologia, este trabalho desenvolverá modelos de RNAs para o Octave, expandindo as opções atualmente disponíveis.

1.1 Objetivos

Expandir as opções de arquiteturas e algoritmos de redes neurais artificiais disponíveis no ambiente GNU Octave.

Objetivos específicos:

a) Implementar as funcionalidades necessárias para a construção de RNAs do tipo *multilayer perceptron;*

b) Implementar os algoritmos de treinamento por retro-propagação de erro usando

descida em gradiente e o algoritmo de Levenberg-Marquardt;

c) Buscar compatibilidade com o módulo de RNA do MATLAB, permitindo a execução de programas desenvolvidos para o MATLAB com poucas ou nenhuma alteração;
d) Manter compatibilidade do código para este ser utilizado no MATLAB em substituição ao módulo de RNA existente;
e) Realizar um estudo comparando o desempenho, em termos de confiabilidade dos resultados, entre o produto obtido e a solução já existente no MATLAB;
f) Aplicar as arquiteturas e modelos desenvolvidos para a solução de problemas;
g) Submeter o módulo desenvolvido para aprovação da comunidade desenvolvedora do Octave-Forge, projeto que reúne os módulos do GNU Octave.

1.2 Escopo do trabalho

O módulo desenvolvido visa aumentar o acesso às tecnologias relacionadas às RNAs. Não faz parte do escopo do projeto atingir o mesmo desempenho (temporal ou computacional) obtido pelas soluções proprietárias existentes, tampouco é o foco do projeto fazer uma cópia do módulo existente no MATLAB. O projeto visa fornecer alguns dos principais métodos para as grandes classes de problemas que podem ser resolvidos utilizando RNAs.

1.3 Procedimentos metodológicos

O trabalho iniciou pela análise do módulo de RNAs já existente no Matlab, catalogando as funções que existem e analisando também o paradigma usado na API. É importante frisar que durante esse processo o código fonte do MATLAB não foi consultado, pois a licença não permite a reprodução. Para a implementação no ambiente Octave, foram escolhidas as arquiteturas e algoritmos mais relevantes. Esta relevância levou em conta as classes de aplicabilidade para cada técnica, visando aumentar a classe de problemas que poderão ser abordados utilizando o Octave juntamente com o módulo criado. Além disso, foi levado em conta também a importância histórica e pedagógica dos métodos. A avaliação da corretude da implementação se deu através de estudos de caso que permitiram comparar os resultados obtidos a partir do Octave com os do MATLAB.

2 REDES NEURAIS ARTIFICIAIS

2.1 Inteligência Artificial e RNAs

A IA é uma disciplina da Computação que tem ligação com diversas outras áreas do conhecimento, como por exemplo filosofia, neurologia e psicologia. Essa interdisciplinariedade é devida à diversidade de abordagens que os cientistas da área usam para tratar da inteligência.

Um dos primeiros trabalhos na área, anterior ao termo "Inteligência Artificial"[1], foi um artigo publicado por McCulloch e Pitts em 1943 (MCCULLOCH; PITTS, 1943). Neste artigo eles mostraram o modelo simplificado de um neurônio. Este modelo foi criado numa tentativa de entender como o cérebro conseguia produzir padrões complexos de funcionamento se baseando apenas em células simples inter-conectadas. Não muito tempo depois, em 1950, Alan Turing propôs um teste para inteligência. Ele partiu da premissa que o ser humano é inteligente assim, se o comportamento do computador for indistinguível do de uma pessoa, este também deve ser inteligente. Neste teste, um juiz faz perguntas para uma pessoa e um computador. Se ele não conseguir distinguir entre um e outro, diz-se que o computador passou no teste. Para Turing essa era uma definição satisfatória para "inteligência". Esse teste, conhecido como Teste de Turing, apesar das críticas em relação à sua aplicabilidade, foi e ainda é relevante para a IA. Ainda no início dos anos 50 pesquisadores já haviam criado alguns programas capazes de jogar Damas. Em 1952, Arthur Samuel elevou os padrões desse tipo de programa. O seu invento contava com aprendizado e generalização. Essas características permitiam ao programa jogar contra si mesmo e assim aprimorar suas estratégias para ganhar. Essa abordagem de competição foi um dos primeiros exemplos de computação evolucionária (JONES, 2008).

Outro marco importante na história da IA foi quando, em 1957, Frank Rosenblatt criou o *perceptron*. O *perceptron* é um modelo neural simples, que usa um algoritmo não supervisionado para classificar dados em duas classes. Esse trabalho trouxe um grande interesse da comunidade científica para as redes neurais (JONES, 2008). Na década de 60, foram publicados vários trabalhos enfatizando a abordagem *bottom-up* de construção da inteligência. Nestes trabalhos temos exemplos de modelagem de neurônios. (WIDROW, 1962) (GROSSBERG, 1967) (GROSSBERG, 1968)

A aparente crença de que as RNAs podiam fazer qualquer coisa caiu por terra em 1969 com a publicação do livro Perceptrons, de Minsky e Papert (MINSKY; PAPERT, 1969). Nessa publicação eles provaram matematicamente as restrições estruturais dos *perceptrons* de única camada. Eles também não acreditavam que o acréscimo de mais camadas fosse suficiente para transpor as limitações descritas. Devido a isso, a área das RNAs sofreu uma grande perda de interesse por parte dos

1 O nome "Inteligência Artificial" foi cunhado na proposta do Dartmouth Summer Research Project on Artificial Intelligence que aconteceu em 1956.

pesquisadores nos anos 70 (RUSSEL; NORVIG, 2003).

Nos anos 80 foram feitas grandes contribuições para a IA. Essa década foi marcada por um crescente interesse nas RNAs. Como trabalhos do início desse período pode-se citar as redes *adaptative ressonance theory (*ART) , redes de Hopfield e mapas auto-organizáveis de Kohonen. Em 1985 foi documentada a primeira rede neural multi-camada, a máquina de Boltzmann. Esse trabalho foi importante, pois mostrou que as especulações de (MINSKY; PAPERT, 1969) sobre as redes multi-camadas eram falsas. No ano seguinte, foi desenvolvido o algoritmo de retro-propagação (*back-propagation)* que veio a ser um dos algoritmos mais populares para o treinamento de redes m*ultilayer perceptron* (MLP) ou perceptrons multi-camadas (HAYKIN, 2001). No ano de 1988 foram criadas as redes *radial basis function (*RBF) ou função de base radial, que também são redes multi-camadas e podem ser usadas como alternativa às MLPs. Alguns anos depois, no início dos anos 90, Vapnik, em alguns trabalhos com outros pesquisadores, desenvolveu uma nova classe de redes de aprendizagem supervisionada, chamadas *support vector machines (SVM)*.

2.2 Os neurônios biológico e artificial

Os neurônios são os constituintes estruturais e as unidades de processamento do sistema nervoso. Até pouco tempo, acreditávamos que o cérebro humano continha aproximadamente 10 bilhões de neurônios (HAYKIN, 2001), mas hoje sabemos que contém em média 86 bilhões (AZEVEDO et. al., 2009).

O neurônio é composto por um núcleo, dendritos e um único axônio. Os dendritos são os sensores do neurônio, recebendo os sinais de outros neurônios. O axônio é a saída, a linha de transmissão do impulso elétrico gerado pelo núcleo. Para o núcleo emitir algum sinal, os seus impulsos de entrada devem atingir um nível que ultrapasse o limiar de disparo. Este sinal emitido recebe o nome de potencial de ação (KOVÁCS, 2002). A comunicação entre os neurônios acontece através das sinapses, que são processos eletro-químicos. Na sinapse, o sinal elétrico vindo de um neurônio causa a liberação de uma substância que, por sua vez, faz com que seja gerado um impulso elétrico no neurônio receptor.

O neurônio artificial é a unidade básica das RNAs. Na Figura 1 está mostrada a representação mais comum de um neurônio artificial, com seus pesos sinápticos, somador e função de ativação representados. Assim como nos neurônios biológicos, as ligações entre neurônios artificiais também são chamadas sinapses e à cada sinapse artificial é atribuído um peso. Os pesos sinápticos servem para modificar a influência exercida pelo neurônio emissor no neurônio receptor. O somador agrega os sinais vindos das sinapses, as entradas do neurônio, e a função de ativação foi incluída para fazer jus ao limiar de disparo encontrado nos neurônios. Neste modelo as entradas e os pesos representam os dendritos e suas sinapes. A função do núcleo é exercida pelo

somador e a função de ativação. Já o axônio biológico está representado na única saída.

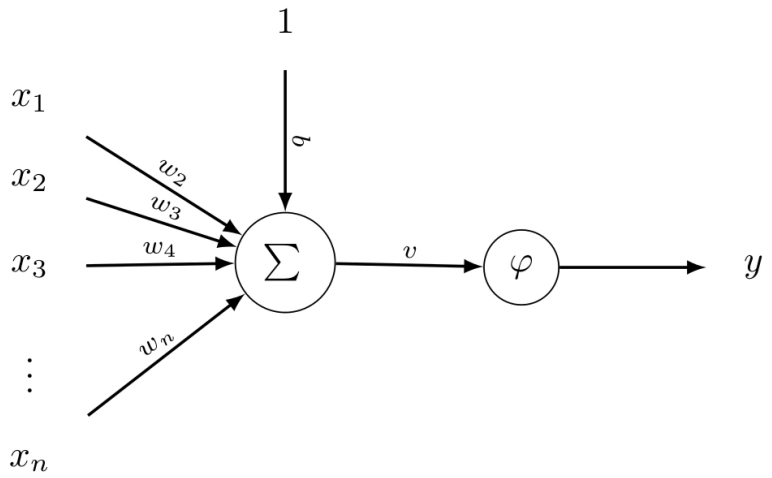

Figura 1: Modelo de um neurônio artificial

Os valores x_i com $1<i<m$ representam o vetor que será submetido como entrada para o neurônio, são chamados de sinais de entrada. O valor x_0 será fixado em 1 e será multiplicado por b , o viés da rede. Os pesos sinápticos, representados por w_i , também são parte de um vetor de dimensão $1\times m \in \mathbb{R}$. O somador realiza a soma dos sinais de entrada ponderados pelos respectivos pesos sinápticos e o viés. A função de ativação $\varphi(\cdot)$ é então aplicada na saída do somador gerando a saída y do neurônio.[2] Este modelo pode também ser representado de maneira formal pela equação (1). Vale notar a importância do viés, que é agregado às entradas para permitir que a superfície (ou linha) de saída do neurônio se distancie da origem.

$$y=\varphi\left(\sum_{i=0}^{m} w_i x_i\right) \tag{1}$$

2.2.1 Funções de ativação

A função degrau, também conhecida de função de Heaviside é bastante utilizada em redes classificadoras. O modelo descrito em (MCCULLOCH; PITTS, 1943) utilizava essa função. Essa função foi escolhida devido à sua aproximação ao comportamento do potencial de ação encontrado nos neurônios biológicos. A função está descrita na equação (2). O seu gráfico, assim como o das funções mencionadas abaixo é mostrado na Figura 2.

2 Optei por utilizar aqui as nomenclaturas clássicas na literatura, tipicamente em inglês, onde *b* vem de *bias* e *w* de *weight*.

$$\varphi(x)=\begin{Bmatrix}1 & se \quad x>0,\\ 0 & se \quad x\leqslant 0.\end{Bmatrix} \tag{2}$$

A função sigmoide, equação (3), foi proposta por se comportar de maneira semelhante à função degrau, dado um parâmetro *a* adequado. Ela tem como vantagem ser diferenciável em todos os seus pontos, característica necessária para usar algoritmo de retro-propagação.

$$\varphi(x)=\frac{1}{1+\exp(-ax)} \tag{3}$$

Existem ainda várias outras opções, entre elas a linear e a linear saturada. Esta última tem comportamento igual à linear dentro do intervalo [0,1], porém sua imagem fora deste intervalo é limitada a 0 ou 1. Suas funções pode ser encontradas nas equações (4) e (5), respectivamente.

$$\varphi(x)=x \tag{4}$$

$$\varphi(x)=\begin{Bmatrix}0 & se \quad x\leqslant 0,\\ x & se \quad 0<x<1,\\ 1 & se \quad x\geqslant 1\end{Bmatrix} \tag{5}$$

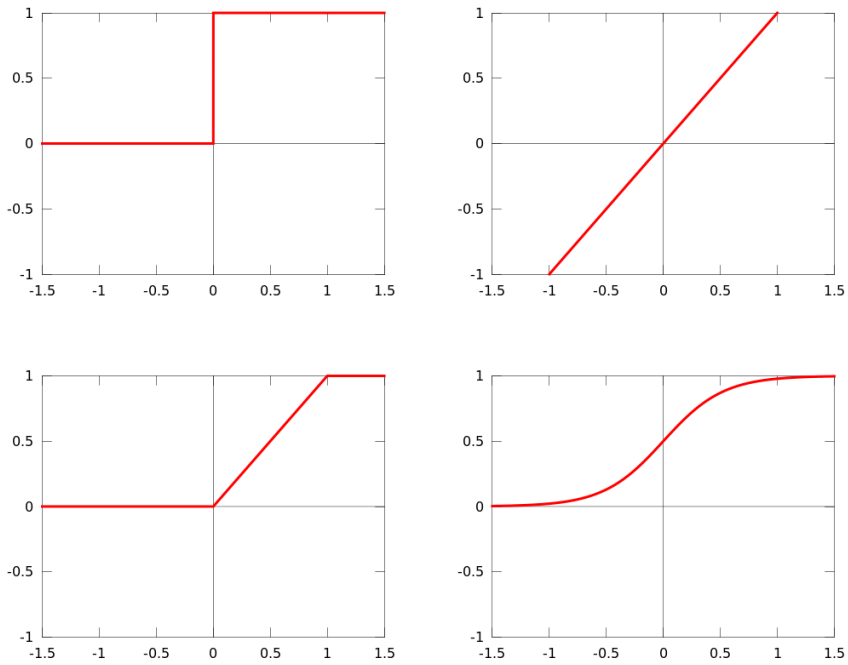

Figura 2: Funções de ativação, em sentido horário a partir do canto superior esquerdo: função degrau, função linear, função sigmoide e função linear saturada

2.3 Treinamento e aprendizagem

Segundo (HAYKIN, 2001), a aprendizagem em redes neurais é:

> (...) um processo pelo qual os parâmetros livres de uma rede neural são adaptados através de um processo de estimulação a partir do ambiente na qual a rede está inserida. O tipo de aprendizagem é determinado pela maneira em que as mudanças nos parâmetros acontecem.

Para que a aprendizagem aconteça, deve-se submeter a rede à um processo de treinamento. De forma geral, o treinamento de uma RNA pode ser conduzido de duas maneiras (EGGERMONT, 1998):

- Treinamento supervisionado. Durante o treinamento supervisionado as respostas corretas já são conhecidas e são utilizadas pelo algoritmo para a aprendizagem. Uma forma especial deste tipo de treinamento é o treinamento por reforço, no qual a resposta exata não é informada à rede, ela só fica sabendo se sua resposta foi certa ou não.
- Treinamento não-supervisionado. Usa-se esse tipo de treinamento quando as respostas corretas não são conhecidas ou não se pretende utilizá-las. Desta forma, a rede deve tentar descobrir sozinha os padrões nos dados de entrada.

Ainda segundo (HAYKIN, 2001) , existem cinco tipos básicos de

aprendizagem, aprendizagem por correção de erro, aprendizagem baseada em memória, aprendizagem Hebbiana, aprendizagem competitiva e aprendizagem de Boltzmann.

A aprendizagem baseada em erro consiste em uma série de ajustes nos pesos sinápticos, para que a saída da rede ($y(t)$), seja cada vez mais próxima do valor esperado ($d(t)$), para cada sinal de entrada $x(t)$. Para isso, depois da aplicação de um sinal de entrada, o erro ($e(t)$) é calculado e com ele, calcula-se a função da energia do erro, definida por:

$$\mathrm{E}=\frac{1}{2}e^2(t) \tag{6}$$

Minimizando-se essa função leva à regra delta de atualização de pesos, também conhecida como regra de Widrow-Hoff (equação (7)). Essa regra leva em conta uma taxa de aprendizagem η , que vai regular o tamanho do "passo" em direção à solução. Com esse ajuste calculado atualiza-se os pesos de todas as sinapses de entrada daquele neurônio usando a equação (8).

$$\Delta w_j(t)=\eta e(t)x_j(t) \tag{7}$$

$$w_j(t+1)=w_j(t)+\Delta w_j(t) \tag{8}$$

A aprendizagem baseada em memória consiste em armazenar todos, ou quase todos, os estímulos recebidos pela rede. Depois de treinada, a rede vai ter um conjunto de pares de entrada e saída corretos. Desta forma, quando for pedido que a rede classifique alguma entrada nunca vista x , a tarefa de classificação consiste em buscar o vizinho mais próximo, que podemos chamar de x' , e atribuir para a x a classificação do seu vizinho (d'). Uma das variações possíveis para essa abordagem é, ao invés de buscar apenas o vizinho mais próxima, buscar o *k* vizinhos mais próximos e atribuir à entrada a classificação mais comum no conjunto dos *k* vizinhos.

A aprendizagem Hebbiana se baseia nas descoberta de Hebb, de onde ganhou o nome, de que quando dois neurônios que estejam ligados, e um deles contribui repetidas vezes para a ativação do outro, a ligação sináptica entre eles é reforçada, de forma que o comportamento de ativação concomitante seja estimulado. Para as RNAs, além do reforço positivo já citado, usa-se também o reforço negativo, que consiste em enfraquecer a sinapse entre neurônios que seja ativados assincronamente. A variação nos pesos mais simples para este tipo de aprendizagem é a hipótese de Hebb, que pode ser vista na equação (9).

$$\Delta w_j(t)=\eta y(t)x_j(t) \tag{9}$$

A competição que acontece entre os neurônios de saída de uma rede caracteriza a aprendizagem competitiva. Nestas redes, para um neurônio de saída se tornar ativo, o campo local induzido (aplicação da saída do somador na função $\varphi(\cdot)$) deve ser maior do que o de todos os outros neurônios de saída. No final do processo, os neurônios

aprendem a responder à uma certa característica de cada classe do conjunto de entradas.

Este tipo de aprendizagem apresenta três elementos básicos, 1) um conjunto de neurônios com pesos sinápticos aleatórios, de modo que eles respondam diferentemente às entradas; 2) limite na grandeza dos pesos sinápticos desses neurônios; e 3) mecanismo que permita a competição pelo direito de ficar ativo.

Neste caso, a regra da variação dos pesos é dada por:

$$\Delta w_j = \begin{cases} \eta(x_j - w_j) & \text{se o neurônio ganhou a competição,} \\ 0 & \text{se o neurônio perdeu a competição} \end{cases} \quad (10)$$

As RNAs que utilizam a aprendizagem de Boltzmann são chamadas de máquinas de Boltzmann. Os neurônios destas máquinas tem suas saídas definidas pelos seus estados, que podem ser “ligado” ou “desligado”, com valores numéricos de 1 e -1, respectivamente. A aprendizagem, neste caso, não muda os pesos sinápticos, mas acontece pela inversão seletiva dos estados dos neurônios. E essa inversão acontece regulada por uma probabilidade, por isso é um processo estocástico.

2.4 Arquiteturas

Existem diversas arquiteturas para as RNAs como redes MLP, ART, de Hopfield; máquinas de Boltzmann e máquina de vetor de suporte. Este trabalho focará nas redes *MLP*.

2.4.1 Perceptrons multi-camadas (MLP)

Os MLP são redes *feed-forward* compostas por ao menos três camadas. Uma camada de entrada, uma de saída, e uma ou mais camadas ocultas. Nestas redes, cada um dos neurônios de uma camada está conectado com todos os neurônios da camada seguinte e usualmente utilizam alguma função sigmoide (ORAVEC; PAVLOVIČOVÁ, 2007). A camada de entrada difere das demais pois seus neurônios não são nodos computacionais, eles somente transmitem as entradas para a segunda camada.

Para treinar este tipo de rede, comumente utiliza-se o algoritmo de retro-propagação do erro. Esse algoritmo utiliza a regra de aprendizagem por correção de erro. O treinamento é dividido em duas etapas; uma etapa no sentido entrada-saída, e outra no sentido contrário. Na primeira etapa os sinais de entrada são aplicados na rede, e o efeito dessa entrada é então propagado pela rede camada a camada. Chegando à última camada, a rede emite sua saída. Com base na saída obtida e na resposta desejada é calculado o erro, que é usado para ajustar os pesos, assim como descrito na seção anterior.

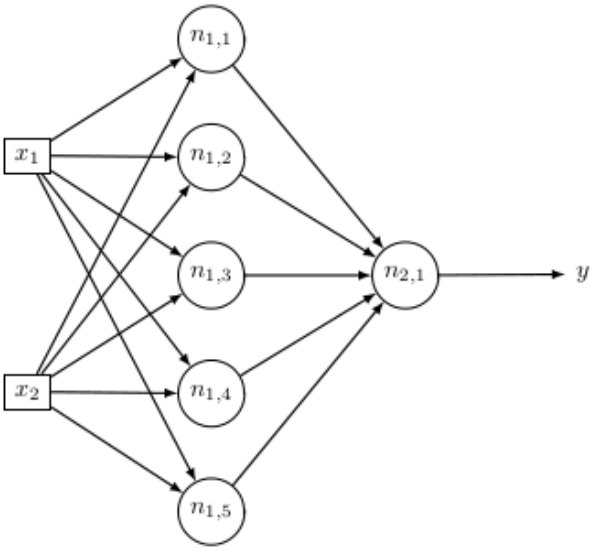

Figura 3: Modelo de uma MLP 2-5-1

Muito embora o algoritmo de retro-propagação tenha sido importante em sua época de criação, hoje em dia ele é tido como ineficiente. O algoritmo de Gauss-Newton pode convergir muito mais rapidamente, porém somente quando for viável uma aproximação quadrática para a função de erro. Do contrário, este método pode divergir.

Kenneth Levenberg e Donald Marquardt desenvolveram independentemente um algoritmo que consegue combinar a rápida convergência do Gauss-Newton e a estabilidade da retro-propagação de erro. Segundo Yu e Wilamowski (YU; WILAMOWSKI, 2011):

> "The basic idea of the Levenberg–Marquardt algorithm is that it performs a combined training process: around the area with complex curvature, the Levenberg–Marquardt algorithm switches to the steepest descent algorithm, until the local curvature is proper to make a quadratic approximation; then it approximately becomes the Gauss–Newton algorithm, which can speed up the convergence significantly. "[3]

Por estas características, ele é usado como opção de treinamento padrão no MATLAB. Ele é tido como o mais rápido para problemas de aproximação de função e previsão de séries temporais. Apesar disso pode ter o desempenho degradado se a rede for muito grande ou se for usado em problemas de reconhecimento de padrões. Para estes casos, algoritmos como gradiente conjugado escalado, apresentam melhor desempenho (BEALE; HAGAN; DEMUTH, 2012).

3 Tradução livre: A ideia básica do algoritmo Levenberg-Marquardt é que ele realiza um processo de treinamento combinado: ao redor da área de curvatura complexa, o algoritmo Levenberg-Marquardt troca para o algoritmo da descida mais íngreme, até que a curvatura seja adequada para fazer uma aproximação quadrática; então ele se torna uma aproximação do algoritmo Gauss-Newton, o que pode acelerar a convergência significativamente.

3 SOFTWARES PARA RNAS

3.1 Matlab

O MATLAB é um software da Mathworks Inc., que integra computação, visualização e programação em um ambiente flexível. A sua linguagem de programação é voltada para cientista e engenheiros, permitindo que estes obtenham resultados mais rapidamente do que em outras linguagens de programação de alto nível (DAVIS, 2011).

Ele possui um módulo para criar e utilizar RNAs. Com ele é possível criar redes supervisionadas; como as redes MLP, RBF e também redes não-supervisionadas, como os SOM e redes com camadas competitivas. Apresenta uma estrutura modular bastante flexível, além de facilitar tarefas como o pré e pós-processamento dos dados.

3.2 GNU Octave

O Octave é "um ambiente numérico em grande parte compatível com o Matlab"[4] segundo Hermoso (HERMOSO, 2012). Em 1992, John W. Eaton começou a trabalhar em um software que serviria de apoio ao livro sobre reações químicas que seu professor, James B. Rawlings, estava escrevendo. Eles logo perceberam que fazer rotinas específicas para os problemas do livro seria bastante limitante, assim optaram por criar uma ferramenta mais flexível. Na época, o Matlab era usado por um número cada vez maior dos colegas de John e James. Para facilitar a adoção do Octave por estes, resolveram que a sintaxe da ferramenta deveria ser em parte compatível com o Matlab. Criaram então uma linguagem de alto-nível, interpretada e focada em computação numérica. No início não existia tanta preocupação com essa compatibilidade. Com o passar do tempo, a pressão dos usuários para que os sistemas fossem compatíveis só fez crescer, mudando então a atitude dos desenvolvedores. Hoje em dia esse é um fator importante pois permite que cada vez mais código produzido para o Matlab, e por vezes livre, seja executado também no Octave (EDWOOD, 2012). Os criadores do Octave, desde o início fizeram com que o código fosse livre, para que qualquer pessoa consiga modificar o programa da maneira que julgar melhor.

Além do Octave, existe o Octave-Forge. Este último projeto gerencia o desenvolvimento de módulos para aplicações mais especializadas. Hoje em dia conta com mais de 90 módulos para as mais diversas aplicações como bioinformática, econometria, lógica nebulosa e também RNAs.

O módulo do Octave-Forge para RNAs é o nnet. Foi desenvolvido principalmente por Michael D. Schmid e está atualmente na versão 0.1.9.1. O módulo possibilita que sejam criadas redes neurais *feed-forward* usando o algoritmo de

4 Tradução livre de: "a free numerical environment mostly compatible with Matlab".

Levenberg-Marquardt no treinamento. É enxuto, contém apenas 14 funções. Com as funções `newff`, `train` e `sim` é possível criar, treinar e utilizar a RNA, respectivamente. O módulo oferece quatro opções de funções de ativação:

- `logsig`, função sigmoide logística;
- `purelin`, função puramente linear;
- `radbas`, função de base radial; e
- `tansig`, função sigmoide tangente hiperbólica.

Além destas, possui outras para pré e pós-processamento dos dados (padronização e des-padronização) e também algumas funções utilitárias.

4 IMPLEMENTAÇÃO DO MÓDULO

O *Neural Network Toolbox* do MATLAB é bastante flexível quanto às opções de construção e configuração das RNAs. Ele permite a configuração de coisas básicas como a quantidade de camadas, a quantidade de neurônios, as conexões entre as camadas e as funções de ativação. Além destas, é possível realizar o ajuste fino, configurando, entre outras, a função de agregação usada nos neurônios, atrasos nas entradas e até a função utilizada para ponderar as entradas e os pesos.Além da flexibilidade, a estrutura da rede é única para todas as aplicações. Ela comporta todas as opções, tanto das redes *feed-forward* mais simples, passando por redes com ligações de retro-alimentação (ou *feedback*) até os SOM, que apresentam diversas particularidades.

Contudo, a principal desvantagem do MATLAB e do módulo é o seu alto custo. Uma licença do núcleo principal custa alguns milhares de reais e este valor é incrementado a medida que são adicionados novos módulos. Para poder usar o módulo de RNAs, seria necessário gastar outros milhares de reais.

O MATLAB é bastante utilizado, além da pesquisa e indústria, também como material didático em universidades. Isso faz com que seja vantajoso criar algo compatível, pois permite-se que pessoas que já tenham conhecimento do MATLAB e do módulo de RNAs migrem para uma alternativa em software livre e gratuita, apesar de possivelmente ser limitada em funcionalidades e desempenho.

4.1 Modelagem

O primeiro passo para a construção do módulo foi realizar a modelagem das classes. Como citado anteriormente, um dos objetivos dos desenvolvedores do Octave é atingir o máximo de compatibilidade com o MATLAB possível. Tendo isso em mente, a modelagem se deu com base nas classes existentes no *Neural Network Toolbox* do MATLAB. Após a análise das classes, chegou-se na modelagem que pode ser vista na Figura 4.

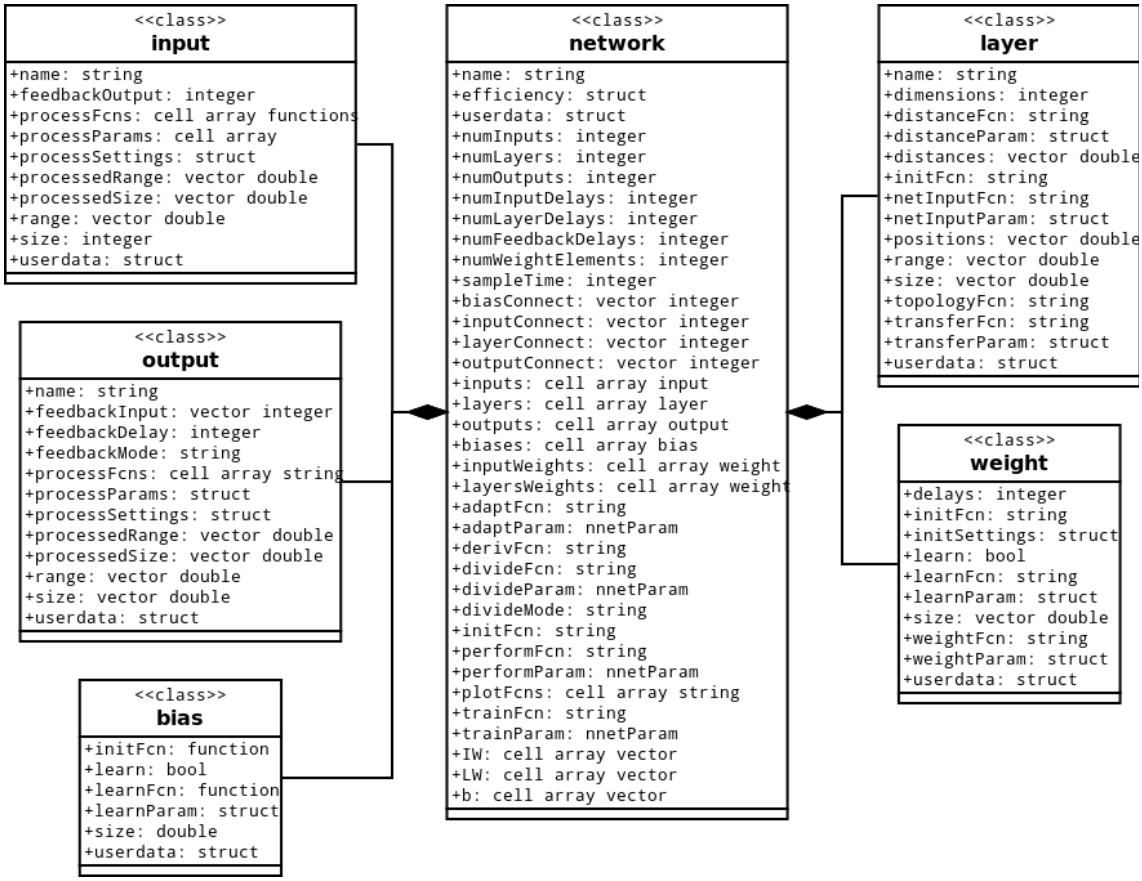

Figura 4: Modelagem das classes do módulo

Na modelagem do módulo foram criadas 6 classes. A principal delas é a classe *network*, que guarda as características gerais da rede. As outras classes (*input*, *output*, *bias*, *layer* e *weight*) são todas agregadas à primeira.

Apesar da grande quantidade de atributos (mais de 30), a arquitetura da rede é definida por apenas seis, sendo eles *numInputs*, *numLayers*, *biasConnect*, *inputConnect*, *layerConnect* e *outputConnect*. Os dois primeiros definem a quantidade de camadas de entrada e de camadas de processamento, respectivamente. Os outros quatro definem como são, respectivamente, as conexões dos viéses, das camadas de entrada, das camadas de processamento e das camadas de saída.

Outros atributos importantes para a rede são três funções: *trainFcn*, *performFcn* e *divideFcn*. Eles definem, respectivamente, o algoritmo de treinamento, a função de desempenho e como a rede vai usar os dados de entrada (a divisão dos dados será explicada mais adiante). Para a rede ficar funcional é preciso ainda definir o tamanho de cada camada (*layer.size*). Depois de pronta, os valores dos pesos são armazenados nos atributos *IW* (input weights, pesos de entrada), LW (*layer weights*, pesos das camadas) e b (*bias*, viés). Todas as informações básicas, com exceção do tamanho das camadas, ficam armazenadas em atributos da classe *network*. As outras classes tem atributos que vão conferir à rede flexibilidade e configurabilidade.

Dentre os atributos das classes dependentes, pode-se destacar alguns de maior interesse. As classes *input* e *output* tem os atributos *processFcn*, *processParams* e

processSettings, que permitem ao usuário especificar funções de pré e pós-processamento dos dados, visando melhorar o resultado no treinamento da rede. Nas classes *bias* e *weight*, o atributo *learn* permite que se definam pesos estáticos, que não serão afetados pelo treinamento. Na classe layer podemos ver como essa estrutura se adapta para representar os SOM, com atributos relativos à topologia (*topology*) e distância (*distanceFcn* e *distanceParam*), que são usados para a representação das camadas N dimensionais dos SOM.

4.2 Características e limitações das linguagens

Tanto o MATLAB quanto o Octave possuem suporte para programação orientada a objetos, porém ela não é feita da mesma maneira nas duas linguagens. Desde a versão 7.6 do MATLAB (versão R2008a de Março de 2008), para criar uma classe usa-se uma construção chamada *classdef*, que permite a definição dos atributos e dos métodos em um mesmo arquivo.

A funcionalidade do *classdef* já está em desenvolvimento no Octave, porém ainda não está pronta e, segundo os próprios desenvolvedores, esta é uma tarefa muito complexa e ainda está longe de ser concluída. Enquanto isso, o Octave ainda usa a maneira antiga de definição de classes do MATLAB (antes da versão 7.6). Para definir uma classe o desenvolvedor deve criar um diretório nomeado "`@nome_da_classe`" e dentro desse diretório colocar todos os arquivos Octave (.m) dos métodos da classe. O único arquivo obrigatório é o construtor, que tem o mesmo nome da classe (`@nome_da_classe/nome_da_classe.m`). Nesse arquivo o desenvolvedor deve colocar o código necessário para construir um objeto dessa classe.

Essa abordagem apresenta algumas limitações. Uma delas é o fato de o Octave não gerar métodos de acesso aos atributos dos objetos. Numa classe onde só existe o construtor, o usuário não consegue acessar nenhum dos atributos dos objetos. Para tal, o desenvolvedor da classe deve incluir duas outras funções no diretório da classe, são elas `subsref` e `subsasgn`. Elas servem para permitir a leitura e escrita aos atributos, respectivamente. Isso é ruim, pois o desenvolvedor deve escrever várias linhas de código para fazer algo que em várias outras linguagens de alto nível, e inclusive no MATLAB, já está disponível.

Outra diferença é a maneira de chamada dos métodos. Em outras linguagens de alto nível como Python, C++ e Java, para chamar um método de um objeto utiliza-se a sintaxe `objeto.método(parâmetros)`. No Octave, para fazer a mesma coisa, deve-se escrever `método(objeto,parâmetros)`, ou seja, chamar o método e passar como parâmetro o objeto.

A principal limitação, e mais séria, é como o Octave trata as funções gerais e os métodos das classes. Para executar as funções, ele as procura em uma lista de diretórios definida como *path* (caminho de pesquisa). Assim que encontra a primeira ocorrência do nome procurado, executa a função. Desta forma, todas as funções disponíveis, sejam elas do próprio Octave, de módulos do Octave-Forge ou métodos internos de classes ficam num mesmo escopo global. Isso faz com que o desenvolvedor tenha que se

preocupar que, em suas classes, o nome dos métodos devem ser únicos globalmente, não pode aparecer em nenhuma outra classe ou outro lugar do caminho de pesquisa. Caso ele não tome esse cuidado, e na sua classe tenha um método cujo nome já seja usado em outro lugar, quando esse nome for chamado, independente do contexto em que isso ocorra, o Octave vai executar a função que ele primeiro encontrar no caminho de pesquisa, não necessariamente a que se deseja executar.

O código do módulo foi criado e testado usando a versão 3.6.4 do Octave, última versão estável disponível (lançada em 21 de Fevereiro de 2013). Em testes realizados em uma versão anterior (3.2.4) foi encontrado um *bug*. Sempre que se tentava adicionar um objeto como membro de um *cell* (vetor que pode armazenar elementos de tipos variados), o objeto não era adicionado e a posição do vetor era apagada.

4.3 Desenvolvimento

Depois de feita a modelagem, iniciou-se o desenvolvimento das estruturas das classes. Visando evitar a necessidade de escrever as funções que permitem o acesso aos atributos, o que exigiria um esforço considerável, foi escrito um *script* em Python que gera o código necessário. Para tal ele recebe o nome da classe e os nomes dos atributos e com isso ele gera o diretório, o construtor e as funções `subsref` e `subsasgn`, deixando a classe pronta para o uso. Contudo, essa abordagem limita a aplicação de controle de visibilidade dos atributos. Da maneira como foi implementado, todos os atributos ficam públicos. Os casos de atributos privados ou atributos somente para leitura ainda devem ser tratados individualmente.

Após foi iniciado o desenvolvimento dos algoritmos de treinamento. Durante esse processo, foi necessário desenvolver diversas funções menores, como as funções de ativação, as funções de divisão de dados, funções de desempenho e funções para gerar gráficos usando o registro do treinamento. Sempre que possível, seguiu-se a convenção de nomenclatura das funções presentes no MATLAB, por motivos de facilitar a utilização deste módulo por quem já conheça o do MATLAB. Ttodas as funções criadas estarão listadas mais adiante.

A escolha do tipo de estrutura de RNAs e algoritmos a serem implementados levou em conta sua relevância e aplicabilidade. Dentre os diversos tipos de RNAs estudados, a que se mostrou mais difundida e utilizada são as redes *feed-forward*. Com este objetivo, partiu-se para a análise dos algoritmos de treinamento disponíveis. Foram escolhidos dois algoritmos de treinamento supervisionado, o algoritmo de retro-propagação por descida em gradiente e o algoritmo de LM. O primeiro foi escolhido devido à sua importância histórica e até educacional, pois foi o primeiro algoritmo de treinamento supervisionado para redes de múltiplas camadas. O segundo foi escolhido por apresentar bom desempenho em problemas de aproximação de funções e previsão de séries temporais (YU; WILAMOWSKI, 2011).

O algoritmo de retro-propagação foi implementado com base em (HAYKIN, 2001). Esse algoritmo está no arquivo *traingd.m* (descida em gradiente, do inglês *gradient descent).*

O algoritmo de LM foi implementado aproveitando-se o código do módulo *nnet* já citado, apenas fazendo alterações para adaptá-lo à nova estrutura e algumas melhorias pontuais, como mais informações no registro de treino. Este algoritmo pode ser visto no arquivo *trainlm.m*.

Observou-se que ambos os algoritmos são bastante sensíveis à inicialização dos pesos. Em testes verificou-se que se os pesos foram inicializados com valores no intervalo [0,1] a taxa de convergência pode ser até 10 vezes menor do que se for utilizado o intervalo [-1,1].

Em ambos os algoritmos de treinamento, foram implementados mais de um critério de parada. No algoritmo de retro-propagação foram implementados o erro que se deseja atingir (desempenho da rede), o número máximo de iterações (épocas) e o número máximo de iterações na qual o desempenho da rede piora, desempenho que é avaliado no conjunto de validação. Para o LM, além destes estão disponíveis o gradiente mínimo, fator MU máximo e tempo máximo de execução. O gradiente mínimo serve para evitar que o algoritmo continue executando sem ganhos significativos na função de desempenho, o fator μ (mi) que diz o quanto o método de aproxima do método de Newton ou do e o tempo máximo serve para limitar o tempo de execução do algoritmo. Esses critérios estão disponíveis também no módulo do MATLAB e estão respaldados na literatura (HAYKIN, 2001) (RUSSEL; NORVIG, 2003).

O método `train` é usado para treinar a rede, ele recebe como parâmetro um objeto da classe *network*, um conjunto de vetores de entrada e um conjunto de vetores alvo. Dependendo do atributo *trainFcn* da rede, o algoritmo selecionado é executado e retorna uma rede com os pesos modificados e um registro do treinamento. Esse registro contém informações sobre as configurações usadas no treinamento (erro que se desejava atingir, máximo de iterações, entre outras) e também informações sobre o treinamento, como motivo da parada, número de épocas, o desempenho da rede em cada época. Informações que são usadas para visualizar o desempenho da rede ao longo do treinamento.

O atributo *divideFcn* também é relevante no processo de treinamento. Ele define se a rede vai ou não separar os dados em conjuntos de treinamento, validação e teste e como acontecerá essa separação. O conjunto de treinamento contém os dados que serão mostrados à rede e é com base neles que a rede será treinada. O conjunto de validação serve como uma parada antecipada, visando evitar o problema de sobre-treinamento e aumentar a capacidade de generalização da rede. O terceiro conjunto, o de teste, serve para avaliar o desempenho da rede em dados desconhecidos, sendo assim possível avaliar a capacidade de generalização do modelo. Foram implementadas três opções, *dividetrain*, *dividerand* e *divideblock*. A primeira (*dividetrain*) utiliza todo o conjunto de dados como conjunto de treinamento, não usando os conjuntos de validação e de teste. Usando alguma das outras opções, os conjuntos de entrada e de alvos serão divididos nos três conjuntos mencionados, usando as proporções definidas no atributo *divideParam*. O método *divideblock* divide os dados em ordem, separando a primeira porção para teste, a segunda para validação e o restante para treino. Já o método *dividerand* irá dividir os dados em ordem aleatória. Vale ressaltar que, caso se quisesse realizar essa divisão utilizando o módulo nnet do Octave-Forge, ela deveria ser feita manualmente antes de submeter os dados à rede. Agora basta definir o método e os parâmetros que o objeto da RNA trata isso

automaticamente.

A função de desempenho utilizada no treinamento também é configurável através do atributo *performFcn*. Para esse foram implementadas duas opções, o erro quadrático médio (*mse*, do inglês *mean squared error*) e a soma dos erros quadráticos (*sse*, do inglês *sum squared error*).

Como funções de ativação, foram implementadas duas opções, a função linear e a função sigmoide (ambas mostradas na figura 2). Essas funções foram escolhidas pois são diferenciáveis, do contrário não poderiam ser utilizadas em conjunto com os algoritmos de treinamento disponíveis. Além da diferenciabilidade, a função sigmoide foi escolhida bastante citada na literatura (HAYKIN, 2001) (RUSSEL; NORVIG, 2003) (KOVÁCS, 2002). A função linear também foi escolhida por ser recomendada para a camada de saída, em casos onde é interessante que a RNA possa ter sinal de saída fora dos intervalos limitados [0,1] (da função sigmoide) ou [-1,1] (da função tangente hiperbólica.

O código foi todo escrito em inglês pois será submetido ao projeto Octave-Forge para que seja incluído dentre os pacotes oficiais do Octave. Todo o material original desenvolvido será disponibilizado sob a Licença Pública Geral GNU versão 3 ou GNU GPLv3 (do inglês *General Public License)*. O código do módulo *nnet*, de Michael Schimd está licenciado sob a GPLv2 e posteriores, sendo assim compatível. O código do NETLAB está sob uma licença própria que permite o uso e modificação, desde que se mantenha a nota de *copyright* original e não se utilize o nome dos autores ou instituição de origem sem permissão explícita.

5 TESTES E ESTUDOS DE CASO

Com o intuito de avaliar o *software* produzido, serão realizados estudos de caso utilizando o ferramental do Matlab e o módulo produzido. Além dos estudos de caso, que serão usados para comparação, serão criados alguns exemplos simples, que servirão de auxílio aos utilizadores do módulo.

Como demonstração do MLP, será usado o caso clássico em que a rede é treinada para aproximar a função XOR. Para este exemplo serão criados dois códigos distintos, um usando o treinamento por retro-propagação clássico e outro por LM.

5.1 Teste 1: XOR usando Gradiente Descendente

O algoritmo de gradiente descendente foi aplicado no problema clássico do XOR (ou exclusivo), onde a rede recebe duas entradas binárias, e deve decidir se o resultado será 0 ou 1. O mesmo conjunto de dados foi testado utilizando o módulo desenvolvido e o do MATLAB. Para ambos os casos, foram utilizados os mesmos parâmetros de treino: limite de 1000 épocas, taxa de aprendizagem fixa em 1 e objetivo de erro (MSE) 0,01. Foram criadas 100 redes em cada *software*, para minimizar o efeito da aleatoriedade na inicialização da RNA e possibilitar uma análise mais adequada. Os gráficos a seguir (figuras 5 e 6) mostram o erro (MSE) versus a quantidade de épocas do treinamento. Para facilitar a visualização foi aplicado o logaritmo natural (ln) nos erros. O ln teve o efeito de diminuir a influência dos maiores valores, permitindo observar com maior clareza as épocas em que as RNAs alcançam o objetivo de erro.

É possível notar que no caso do MATLAB (figura 5), a convergência acontece a partir de 120 épocas, a última convergência acontecendo em 763 épocas. No gráfico do Octave (figura 6), pode-se ver que o mínimo de épocas para a convergência é maior, agora 186. O mesmo acontece com o limite superior, que agora é 992 épocas.

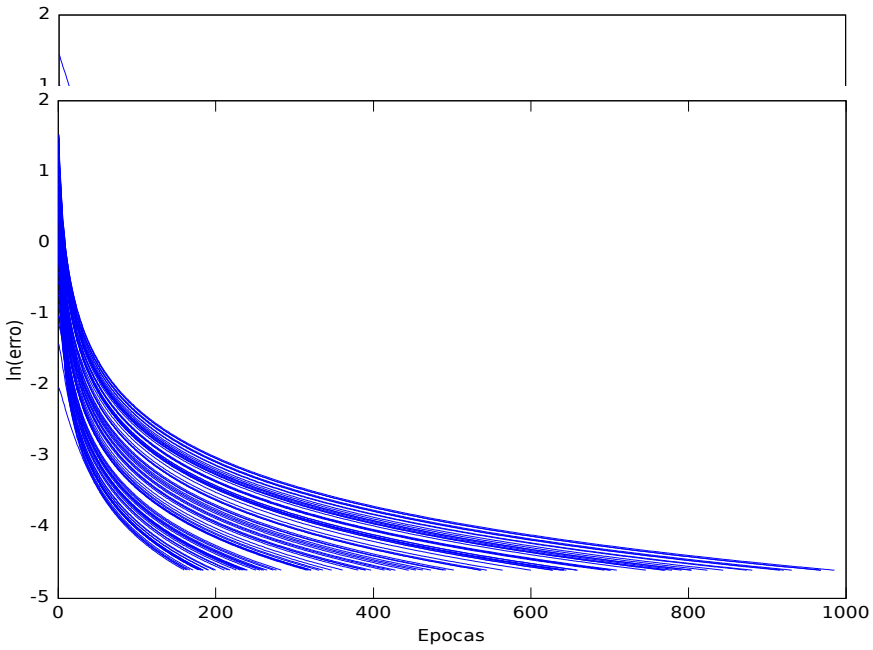

Figura 6: ln(Erro) (MSE) x Épocas - Octave

Na tabela 1 pode-se ver a taxa de convergência, a média de iterações e a média de tempo para convergir. A taxa de convergência é definida como o número de RNAs que atingiram o objetivo de erro, mas como foram executados 100 testes de cada, os valores já estão em porcentagens.

Tabela 1: Comparação entre as implementações do gradiente descendente

Software utilizado	Taxa de convergência (%)	Média de iterações	Média de tempo (s)
Octave	100	458,12	2,76
MATLAB	100	365,37	1,20

A análise dos gráficos das figuras 5 e 6, juntamente com a tabela 1 permite dizer que ambas as implementações tem a mesma taxa de convergência. Esse último fator influencia diretamente a média de tempo para convergência.

5.2 Teste 2: XOR usando Levenberg-Marquardt

Assim como o gradiente descendente, o algoritmo Levenberg-Marquardt foi aplicado no problema do XOR. O mesmo conjunto de dados foi testado utilizando o módulo desenvolvido, o nnet e o MATLAB. Para todos os casos, foram utilizados os mesmos parâmetros de treino: limite de 100 épocas, taxa de aprendizagem fixa em 1 e objetivo de erro (MSE) 0,01. Assim como no teste anterior, foram criadas 100 redes em cada *software*, para minimizar o efeito da aleatoriedade na inicialização da RNA e possibilitar uma análise mais adequada. Os gráficos mostram o logaritmo natural do erro versus a quantidade de épocas do treinamento.

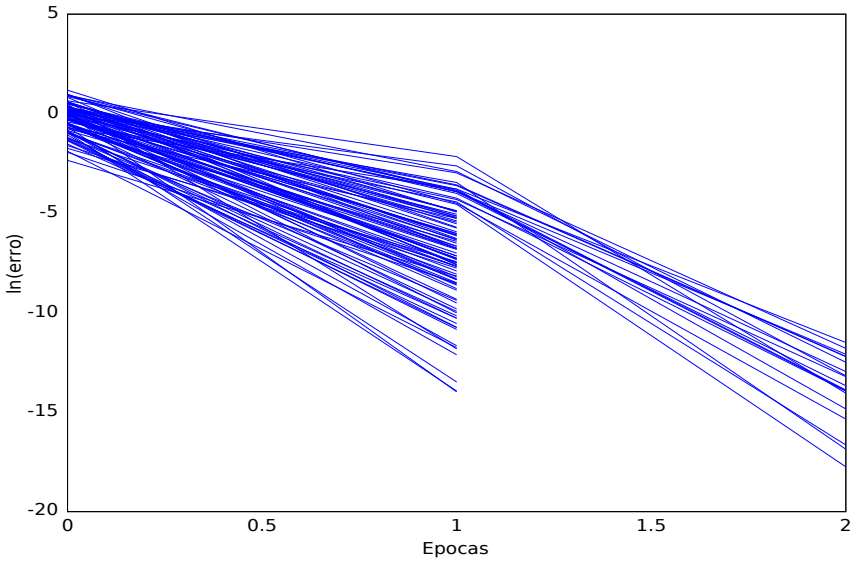

Figura 7: ln(Erro) (MSE) x Épocas - MATLAB

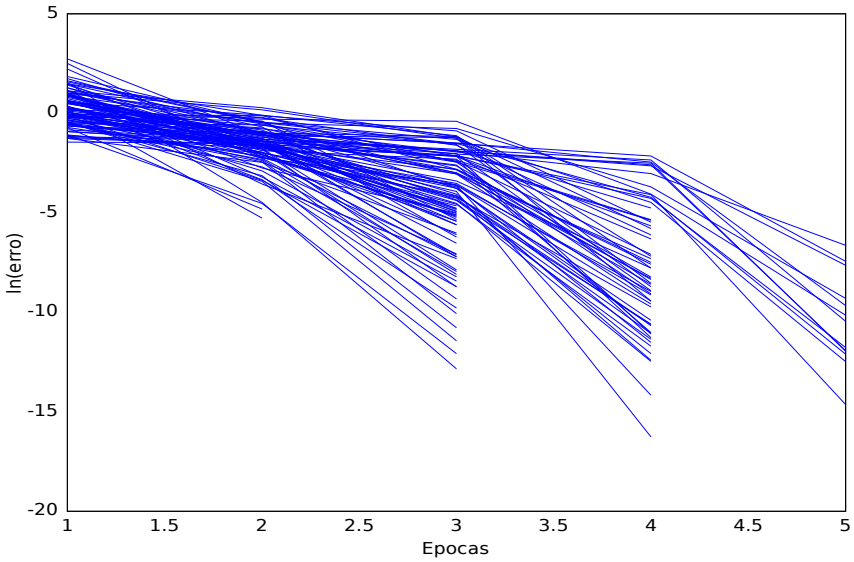

Figura 8: ln(Erro) (MSE) x Épocas - nnet

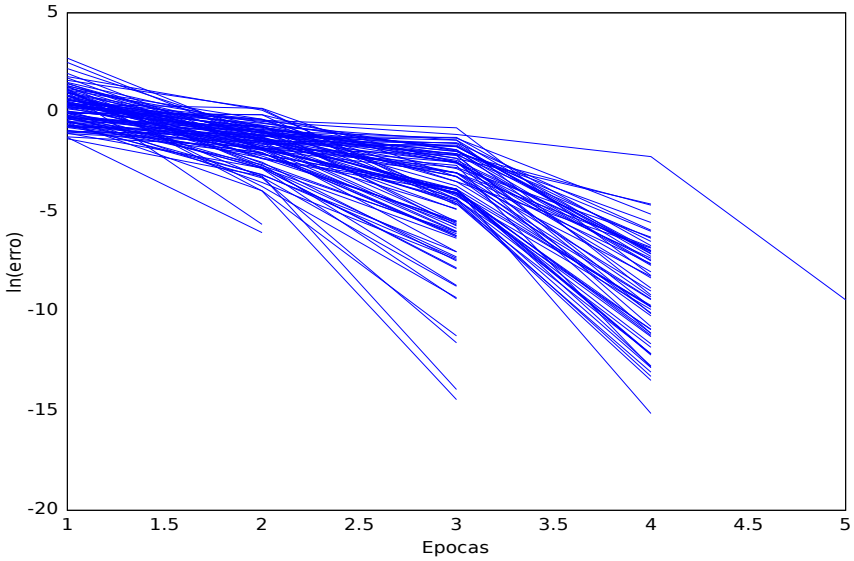

Figura 9: ln(Erro) (MSE) x Épocas - Octave

Tabela 2: Comparação entre as implementações do Levenberg-Marquardt

Software utilizado	Taxa de convergência (%)	Média de iterações	Média de tempo (s)
Octave	100	2,54	0,02
nnet	100	3,69	n/a
MATLAB	100	1,19	0,01

Analisando os gráficos das figuras 7, 8 e 9 juntamente com a tabela 2, a primeira coisa que se percebe é a diferença de duas ordens de grandeza entre as quantidades médias de iterações entre o gradiente descendente e o Levenberg-Marquardt. Pode-se ver, novamente, que todas as implementações atingiram taxa de convergência de 100%. As taxas de convergência estão próximas, novamente o MATLAB um pouco menor. Apesar da taxa de convergência da implementação deste trabalho ser menor que a do nnet, isso foi um caso específico desta reprodução do experimento. Durante a adaptação do código, não foi corrigida nem otimizada nenhuma funcionalidade relacionada ao treinamento, apenas foi melhorado o registro dos fatores do treinamento.

5.3 Estudo de caso: Previsão de energia usando MLP

O módulo também foi utilizado para um problema utilizando dados reais. O

módulo foi utilizado para a construção de RNA para realizar a previsão de energia elétrica residencial mensal de um ano a frente com base em variáveis explicativas diversas. Este trabalho foi desenvolvido dentro do projeto de Pesquisa e Desenvolvimento Celesc/IDESTI nº 5697-0210/2011, executado pelo Instituto de Capacitação, Pesquisa e Desenvolvimento Institucional de Gestão Social de Tecnologia de Informação (IDESTI) e com financiamento da Celesc Distribuição S.A. Neste, busca-se aplicar o produto desenvolvido na resolução de problemas reais.

Os dados de energia elétrica utilizados foram cedidos pela Centrais Elétricas de Santa Catariana (CELESC). Estes representam o consumo mensal dos clientes da classe residencial. A energia possui uma tendência crescente, o que pode ser um problema para as RNAs, assim, a série temporal foi estacionada dividindo o seu valor pela média móvel centrada de 24 meses. Além disto, foram usadas 14 variáveis explicativas: 11 variáveis binárias, uma para cada mês de Fevereiro a Dezembro; proporção de dias úteis no mês; proporção de dias no horário de verão no mês e um mapeamento da relação da temperatura ambiente com a energia utilizando um polinômio. Cada variável é uma série temporal de 10 anos e meio (126 medições), com observações mensais (Janeiro de 2001 até Junho de 2011). Dessas séries, o último ano foi usado como conjunto de teste, ou seja, esse ano não vai ser usado no treinamento.

Para encontrar uma boa arquitetura, foram testadas diversas configurações. Percebeu-se que as RNAs com apenas uma camada oculta apresentam melhores resultados. Assim, foram testadas redes com 1 a 20 neurônios na camada oculta e, cada uma dessas arquiteturas foi testada 10 vezes, para permitir a observação do desempenho da arquitetura, diminuindo a influência da aleatoriedade. Nas figuras 11 e 10, o eixo X representa o número de neurônios na camada oculta e o eixo Y representa o erro, podendo ser MSE ou MAPE (erro médio percentual absoluto, do inglês *mean absolute percentage error*).

Apesar da função de desempenho utilizada no treinamento ser o MSE, para a avaliação será usado o MAPE (erro médio percentual absoluto, do inglês *mean absolute percentage error*) mostrado nas figuras 11 e 10.

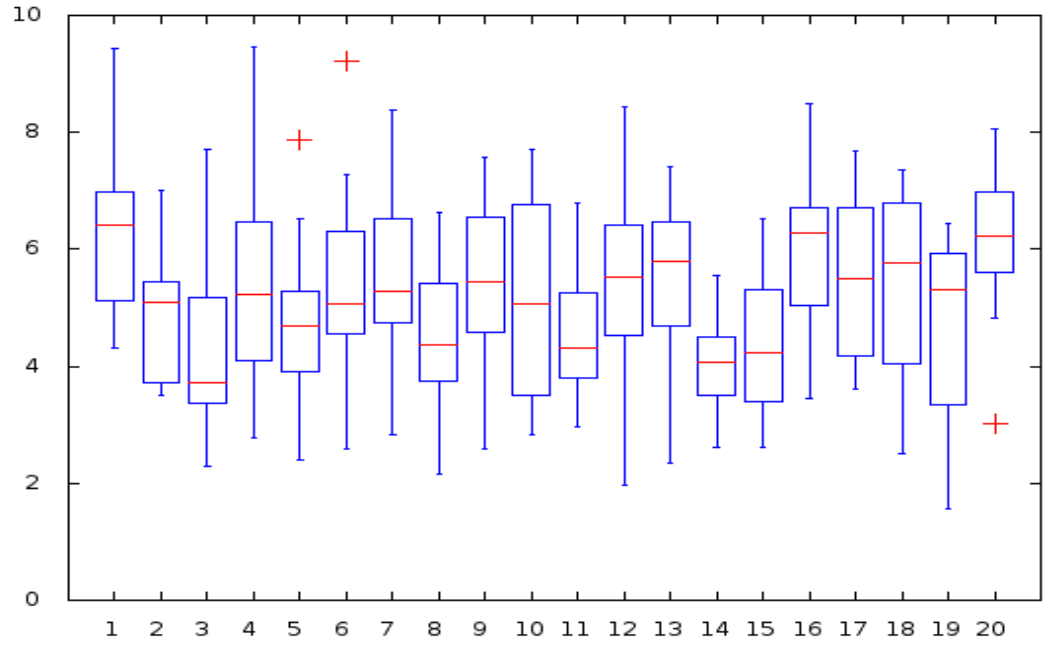

Figura 10: MAPE x Quantidade de neurônios na camada oculta - Previsão

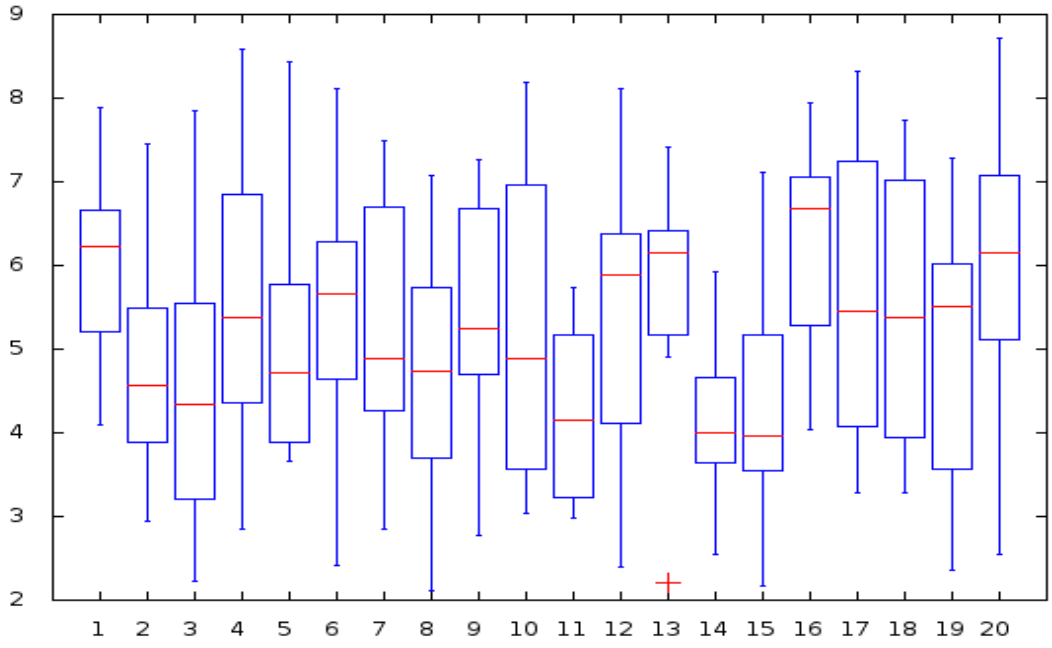

Figura 11: MAPE x Quantidade de neurônios na camada oculta - Treinamento

Estes gráficos permitem, além de avaliar os desempenho das RNAs, avaliar a estabilidade de arquitetura, o quanto ela é influenciada pela aleatoriedade dos parâmetros. Pode-se perceber que os erros na figura 11 são, no geral, menores que na figura 10. Isso acontece porque a figura 11 apresenta o desempenho das RNAs no conjunto de dados que foi usado no treinamento. A segunda figura (10) apresenta o desempenho das RNAs em um conjunto de dados desconhecido, o que nos permite

avaliar a capacidade de generalização da rede.

Agora a análise terá foco na arquitetura com 14 neurônios na camada oculta, por apresentar a maior estabilidade. Os dados de desempenho podem ser vistos na tabela 3 abaixo.

Tabela 3: Desempenho (MAPE) da RNA com 14 neurônios ocultos[5]

Conjunto de dados	*Mínimo*	*Máximo*	*Mediana*	*Variabilidade*
Treinamento	2,55	5,93	3,99	3,39
Teste	2,61	5,55	4,07	2,93

Esta arquitetura apresentou um bom desempenho na previsão, chegando a pouco mais de 2% no mínimo, 4% na mediana, assumindo previsões perfeitas das variáveis explicativas. Desempenho que mostra que a implementação é robusta para ser usada em situações reais. Em especial, este caso apresentou desempenho similar no conjunto de treinamento e de teste.

5 Valores arredondados.

6 CONCLUSÕES

O módulo *nnet* do Octave-Forge, na sua última versão disponibiliza para os usuários a utilização de redes MLP com o algoritmo de treinamento Levenberg-Marquardt. O objetivo principal do trabalho é expandir as opções de arquiteturas e algoritmos de redes neurais artificiais disponíveis no ambiente GNU Octave. Esse objetivo foi alcançado, pois além daquele, está disponível o algoritmo de descida em gradiente.

O módulo conta com funções para criar redes do tipo MLP, sendo configurável a quantidade e as conexões entre as camadas. A classe criada pode comportar facilmente outros tipos de RNAs, permitindo que novas funcionalidades sejam adicionadas sem que seja necessário modificar a classe, apenas fazer com que os atributos sejam utilizados da maneira correta. Um ponto forte dessa implementação é o nível de compatibilidade alcançado, que pôde ser verificado nos testes, onde foi usado exatamente o mesmo código para criar, treinar as RNAs e gerar os gráficos e as análises com base nos registros de treinamento. Além de possibilitar que os dois módulos sejam utilizados com a mesma interface dos métodos, todo o código desenvolvido pode ser utilizado tanto no Octave como no MATLAB (testes feitos no Octave 3.6.4 e MATLAB 7.12.0 – R2011a).

Os testes mostraram que, apesar de ter desempenho inferior às opções já existentes, o módulo pode ser usado tanto para fins didáticos como para a resolução de problema reais.

6.1 Trabalhos futuros

Foi visto que a inicialização aleatória dos pesos é muito importante para o resultado final. Dessa maneria, um trabalho importante seria melhorar essa inicialização, implementando possivelmente um algoritmo como o de Nguyen-Widrow, que é o atualmente utilizado pelo MATLAB. Além desse, seria interessante implementar algum algoritmo de treinamento não-supervisionado, o que aumentaria o rol de situações em que o módulo poderia ser aplicado.

7 REFERÊNCIAS BIBLIOGRÁFICAS

AZEVEDO, F. et. al. . **Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain**. The Journal of comparative neurology, v. , n. 5, p. 532-541, 2009.

BEALE, M.; HAGAN, M.; DEMUTH, H.. **Neural Network Toolbox**. 1 ed. Mathworks, 2012. 420 p.

BITTENCOURT, G.. **Inteligência Artificial: ferramentas e teorias**. 1 ed. Editora UFSC, 1998. p.

DAVIS, T.A.. **MATLAB Primer**. 8 ed. Taylor & Francis, 2011. p.

EGGERMONT, 1998: EGGERMONT, J., Rule-extraction and learning in the BP-SOM architecture, 1998

GROSSBERG, S.. **Nonlinear difference-differential equations in prediction and learning theory**. Proceedings of the National Academy of Sciences of the United States of America, v. , n. 4, p. 1329-1334, 1967.

HAYKIN, S.. **Redes Neurais: Princípios e Práticas**. 2 ed. Bookman, 2001. p.

HERMOSO, 2012: HERMOSO, J. G., What is Octave?, 2012

JONES, T.. **Artificial Intelligence: A Systems Approach**. 1 ed. Jones and Bartlett Publishers, Inc., 2008. p.

KOVÁCS, Z. L.. **Redes Neurais Artificiais**. ed. Livraria da Física, 2002. p.

McCulloch, W.;PITTS, W. **A logical calculus of the ideas immanent in nervous activity**. The bulletin of mathematical biophysics, v. , n. 5, p. 115-133, 1943.

MCCULLOCH, W.;PITTS, W. **Artificial Intelligence: A Systems Approach**. 1 ed. Jones and Bartlett Publishers, Inc., 2008. p.

MINSKY, M.; PAPERT, S. A.. **Perceptrons: An Introduction to Computational Geometry**. ed. MIT Press, 1969. p.

ORAVEC, M.; PAVLOVIČOVÁ, J.. **Face Recognition Methods Based on FeedforwardNeural Networks, Principal Component Analysisand Self-Organizing Map**. Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on, v. , n. 1, p. , 2007.

RUSSEL, S. J.; NORVIG, P.. **Artificial Intelligence: A Modern Approach**. 2 ed. Pearson Education, 2003. p.

WIDROW, B.. **Generalization and information storage in networks of adaline 'neurons'**. , v. , n. , p. , 1962.

YU, H.; WILAMOWSKI, B. M., Levenberg-Marquardt Trainning, 2011

EDWOOD, 2012: , GNU Octave: An interview with John W. Eaton and Jordi Gutiérrez, 2012

8 ANEXOS

8.1 Anexo 1: Código fonte do módulo desenvolvido

```
./@output/subsref.m
function ret = subsref(obj, sub)
if isempty(sub)
 error('@output/subsref: missing index');
end

switch sub.type
case '.'
 attribute = sub.subs;
 switch attribute
 case 'name'
    ret = obj.name;
 case 'feedbackInput'
    ret = obj.feedbackInput;
 case 'feedbackDelay'
    ret = obj.feedbackDelay;
 case 'feedbackMode'
    ret = obj.feedbackMode;
 case 'processFcns'
    ret = obj.processFcns;
 case 'processParams'
    ret = obj.processParams;
 case 'processSettings'
    ret = obj.processSettings;
 case 'processedRange'
    ret = obj.processedRange;
 case 'processedSize'
```

```
            ret = obj.processedSize;
        case 'range'
            ret = obj.range;
        case 'size'
            ret = obj.size;
        case 'userdata'
            ret = obj.userdata;
        otherwise
            error("@output/subsref: invalid property '%s'",
attribute);
        end
       otherwise
        error("@output/subsref:   invalid   property   '%s'",
attribute);
       end

       end
       ======================
       ./@output/subsasgn.m
       function ret = subsasgn(obj, sub, rhs)
       if isempty(sub)
        error('@output/subsasgn: missing index');
       end

       ret = obj;

       switch sub.type
       case '.'
        attribute = sub.subs;
        switch attribute
        case 'name'
            ret.name = rhs;
        case 'feedbackInput'
```

```
            ret.feedbackInput = rhs;
        case 'feedbackDelay'
            ret.feedbackDelay = rhs;
        case 'feedbackMode'
            ret.feedbackMode = rhs;
        case 'processFcns'
            ret.processFcns = rhs;
        case 'processParams'
            ret.processParams = rhs;
        case 'processSettings'
            ret.processSettings = rhs;
        case 'processedRange'
            ret.processedRange = rhs;
        case 'processedSize'
            ret.processedSize = rhs;
        case 'range'
            ret.range = rhs;
        case 'size'
            ret.size = rhs;
        case 'userdata'
            ret.userdata = rhs;
        otherwise
            error("@output/subsasgn: invalid property '%s'",
attribute);
        end
       otherwise
        error("@output/subsasgn:  invalid  property  '%s'",
attribute);
       end

       end
       ======================
       ./@output/display.m
```

```
function display(obj)
 disp(sprintf('%s object', class(obj)))
 disp(struct(obj))
end

======================
./@output/output.m
function object = output()
object.name = 'Output';
object.feedbackInput = [];
object.feedbackDelay = 0;
object.feedbackMode = 'none';
object.processFcns = [];
object.processParams = cell(1,0);
object.processSettings = cell(1,0);
object.processedRange = [];
object.processedSize = 0;
object.range = [];
object.size = 0;
object.userdata = struct('note','Put your custom information here.');

object = class(object, 'output');
end

======================
./@layer/layer.m
function object = layer()
object.name = 'Layer';
object.dimensions = 0;
object.distanceFcn = '';
object.distanceParam = 'No Neural Function Parameters';
```

```
      object.distances = [];
      object.initFcn = 'initwb';
      object.netInputFcn = 'netsum';
      object.netInputParam    =    'No    Neural    Function
Parameters';
      object.positions = [];
      object.range = [];
      object.size = 0;
      object.topologyFcn = '';
      object.transferFcn = 'tansig';
      object.transferParam    =    'No    Neural    Function
Parameters';
      object.userdata  =  struct('note',  'Put  your  custom
information here.');

      object = class(object, 'layer');
      end

      ======================
      ./@layer/subsref.m
      function ret = subsref(obj, sub)
      if isempty(sub)
       error('@layer/subsref: missing index');
      end

      switch sub.type
      case '.'
       attribute = sub.subs;
       switch attribute
       case 'name'
          ret = obj.name;
       case 'dimensions'
          ret = obj.dimensions;
```

```
        case 'distanceFcn'
            ret = obj.distanceFcn;
        case 'distanceParam'
            ret = obj.distanceParam;
        case 'distances'
            ret = obj.distances;
        case 'initFcn'
            ret = obj.initFcn;
        case 'netInputFcn'
            ret = obj.netInputFcn;
        case 'netInputParam'
            ret = obj.netInputParam;
        case 'positions'
            ret = obj.positions;
        case 'range'
            ret = obj.range;
        case 'size'
            ret = obj.size;
        case 'topologyFcn'
            ret = obj.topologyFcn;
        case 'transferFcn'
            ret = obj.transferFcn;
        case 'transferParam'
            ret = obj.transferParam;
        case 'userdata'
            ret = obj.userdata;
        otherwise
            error("@layer/subsref:  invalid  property  '%s'",
attribute);
        end
       otherwise
        error("@layer/subsref:   invalid   property   '%s'",
attribute);
```

```
end

end
======================
./@layer/subsasgn.m
function ret = subsasgn(obj, sub, rhs)
if isempty(sub)
 error('@layer/subsasgn: missing index');
end

ret = obj;

switch sub.type
case '.'
 attribute = sub.subs;
 switch attribute
 case 'name'
    ret.name = rhs;
 case 'dimensions'
    ret.dimensions = rhs;
 case 'distanceFcn'
    ret.distanceFcn = rhs;
 case 'distanceParam'
    ret.distanceParam = rhs;
 case 'distances'
    ret.distances = rhs;
 case 'initFcn'
    ret.initFcn = rhs;
 case 'netInputFcn'
    ret.netInputFcn = rhs;
 case 'netInputParam'
    ret.netInputParam = rhs;
```

```
        case 'positions'
            ret.positions = rhs;
        case 'range'
            ret.range = rhs;
        case 'size'
            ret.size = rhs;
        case 'topologyFcn'
            ret.topologyFcn = rhs;
        case 'transferFcn'
            ret.transferFcn = rhs;
        case 'transferParam'
            ret.transferParam = rhs;
        case 'userdata'
            ret.userdata = rhs;
        otherwise
            error("@layer/subsasgn: invalid property '%s'",
attribute);
        end
      otherwise
        error("@layer/subsasgn:  invalid  property  '%s'",
attribute);
      end

      end
      ======================
      ./@layer/display.m
      function display(obj)
        disp(sprintf('%s object', class(obj)))
        disp(struct(obj))
      end

      ======================
      ./@layer/netsum.m
```

```
function outputs = netsum(inputs)
% expects a cell array, containing several vectors
% outputs a elementwise sum of them
outputs = inputs{1};
inputs(1) = [];

for vector=inputs
 outputs = outputs+vector{1};
end

end

======================
./@weight/subsref.m
function ret = subsref(obj, sub)
if isempty(sub)
 error('@weight/subsref: missing index');
end

switch sub(1).type
case '.'
 attribute = sub(1).subs;
 switch attribute
 case 'delays'
    ret = obj.delays;
 case 'initFcn'
    ret = obj.initFcn;
 case 'initSettings'
    ret = obj.initSettings;
 case 'learn'
    ret = obj.learn;
 case 'learnFcn'
```

```
            ret = obj.learnFcn;
        case 'learnParams'
            ret = obj.learnParams;
        case 'size'
            ret = obj.size;
        case 'weightFcn'
            ret = obj.weightFcn;
        case 'weightParam'
            ret = obj.weightParam;
        case 'userdata'
            ret = obj.userdata;
        otherwise
            error("@weight/subsref: invalid property '%s'",
attribute);
        end
        sub(1) = [];
        if length(sub)
            ret = subsref(ret,sub);
        end
      otherwise
        error('@weight/subsref: invalid subscript reference
operator');
      end

      end
      ======================
      ./@weight/subsasgn.m
      function ret = subsasgn(obj, sub, rhs)
      if isempty(sub)
        error('@weight/subsasgn: missing index');
      end

      ret = obj;
```

```
        switch sub(1).type
        case '.'
         attribute = sub(1).subs;
         switch attribute
         case 'delays'
            ret.delays = rhs;
         case 'initFcn'
            ret.initFcn = rhs;
         case 'initSettings'
            ret.initSettings = rhs;
         case 'learn'
            ret.learn = rhs;
         case 'learnFcn'
            ret.learnFcn = rhs;
         case 'learnParams'
            ret.learnParams = rhs;
         case 'size'
            ret.size = rhs;
         case 'weightFcn'
            ret.weightFcn = rhs;
         case 'weightParam'
            ret.weightParam = rhs;
         case 'userdata'
            ret.userdata = rhs;
         otherwise
            error("@weight/subsasgn: invalid property '%s'",
attribute);
         end
         sub(1) = [];
         if length(sub)
            ret = subsasgn(ret,sub,rhs);
```

```
       end
      otherwise
       error("@weight/subsasgn:      invalid      subscript
assignment operator");
      end

      end
      ======================
      ./@weight/display.m
      function display(obj)
       disp(sprintf('%s object', class(obj)))
       disp(struct(obj))
      end

      ======================
      ./@weight/weight.m
      function object = weight()
      object.delays = 0;
      object.initFcn = '';
      object.initSettings = '';
      object.learn = true;
      object.learnFcn = '';
      object.learnParams = 'No Neural Function Parameters';
      object.size = [0 0];
      object.weightFcn = 'dotprod';
      object.weightParam = 'No Neural Function Parameters';
      object.userdata  =  struct('note',  'Put  your  custom
information here.');
      object = class(object, 'weight');
      end

      ======================
      ./@weight/dotprod.m
```

```
function output = dotprod(x, y)
 if size(x,2) ~= size(y,1)
    error('@weight/dotprod:   weights   and   inputs
dimensions do not match')
 end

 output = x*y;
end

======================
./@bias/subsref.m
function ret = subsref(obj, sub)
if (isempty (sub))
          error ("bias: missing index");
endif

switch sub(1).type
case '.'
 ret = builtin('subsref',obj, sub);
otherwise
 error("@neuralnet/subsref:     invalid     subscript
reference operator");
end

end

======================
./@bias/subsasgn.m
function ret = subsasgn(obj, sub, rhs)
if (isempty (sub))
  error ("bias: missing index");
end
```

```
      ret = obj

      switch (sub.type)
      case '.'
       field = sub.subs;
       switch field
       case 'initFcn'
          ret.initFcn = rhs;
       case 'learn'
          ret.learn = rhs;
       case 'learnFcn'
          ret.learnFcn = rhs;
       case 'size'
          ret.size = rhs;
       case 'userdata'
          ret.userdata = rhs;
       otherwise
          error          ("@bias/subsasgn:          invalid
property \'%s\'", field);
       endswitch
      otherwise
       error ("invalid subscript type");
      endswitch

      end

      ======================
      ./@bias/display.m
      function display(obj)
       disp(sprintf('%s object', class(obj)))
       disp(struct(obj))
      end
```

```
======================
./@bias/bias.m
function object = bias()
 object.initFcn = '';
 object.learn = true;
 object.learnFcn = '';
 object.learnParam = 'No Neural Function Parameters';
 object.size = 0; %equal to layer(i) size
 object.userdata = struct('note', 'Put your custom
information here.');

 object = class(object, 'bias');
endfunction

======================
./feedforwardnet.m
function net = feedforwardnet(layerSizes)
%% Define initial parameters
if isempty(who('layerSizes'))
 layerSizes = [5];
end

numInputs = 1;
numLayers = length(layerSizes)+1;

% there are bias connected to each layer
biasConnect = ones(numLayers,1);

% input is only connected to the first layer
inputConnect = [1;zeros(numLayers-1,1)];
```

```
      % each layer receives a connection from the previous
and sends a connection to
      % the next one
      [x,y] = meshgrid(1:numLayers);
      layerConnect = y == x+1;

      % only the last layer is connected to the output
      outputConnect = [zeros(1,numLayers-1) 1];

      %% Create the network and define layer sizes
      net = network(numInputs, numLayers, biasConnect,...
                                 inputConnect,  layerConnect,
outputConnect);

      layers = net.layers;
      % last layer will have size determined by the targets
dimensions during init
      for i = 1:numLayers-1
       layers{i}.size = layerSizes(i);
      end
      layers{numLayers}.transferFcn = 'purelin';
      net.layers = layers;

      % set 'default' options
      net.trainFcn = 'trainlm';
      net.performFcn = 'mse';
      net.divideFcn = 'dividetrain';

      end

      ======================
      ./@network/purelinderiv.m
      function out = purelinderiv(in)
```

```
 out = ones(size(in));
end

======================
./@network/tansigderiv.m
function out = tansigderiv(in)
 out = tansig(in);
 out = 1-(out.*out);
end

======================
./@network/traingd.m
function [net,TR] = traingd(net,TR,trainV,valV,testV)
% TODO: because of 'lw' in line 73, only works if net
has 2 or more layers
 % if requested, return the default parameters
 if strcmp(net,'pdefaults')
    net = default_parameters();
    TR = [];
    return
 end

 % auxiliary indexes
 ilfIndex = 1;
 outputIndex = 2;

 % auxiliary stuff
 fails = 0;
 layers = net.layers;
 lw = net.LW;
 iw = net.IW;
 trainPerf = Inf;
```

```
trainErrors = [];

performFcn = net.performFcn;
maxEpochs = net.trainParam.epochs;

% train parameters
lr = net.trainParam.lr;
maxFail = net.trainParam.max_fail;

% indexes
INPUT_IDX = 1;
TARGET_IDX = 2;

% store performance at epoch 0
for i = 1:size(trainV,2)
    input = trainV{INPUT_IDX}{i};
    target = trainV{TARGET_IDX}{i};
    trainOutputs = sim(net,input);
    trainError = trainOutputs - target;
    trainErrors = [trainErrors trainError];
end
perf = feval(performFcn,trainErrors);
TR.epoch = [TR.epoch 0];
TR.perf = [TR.perf perf];
TR.gradient = [TR.gradient 0];
TR.time = [TR.time 0];
TR.num_epochs = 0;

epoch = 1;
t0 = clock();
while epoch <= maxEpochs
```

```
          for i = 1:size(trainV{1},2)
              input = trainV{INPUT_IDX}{i};
              target = trainV{TARGET_IDX}{i};

              % forward propagation
              [trainOutputs,        ilfAndOutputs]        =
sim(net,input);

              trainError = target - trainOutputs;
              trainErrors = [trainErrors trainError];

              %% backward propagation
              %% OUTPUT LAYER
              layerIndex  =  net.numLayers;  %  assumes  the
last layer is the output layer
              % getting function names and some data
              layerTransferFcn                              =
layers{layerIndex}.transferFcn;
              layerDerivFcn    =    strcat(layerTransferFcn,
'deriv');
              layerIncomingWeightsIdx                       =
~cellfun('isempty',lw(layerIndex,:));
              layerIncomingWeights                          =
lw{layerIndex,layerIncomingWeightsIdx};

              % actual learning
              gradient        =        trainError        .*
feval(layerDerivFcn,...
                                  ilfAndOutputs{ilfIndex,la
yerIndex});
              dw      =      lr      *      gradient'      *
ilfAndOutputs{outputIndex,layerIndex-1};
              dw = dw';
              if    ~    all(size(layerIncomingWeights)    ==
size(dw))
```

```
                TR.stop = 'Weight error';
                return
            end
            if any(isnan(gradient))
                TR.stop = 'Gradient error';
                return
            end
            newWeights = layerIncomingWeights + dw;

            % storing the new weights
            lw{layerIndex,layerIncomingWeightsIdx}      =
newWeights;
            net.LW = lw;

            %% HIDDEN LAYERS
            for layerIndex = net.numLayers-1:-1:2
                layerIndex = layerIndex;
                % getting function names and some data
                layerTransferFcn                            =
layers{layerIndex}.transferFcn;
                layerDerivFcn  =  strcat(layerTransferFcn,
'deriv');
                layerIncomingWeightsIdx                     =
~cellfun('isempty',lw(layerIndex,:));
                layerIncomingWeights                        =
lw{layerIndex,layerIncomingWeightsIdx};

                % actual learning
                %gradient                                   =
feval(layerDerivFcn,ilfAndOutputs{ilfIndex,layerIndex})
*...
                %      sum(gradients*weights)
                       % TODO: THIS
                %dw     =     lr     *     gradient     .*
ilfAndOutputs{outputIndex,layerIndex};
```

```
                newWeights = layerIncomingWeightsi;% + dw;

                % storing the new weights
                lw{layerIndex,layerIncomingWeightsIdx}    =
newWeights;
                net.LW = lw;
            end

            %% INPUT LAYER
            layerIndex = 1;
            % getting function names and some data
            layerTransferFcn                                =
layers{layerIndex}.transferFcn;
            layerDerivFcn    =    strcat(layerTransferFcn,
'deriv');
            layerIncomingWeightsIdx                         =
~cellfun('isempty',iw(layerIndex,:));
            layerIncomingWeights                            =
iw{layerIndex,layerIncomingWeightsIdx};

            % actual learning
            %gradient = feval(layerDerivFcn,...
            %         ilfAndOutputs{ilfIndex,layerIndex})
*...
            %         sum(gradients*weights)
            %dw      =      lr      *      gradient      .*
ilfAndOutputs{outputIndex,layerIndex};
            newWeights = layerIncomingWeights;% + dw;

            % storing the new weights
            iw{layerIndex,layerIncomingWeightsIdx}       =
newWeights;
            net.IW = iw;
        end
```

```
            trainPerf = feval(performFcn,trainErrors);

            %% test stop criteria
            if trainPerf <= net.trainParam.goal
               TR.stop = 'Goal error achieved.';
               return
            end

            if trainPerf <= TR.best_perf
               TR.best_perf = trainPerf;
               fails = 0;
            else
               if fails == 0
                  bestNet = net;
               end
               fails = fails + 1;
               if fails == maxFail
                  TR.stop = 'Validation stop.';
                  net = bestNet;
                  return
               end
            end

            if net.trainParam.showCommandLine
               printf("Epoch    %d;    Train    Performance:
%g\n",epoch,trainPerf);
            end

            elapsedTime = etime(clock(),t0);

            TR.epoch = [TR.epoch epoch];
            TR.perf = [TR.perf trainPerf];
```

```
    TR.gradient = [TR.gradient gradient];
    TR.time = [TR.time elapsedTime];
    TR.num_epochs = epoch;

    epoch = epoch + 1;
    %fig = plotPerform(net,TR,1);
 end

 TR.stop = 'Maximum epochs reached.';
end

function parameters = default_parameters()
 % maximum epochs
 parameters.epochs = 1000;
 % performance goal
 parameters.goal = 0.01;
 % not yet used
 parameters.showCommandLine = false;
 % not used
 parameters.showWindow = false;
 % learning rate
 parameters.lr = 1;
 % maximum validation checks
 parameters.max_fail = 6;
 % minimum gradient
 parameters.min_grad = 1e-5;
 % not yet used
 parameters.show = 50;
 % maximum time
 parameters.time = Inf;
end
```

```
      ======================
      ./@network/dividetrain.m
      function          [trainInd,valInd,testInd]          =
dividetrain(Q,trainRatio,valRatio,testRatio)
      % if requested, return the default parameters
      if strcmp(Q,'pdefaults')
       trainInd = [];
       valInd = [];
       testInd = [];
       return
      end

      trainInd = 1:Q;
      valInd = [];
      testInd = [];
      end


      ======================
      ./@network/subsref.m
      function ret = subsref(obj, sub)
      if isempty(sub)
       error('@neuralnet/subsref: missing index');
      end

      switch sub(1).type
      case '()'
       vectors = sub(1).subs{1};
       ret = sim(obj,vectors);
      case '.'
       ret = builtin('subsref',obj, sub);
      otherwise
```

```
      error("@neuralnet/subsref: invalid subscript
reference operator");
    end

    end

    ======================
    ./@network/__divideindexes__.m
    function [trainInd,valInd,testInd] =
__divideindexes__(indexes,...
                                    trainRatio,valRati
o,testRatio)
    totalLength = length(indexes);
    valLength = ceil(totalLength*valRatio);
    testLength = ceil(totalLength*testRatio);
    trainLength = totalLength - valLength - testLength;

    first = 1;
    last = trainLength;
    trainInd = indexes(first:last);

    first = last+1;
    last = first+valLength-1;
    valInd = indexes(first:last);

    first = last+1;
    last = first+testLength-1;
    testInd = indexes(first:last);

    end

    ======================
    ./@network/subsasgn.m
```

```
function ret = subsasgn(obj, sub, rhs)
if isempty(sub)
 error('@neuralnet/subsasgn: missing index');
end

ret = obj;

switch sub(1).type
case '.'
 attribute = sub(1).subs;
 switch attribute
 case 'numInputs'
    %TODO: update inputConnect, inputs and IW
    ret.numInputs = rhs;
 case 'numLayers'
    %TODO:  update  {bias,input,layer,output}Connect,
biases,
    %      {input,layer}Weights, outputs, IW, LW and
b
    ret.numLayers = rhs;
 case 'numOutputs'
    error('@neuralnet/subasgn.m:    numOutputs    is
read-only');
 case 'divideFcn'
    ret.divideFcn = rhs;
    ret.divideParam = feval(rhs,'pdefaults');
 case 'performFcn'
    ret.performFcn = rhs;
    ret.performParam = feval(rhs,'pdefaults');
 case 'trainFcn'
    ret.trainFcn = rhs;
    [ret.trainParam,~] = feval(rhs,'pdefaults');
 otherwise
```

```
            ret = builtin('subsasgn', obj, sub, rhs);
            return
        end
        sub(1) = [];
        if length(sub)
            ret = subsasgn(ret,sub,rhs);
        end
       otherwise
        error("@neuralnet/subsasgn:    invalid    subscript
assignment operator");
       end

       end

       ======================
       ./@network/display.m
       function display(obj)
        disp(sprintf('%s object', class(obj)))
        disp(struct(obj))
       end

       ======================
       ./@network/__divide_datasets__.m
       function          [trainSet,valSet,testSet]          =
__divide_datasets__(net,P,T,...
                                                        train
ingRecord)

       [trainInd,         valInd,         testInd]         =
feval(net.divideFcn,length(P));

       trainingRecord.trainInd = trainInd;
       trainingRecord.valInd = valInd;
```

```
      trainingRecord.testInd = testInd;

      trainInputSet = P(:,trainInd);
      valInputSet = P(:,valInd);
      testInputSet = P(:,testInd);

      trainTargetSet = T(:,trainInd);
      valTargetSet = T(:,valInd);
      testTargetSet = T(:,testInd);

      trainSet      =      {num2cell(trainInputSet,1)      ;
num2cell(trainTargetSet,1)};
      valSet        =        {num2cell(valInputSet)        ;
num2cell(valTargetSet)};
      testSet       =       {num2cell(testInputSet)       ;
num2cell(testTargetSet)};

      end

      ======================
      ./@network/isposint.m
      ##    Copyright    (C)    2005    Michel    D.    Schmid
<michaelschmid@users.sourceforge.net>
      ##
      ##
      ##   This   program   is   free   software;   you   can
redistribute it and/or modify it
      ## under the terms of the GNU General Public License
as published by
      ## the Free Software Foundation; either version 2, or
(at your option)
      ## any later version.
      ##
      ## This program is distributed in the hope that it
will be useful, but
```

```
      ## WITHOUT ANY WARRANTY; without even the implied
warranty of
      ## MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.  See the GNU
      ## General Public License for more details.
      ##
      ## You should have received a copy of the GNU General
Public License
      ## along with this program; see the file COPYING.  If
not, see
      ## <http://www.gnu.org/licenses/>.

      ## -*- texinfo -*-
      ## @deftypefn {Function File} {} @var{f} =
isposint(@var{n})
      ## @code{isposint} returns true for positive integer
values.
      ##
      ## @example
      ##   isposint(1)   # this returns TRUE
      ##   isposint(0.5) # this returns FALSE
      ##   isposint(0)   # this also return FALSE
      ##   isposint(-1)  # this also returns FALSE
      ## @end example
      ##
      ##
      ## @end deftypefn

      ## Author: Michel D. Schmid

      function f = isposint(n)

        ## check range of input arguments
        error(nargchk(1,1,nargin))
```

```
        ## check input arg
        if (length(n)>1)
          error("Input argument must not be a vector, only
scalars are allowed!")
        endif

        f = 1;
        if ( (!isreal(n)) | (n<=0) | (round(n) != n) )
          f = 0;
        endif



      endfunction

      %!shared
      %! disp("testing isposint")
      %!assert(isposint(1)) # this should pass
      %!assert(isposint(0.5),0) # should return zero
      %!assert(isposint(-1),0) # should return zero
      %!assert(isposint(-1.5),0) # should return zero
      %!assert(isposint(0),0) # should return zero
      %!fail("isposint([0 0])","Input argument must not be
a vector, only scalars are allowed!")
      %!fail("isposint('testString')","Input argument must
not be a vector, only scalars are allowed!")



      ======================
      ./@network/divideblock.m
      function          [trainInd,valInd,testInd]          =
divideblock(Q,trainRatio,valRatio,testRatio)
      % if requested, return the default parameters
```

```
      if strcmp(Q,'pdefaults')
       trainInd = __default_divide_parameters__();
       valInd = [];
       testInd = [];
       return
      end

      indexes = 1:Q;
      [trainInd,         valInd,         testInd]         =
__divideindexes__(indexes,trainRatio,valRatio,testRatio);

      end

      ======================
      ./@network/plotPerform.m
      function fig = plotPerform(net,tr,fig)
      fig = figure(fig);
      plot(tr.epoch,tr.perf)
      %title('Neural network performance')
      %xlabel('epoch')
      %ylabel(net.performFcn)
      end

      ======================
      ./@network/__trainlm.m
      ##    Copyright    (C)    2006    Michel    D.    Schmid
<michaelschmid@users.sourceforge.net>
      ##
      ##
      ##   This   program   is   free   software;   you   can
redistribute it and/or modify it
      ## under the terms of the GNU General Public License
as published by
```

```
      ## the Free Software Foundation; either version 2, or
(at your option)
      ## any later version.
      ##
      ## This program is distributed in the hope that it
will be useful, but
      ## WITHOUT ANY WARRANTY; without even the implied
warranty of
      ##  MERCHANTABILITY  or  FITNESS  FOR  A  PARTICULAR
PURPOSE.  See the GNU
      ## General Public License for more details.
      ##
      ## You should have received a copy of the GNU General
Public License
      ## along with this program; see the file COPYING.  If
not, see
      ## <http://www.gnu.org/licenses/>.


      ## -*- texinfo -*-
      ##  @deftypefn  {Function  File}  {}[@var{netOut}]  =
__trainlm
(@var{net},@var{mInputN},@var{mOutput},@var{[]},@var{[]},@
var{VV})
      ## A neural feed-forward network will be trained with
@code{__trainlm}
      ##
      ## @example
      ## [netOut,tr,out,E] = __trainlm(net,mInputN,mOutput,
[],[],VV);
      ## @end example
      ## @noindent
      ##
      ## left side arguments:
      ## @example
      ## netOut: the trained network of the net structure
@code{MLPnet}
```

```
      ## tr :
      ## out:
      ## E  : Error
      ## @end example
      ## @noindent
      ##
      ## right side arguments:
      ## @example
      ##  net       : the untrained network, created with
@code{newff}
      ## mInputN: normalized input matrix
      ## mOutput: output matrix
      ## []     : unused parameter
      ## []     : unused parameter
      ## VV     : validize structure
      ## out:
      ## E  : Error
      ## @end example
      ## @noindent
      ##
      ##
      ## @noindent
      ## are equivalent.
      ## @end deftypefn


      ## @seealso{newff,prestd,trastd}


      ## Author: Michel D. Schmid


      ## Comments: see in "A neural network toolbox for
Octave User's Guide" [4]
      ##   for variable naming... there have inputs or
targets only one letter,
```

```
      ## e.g. for inputs is P written. To write a program,
this is stupid, you can't
      ## search for 1 letter variable... that's why it is
written here like Pp, or Tt
      ## instead only P or T.


      function [net,TR] = trainlm(net,TR,trainV,valV,testV,
IM,P,T)
       if strcmp(net,'pdefaults')
          net = default_parameters();
          TR = [];
          return
       end
      %function [net] = trainlm(net,Im,Pp,Tt,VV)

        ## check range of input arguments
        error(nargchk(5,5,nargin))

        ## Initialize
        ##------------

        ## get parameters for training
        epochs   = net.trainParam.epochs;
        goal     = net.trainParam.goal;
        maxFail  = net.trainParam.max_fail;
        minGrad  = net.trainParam.min_grad;
        mu       = net.trainParam.mu;
        muInc    = net.trainParam.mu_inc;
        muDec    = net.trainParam.mu_dec;
        muMax    = net.trainParam.mu_max;
        show     = net.trainParam.show;
        time     = net.trainParam.time;
```

```
        ## parameter checking
        checkParameter(epochs,goal,maxFail,minGrad,mu,\
                       muInc,muDec,muMax,show,time);

        ## Constants
        shortStr = "TRAINLM";    # TODO: shortStr is longer
as TRAINLM !!!!!!!!!!!
        stop = "";



        #startTime = clock(); # TODO: maybe this row can be
placed
                              # some rows later

        ## the weights are used in column vector format
        xx = __getx(net); # x is the variable with respect
to, but no
                                 # variables with only one
letter!!
        ## define identity matrix
        muI = eye(length(xx));

        startTime = clock();  # if the next some tests are
OK, I can delete
                               # startTime = clock(); 9 rows
above..

        ## calc performance of the actual net
        [perf,vE,Aa,Nn] = __calcperf(net,xx,Im,Tt);

        nLayers = net.numLayers;
        for iEpochs = 0:epochs # longest loop & one of the
stop criterias
          ve = vE{nLayers,1};
```

```
          ## calc jacobian
          ## Jj is jacobian matrix
          [Jj] = __calcjacobian(net,Im,Nn,Aa,vE);

          ## rerange error vector for jacobi matrix
          ve = ve(:);

           Jjve = (Jj' * ve); # will be used to calculate
the gradient

          normGradX = sqrt(Jjve'*Jjve);

          ## record training progress for later plotting
          ## if requested
          trainRec.perf(iEpochs+1) = perf;
          trainRec.mu(iEpochs+1) = mu;

          ## stoping criteria
                                    [stop,currentTime]       =
stopifnecessary(stop,startTime,perf,goal,\
                                    iEpochs,epochs,time,normGr
adX,minGrad,mu,muMax,\
                                    doValidation,maxFail);

          ## show train progress
          showtrainprogress(show,stop,iEpochs,epochs,time,c
urrentTime, \
            goal,perf,minGrad,normGradX,shortStr,net);

          ## show performance plot, if needed
            if !isnan(show) # if no performance plot is
needed
              ## now make it possible to define after how
much loops the
```

```
            ## performance plot should be updated
            if (mod(iEpochs,show)==0)
              plot(1:length(trainRec.perf),trainRec.perf);
        if (doValidation)
          hold on;

plot(1:length(trainRec.vperf),trainRec.vperf,"--g");
        endif
             endif
           endif # if !(strcmp(show,"NaN"))
       #    legend("Training","Validation");

           ## stop if one of the criterias is reached.
           if length(stop)
             break
           endif

           ## calculate DeltaX
           while (mu <= muMax)
             ## calculate change in x
             ## see [4], page 12-21
             dx = -((Jj' * Jj) + (muI*mu)) \ Jjve;

             ## add changes in x to actual x values (xx)
             x1 = xx + dx;
                ## now add x1 to a new network to see if
performance will be better
             net1 = __setx(net,x1);
             ## calc now new performance with the new net
             [perf1,vE1,Aa1,N1] = __calcperf(net1,x1,Im,Tt);

             if (perf1 < perf)
                   ## this means, net performance with new
```

```
weight values is better...
              ## so save the new values
              xx = x1;
              net = net1;
              Nn = N1;
              Aa = Aa1;
              vE = vE1;
              perf = perf1;

              mu = mu * muDec;
                if (mu < 1e-20)    # 1e-20 is properly the
hard coded parameter in MATLAB(TM)
                 mu = 1e-20;
              endif
              break
            endif
            mu = mu * muInc;
          endwhile

        endfor #for iEpochs = 0:epochs

      #=================================================
===
      #
      # additional functions
      #
      #=================================================
===
                                                    function
checkParameter(epochs,goal,maxFail,minGrad,mu,\
                      muInc, muDec, muMax, show, time)
          ## Parameter Checking
```

```
          ## epochs must be a positive integer
          if ( !isposint(epochs) )
            error("Epochs is not a positive integer.")
          endif

          ## goal can be zero or a positive double
          if ( (goal<0) | !(isa(goal,"double")) )
              error("Goal is not zero or a positive real
value.")
          endif

          ## maxFail must be also a positive integer
          if ( !isposint(maxFail) ) # this will be used, to
see if validation can
            # break the training
            error("maxFail is not a positive integer.")
          endif

           if (!isa(minGrad,"double")) | (!isreal(minGrad))
| (!isscalar(minGrad)) | \
            (minGrad < 0)
             error("minGrad is not zero or a positive real
value.")
          end

             ## mu must be a positive real value. this
parameter is responsible
            ## for moving from stepest descent to quasi
newton
              if ((!isa(mu,"double")) | (!isreal(mu)) |
(any(size(mu)) != 1) | (mu <= 0))
            error("mu is not a positive real value.")
          endif
```

```
          ## muDec defines the decrement factor
           if ((!isa(muDec,"double")) | (!isreal(muDec)) |
(any(size(muDec)) != 1) | \
               (muDec < 0) | (muDec > 1))
            error("muDec is not a real value between 0 and
1.")
          endif

          ## muInc defines the increment factor
            if (~isa(muInc,"double")) | (!isreal(muInc)) |
(any(size(muInc)) != 1) | \
            (muInc < 1)
             error("muInc is not a real value greater than
1.")
          endif

          ## muMax is the upper boundary for the mu value
            if (!isa(muMax,"double")) | (!isreal(muMax)) |
(any(size(muMax)) != 1) | \
            (muMax <= 0)
            error("muMax is not a positive real value.")
          endif

          ## check for actual mu value
          if (mu > muMax)
            error("mu is greater than muMax.")
          end

          ## check if show is activated
          if (!isnan(show))
         if (!isposint(show))
               error(["Show is not " "NaN" " or a positive
integer."])
            endif
```

```
          endif

          ## check at last the time argument, must be zero
or a positive real value
             if  (!isa(time,"double"))  |  (!isreal(time))  |
(any(size(time)) != 1) | \
            (time < 0)
              error("Time is not zero or a positive real
value.")
          end

        endfunction # parameter checking

      #
      #
----------------------------------------------------------
------------------
      #

                                                    function
showtrainprogress(show,stop,iEpochs,epochs,time,currentTim
e, \
                goal,perf,minGrad,normGradX,shortStr,net)

          ## check number of inputs
          error(nargchk(12,12,nargin));

          ## show progress
              if  isfinite(show)  &  (!rem(iEpochs,show)  |
length(stop))
              fprintf(shortStr);    # outputs the training
algorithm
            if isfinite(epochs)
              fprintf(", Epoch %g/%g",iEpochs, epochs);
            endif
```

```
            if isfinite(time)
                                fprintf(",   Time   %4.1f%
%",currentTime/time*100);      #  \todo:  Time  wird  nicht
ausgegeben
            endif
            if isfinite(goal)
                                  fprintf(",   %s   %g/
%g",upper(net.performFcn),perf,goal);   #   outputs   the
performance function
            endif
            if isfinite(minGrad)
                                fprintf(",   Gradient   %g/
%g",normGradX,minGrad);
            endif
            fprintf("\n")
            if length(stop)
              fprintf("%s, %s\n\n",shortStr,stop);
            endif
             fflush(stdout); # writes output to stdout as
soon as output messages are available
          endif
        endfunction


      #
      #
-------------------------------------------------------
-------------------
      #


                  function      [stop,currentTime]      =
stopifnecessary(stop,startTime,perf,goal,\
                               iEpochs,epochs,time,normGradX
,minGrad,mu,muMax,\
                        doValidation,maxFail)
```

```
          ## check number of inputs
          error(nargchk(14,14,nargin));

          currentTime = etime(clock(),startTime);
          if (perf <= goal)
            stop = "Performance goal met.";
          elseif (iEpochs == epochs)
            stop = "Maximum epoch reached, performance goal
was not met.";
          elseif (currentTime > time)
             stop = "Maximum time elapsed, performance goal
was not met.";
          elseif (normGradX < minGrad)
             stop = "Minimum gradient reached, performance
goal was not met.";
          elseif (mu > muMax)
              stop = "Maximum MU reached, performance goal
was not met.";
          endif
        endfunction


      #
============================================================
===========
      #
      # END additional functions
      #
      #
============================================================
===========


      endfunction


      ======================
      ./@network/mse.m
```

```
function perf = mse(E,Y,X)
if strcmp(E,'pdefaults')
 perf = default_parameters();
 return
end

if isempty(E)
 E = Y - X;
end


perf = mean(E.^2);

end

function parameters = default_parameters()
 parameters.regularization = 0;
 parameters.normalization = 'none';
 parameters.squaredWeighting = true;
end

======================
./@network/__getx.m
## Copyright (C) 2005 Michel D. Schmid <michaelschmid@users.sourceforge.net>
##
##
## This program is free software; you can redistribute it and/or modify it
## under the terms of the GNU General Public License as published by
## the Free Software Foundation; either version 2, or (at your option)
```

```
      ## any later version.
      ##
      ## This program  is distributed in the hope that it
will be useful, but
      ## WITHOUT ANY WARRANTY; without even the implied
warranty of
      ## MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.  See the GNU
      ## General Public License for more details.
      ##
      ## You should have received a copy of the GNU General
Public License
      ## along with this program ; see the file COPYING.
If not, see
      ## <http://www.gnu.org/licenses/>.


      ## -*- texinfo -*-
      ## @deftypefn {Function File} {} @var{x} = __getx
(@var{net})
      ## @code{__getx} will rerange the weights in one
columns vector.
      ##
      ##
      ## @noindent
      ## @end deftypefn



      ## Author: Michel D. Schmid


      function x = __getx(net)


        ## check number of inputs
        error(nargchk(1,1,nargin));
```

```
        ## check input args
        ## check "net", must be a net structure
        if !__checknetstruct(net)
            error("Structure doesn't seem to be a neural
network")
        endif

        ## inputs
        x = net.IW{1,1}(:);
        x = [x; net.b{1}(:)];

        nNumLayers = net.numLayers;
        for iLayers = 2:nNumLayers # 1 would be the input
layer

          ## layers
          x = [x; net.LW{iLayers,iLayers-1}(:)];
          x = [x; net.b{iLayers}(:)];

        endfor



      endfunction
      ======================
      ./@network/__calcperf.m
      ## Copyright (C) 2006 Michel D. Schmid     <email:
michaelschmid@users.sourceforge.net>
      ##
      ##
      ##   This program is free software; you can
redistribute it and/or modify it
      ## under the terms of the GNU General Public License
as published by
      ## the Free Software Foundation; either version 2, or
```

```
(at your option)
      ## any later version.
      ##
      ## This program is distributed in the hope that it will be useful, but
      ## WITHOUT ANY WARRANTY; without even the implied warranty of
      ## MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
      ## General Public License for more details.
      ##
      ## You should have received a copy of the GNU General Public License
      ## along with this program; see the file COPYING.  If not, see
      ## <http://www.gnu.org/licenses/>.


      ## -*- texinfo -*-
      ## @deftypefn {Function File} {}[@var{perf}, @var{Ee}, @var{Aa}, @var{Nn}] = __calcperf (@var{net},@var{xx},@var{Im},@var{Tt})
      ## @code{__calcperf} calculates the performance of a multi-layer neural network.
      ## PLEASE DON'T USE IT ELSEWHERE, it proparly won't work.
      ## @end deftypefn


      ## Author: Michel D. Schmid



      function [perf,Ee,Aa,Nn] = __calcperf(net,xx,Im,Tt)


        ## comment:
         ## perf, net performance.. from input to output through the hidden layers
```

```
        ## Aa, output values of the hidden and last layer
(output layer)
        ## is used for NEWFF network types

        ## calculate bias terms
        ## must have the same number of columns like the
input matrix Im
        [nRows, nColumns] = size(Im);
        Btemp = cell(net.numLayers,1); # Btemp: bias matrix
        ones1xQ = ones(1,nColumns);
        for i= 1:net.numLayers
          Btemp{i} = net.b{i}(:,ones1xQ);
        endfor

        ## shortcuts
        IWtemp = cell(net.numLayers,net.numInputs,1);# IW:
input weights ...
        LWtemp = cell(net.numLayers,net.numLayers,1);# LW:
layer weights ...
        Aa = cell(net.numLayers,1);# Outputs hidden and
output layer
        Nn = cell(net.numLayers,1);# outputs before the
transfer function
        IW = net.IW; # input weights
        LW = net.LW; # layer weights

        ## calculate the whole network till outputs are
reached...
        for iLayers = 1:net.numLayers

           ## calculate first input weights to weighted
inputs..
          ## this can be done with matrix calculation...
          ## called "dotprod"
          ## to do this, there must be a special matrix ...
```

```
          ## e.g.  IW = [1 2 3 4 5; 6 7 8 9 10] * [ 1 2 3;
4 5 6; 7 8 9; 10 11 12; 1 2 3];
          if (iLayers==1)
            IWtemp{iLayers,1} = IW{iLayers,1} * Im;
                      onlyTempVar   =   [IWtemp(iLayers,1)
Btemp(iLayers)];
          else
            IWtemp{iLayers,1} = [];
          endif

          ## now calculate layer weights to weighted layer
outputs
          if (iLayers>1)
            Ad = Aa{iLayers-1,1};
            LWtemp{iLayers,1} = LW{iLayers,iLayers-1} * Ad;
                      onlyTempVar   =   [LWtemp(iLayers,1)
Btemp(iLayers)];
          else
            LWtemp{iLayers,1} = [];
          endif

          Nn{iLayers,1} = onlyTempVar{1};
          for k=2:length(onlyTempVar)
            Nn{iLayers,1} = Nn{iLayers,1} + onlyTempVar{k};
          endfor

          ## now calculate with the transfer functions the
layer output
          switch net.layers{iLayers}.transferFcn
          case "purelin"
            Aa{iLayers,1} = purelin(Nn{iLayers,1});
          case "tansig"
            Aa{iLayers,1} = tansig(Nn{iLayers,1});
          otherwise
```

```
                          error(["Transfer   function:   "
net.layers{iLayers}.transferFcn " doesn't exist!"])
          endswitch

        endfor  # iLayers = 1:net.numLayers

        ## now calc network error
        Ee = cell(net.numLayers,1);

        for i=net.numLayers
          Ee{i,1} = Tt{i,1} - Aa{i,1};# Tt: target
          # Ee will be the error vector cell array
        endfor

        ## now calc network performance
        perf = feval(net.performFcn,Ee{2});

      endfunction

      ======================
      ./@network/__default_divide_parameters__.m
      function parameters = __default_divide_parameters__()
      parameters.trainRatio = 0.7;
      parameters.valRatio = 0.15;
      parameters.testRatio = 0.15;
      end

      ======================
      ./@network/sse.m
      function perf = sse(E,Y,X)
      if strcmp(E,'pdefaults')
       perf = default_parameters();
```

```
end

if isempty(E)
 E = Y - X;
end

perf = sum(E.^2);

end

function parameters = default_parameters()
 parameters.regularization = 0;
 parameters.normalization = 'none';
 parameters.squaredWeighting = true;
end

======================
./@network/trainlm.m
## Copyright (C) 2006 Michel D. Schmid <michaelschmid@users.sourceforge.net>
##
##
## This program is free software; you can redistribute it and/or modify it
## under the terms of the GNU General Public License as published by
## the Free Software Foundation; either version 2, or (at your option)
## any later version.
##
## This program is distributed in the hope that it will be useful, but
## WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
      ##  MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.  See the GNU

      ## General Public License for more details.

      ##

      ## You should have received a copy of the GNU General
Public License

      ## along with this program; see the file COPYING.  If
not, see

      ## <http://www.gnu.org/licenses/>.


      ## Author: Michel D. Schmid


      ## Comments: see in "A neural network toolbox for
Octave User's Guide" [4]

      ##   for variable naming... there have inputs or
targets only one letter,

      ## e.g. for inputs is P written. To write a program,
this is stupid, you can't

      ## search for 1 letter variable... that's why it is
written here like Pp, or Tt

      ## instead only P or T.


      function [net,TR] = trainlm(net,TR,trainV,valV, data)
       if strcmp(net,'pdefaults')
           net = default_parameters();
           TR = [];
           return
       end


       Im = data.Im;
       Tt = data.Tt;


        ## check range of input arguments
        error(nargchk(5,5,nargin))
```

```
        ## Initialize
        ##------------

        ## get parameters for training
        epochs   = net.trainParam.epochs;
        goal     = net.trainParam.goal;
        maxFail  = net.trainParam.max_fail;
        minGrad  = net.trainParam.min_grad;
        mu       = net.trainParam.mu;
        muInc    = net.trainParam.mu_inc;
        muDec    = net.trainParam.mu_dec;
        muMax    = net.trainParam.mu_max;
        show     = net.trainParam.show;
        time     = net.trainParam.time;

        ## parameter checking
        checkParameter(epochs,goal,maxFail,minGrad,mu,\
                       muInc,muDec,muMax,show,time);

        ## Constants
        shortStr = "TRAINLM";    # TODO: shortStr is longer
as TRAINLM !!!!!!!!!!
        stop = "";



        #startTime = clock(); # TODO: maybe this row can be
placed
                              # some rows later

        ## the weights are used in column vector format
        xx = __getx(net); # x is the variable with respect
to, but no
```

```
                                    # variables with only one
letter!!
        ## define identity matrix
        muI = eye(length(xx));

        startTime = clock();  # if the next some tests are
OK, I can delete
                                # startTime = clock(); 9 rows
above..

        ## calc performance of the actual net
        [perf,vE,Aa,Nn] = __calcperf(net,xx,Im,Tt);

        nLayers = net.numLayers;
        for iEpochs = 0:epochs # longest loop & one of the
stop criterias
          ve = vE{nLayers,1};
          ## calc jacobian
          ## Jj is jacobian matrix
          [Jj] = __calcjacobian(net,Im,Nn,Aa,vE);

          ## rerange error vector for jacobi matrix
          ve = ve(:);

           Jjve = (Jj' * ve); # will be used to calculate
the gradient

          normGradX = sqrt(Jjve'*Jjve);

          ## record training progress for later plotting
          ## if requested
       TR.epoch(iEpochs+1) = iEpochs+1;
          TR.perf(iEpochs+1) = perf;
          TR.mu(iEpochs+1) = mu;
```

```
      TR.time(iEpochs+1) = etime(clock(),startTime);


          ## stoping criteria
                                    [stop,currentTime]       =
stopifnecessary(stop,startTime,perf,goal,\
                                    iEpochs,epochs,time,normGr
adX,minGrad,mu,muMax,\
                                    maxFail);

          ## show train progress
          showtrainprogress(show,stop,iEpochs,epochs,time,c
urrentTime, \
            goal,perf,minGrad,normGradX,shortStr,net);

          ## stop if one of the criterias is reached.
          if length(stop)
         TR.stop = stop;
         TR.num_epochs = iEpochs;
            break
          endif

          ## calculate DeltaX
          while (mu <= muMax)
            ## calculate change in x
            ## see [4], page 12-21
            dx = -((Jj' * Jj) + (muI*mu)) \ Jjve;

            ## add changes in x to actual x values (xx)
            x1 = xx + dx;
               ## now add x1 to a new network to see if
performance will be better
            net1 = __setx(net,x1);
```

```
            ## calc now new performance with the new net
            [perf1,vE1,Aa1,N1] = __calcperf(net1,x1,Im,Tt);

            if (perf1 < perf)
                  ## this means, net performance with new
weight values is better...
              ## so save the new values
              xx = x1;
              net = net1;
              Nn = N1;
              Aa = Aa1;
              vE = vE1;
              perf = perf1;

              mu = mu * muDec;
                if (mu < 1e-20)    # 1e-20 is properly the
hard coded parameter in MATLAB(TM)
                  mu = 1e-20;
              endif
              break
            endif
            mu = mu * muInc;
          endwhile

        endfor #for iEpochs = 0:epochs

      #=================================================
===
      #
      # additional functions
      #
      #=================================================
===
```

```
                                                        function
checkParameter(epochs,goal,maxFail,minGrad,mu,\
                    muInc, muDec, muMax, show, time)
          ## Parameter Checking

          ## epochs must be a positive integer
          if ( !isposint(epochs) )
            error("Epochs is not a positive integer.")
          endif

          ## goal can be zero or a positive double
          if ( (goal<0) || !(isa(goal,"double")) )
              error("Goal is not zero or a positive real
value.")
          endif

          ## maxFail must be also a positive integer
          if ( !isposint(maxFail) ) # this will be used, to
see if validation can
            # break the training
            error("maxFail is not a positive integer.")
          endif

          if (!isa(minGrad,"double")) || (!isreal(minGrad))
|| (!isscalar(minGrad)) || \
            (minGrad < 0)
             error("minGrad is not zero or a positive real
value.")
          end

            ## mu must be a positive real value. this
parameter is responsible
            ## for moving from stepest descent to quasi
newton
```

```
            if  ((!isa(mu,"double"))  ||  (!isreal(mu))  ||
(any(size(mu)) != 1) || (mu <= 0))
              error("mu is not a positive real value.")
            endif

            ## muDec defines the decrement factor
            if ((!isa(muDec,"double")) || (!isreal(muDec)) ||
(any(size(muDec)) != 1) || \
                (muDec < 0) || (muDec > 1))
              error("muDec is not a real value between 0 and
1.")
            endif

            ## muInc defines the increment factor
            if (~isa(muInc,"double")) || (!isreal(muInc)) ||
(any(size(muInc)) != 1) || \
              (muInc < 1)
               error("muInc is not a real value greater than
1.")
            endif

            ## muMax is the upper boundary for the mu value
            if (!isa(muMax,"double")) || (!isreal(muMax)) ||
(any(size(muMax)) != 1) || \
              (muMax <= 0)
              error("muMax is not a positive real value.")
            endif

            ## check for actual mu value
            if (mu > muMax)
              error("mu is greater than muMax.")
            end

            ## check if show is activated
```

```
          if (!isnan(show))
         if (!isposint(show))
               error(["Show is not " "NaN" " or a positive
integer."])
            endif
          endif


          ## check at last the time argument, must be zero
or a positive real value
            if (!isa(time,"double")) || (!isreal(time)) ||
(any(size(time)) != 1) || \
            (time < 0)
              error("Time is not zero or a positive real
value.")
          end


        endfunction # parameter checking


      #
      #
------------------------------------------------------------
------------------
      #


                                                    function
showtrainprogress(show,stop,iEpochs,epochs,time,currentTim
e, \
                goal,perf,minGrad,normGradX,shortStr,net)


          ## check number of inputs
          error(nargchk(12,12,nargin));


          ## show progress
             if isfinite(show) && (!rem(iEpochs,show) ||
length(stop))
```

```
              fprintf(shortStr);    # outputs the training
algorithm
            if isfinite(epochs)
              fprintf(", Epoch %g/%g",iEpochs, epochs);
            endif
            if isfinite(time)
                                  fprintf(",   Time   %4.1f%
%",currentTime/time*100);      #  \todo:  Time  wird  nicht
ausgegeben
            endif
            if isfinite(goal)
                                     fprintf(",   %s   %g/
%g",upper(net.performFcn),perf,goal);   #   outputs   the
performance function
            endif
            if isfinite(minGrad)
                                  fprintf(",   Gradient   %g/
%g",normGradX,minGrad);
            endif
            fprintf("\n")
            if length(stop)
              fprintf("%s, %s\n\n",shortStr,stop);
            endif
             fflush(stdout); # writes output to stdout as
soon as output messages are available
          endif
        endfunction


      #
      #
------------------------------------------------------------
------------------
      #

                    function       [stop,currentTime]       =
```

```
stopifnecessary(stop,startTime,perf,goal,\
                                iEpochs,epochs,time,normGradX
,minGrad,mu,muMax,\
                        maxFail)

          ## check number of inputs
          error(nargchk(12,12,nargin));

          currentTime = etime(clock(),startTime);
          if (perf <= goal)
         stop = 'Goal error achieved.';
          elseif (iEpochs == epochs)
            stop = "Maximum epochs reached.";
          elseif (currentTime > time)
             stop = "Maximum time elapsed, performance goal
was not met.";
          elseif (normGradX < minGrad)
             stop = "Minimum gradient reached, performance
goal was not met.";
          elseif (mu > muMax)
              stop = "Maximum MU reached, performance goal
was not met.";
          endif
        endfunction

       function parameters = default_parameters()
          % maximum epochs (stop criterion)
          parameters.epochs = 1000;
          % performance goal (stop criterion)
          parameters.goal = 0.01;
          % defines if progress will be displayed at cmd
line
          parameters.showCommandLine = false;
          % not used
```

```
            parameters.showWindow = false;
            % learning rate
            parameters.lr = 1;
            % maximum validation checks (stop criterion)
            parameters.max_fail = 6;
            % minimum gradient (stop criterion)
            parameters.min_grad = 1e-5;
            % initial mu value
            parameters.mu = 0.001;
            % mu decrease factor
            parameters.mu_dec = 0.1;
            % mu increase factor
            parameters.mu_inc = 10;
            % maximum mu value (stop criterion)
            parameters.mu_max = 10000000000;
            % show frequency
            parameters.show = 25;
            % maximum time (stop criterion)
            parameters.time = Inf;
        endfunction

        #
=============================================================
===========
        #
        # END additional functions
        #
        #
=============================================================
===========

        endfunction
```

```
      ======================

      ./@network/__calcjacobian.m

      ## Copyright (C) 2006 Michel D. Schmid
<michaelschmid@users.sourceforge.net>

      ##

      ##

      ## This program is free software; you can
redistribute it and/or modify it

      ## under the terms of the GNU General Public License
as published by

      ## the Free Software Foundation; either version 2, or
(at your option)

      ## any later version.

      ##

      ## This program is distributed in the hope that it
will be useful, but

      ## WITHOUT ANY WARRANTY; without even the implied
warranty of

      ## MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.  See the GNU

      ## General Public License for more details.

      ##

      ## You should have received a copy of the GNU General
Public License

      ## along with this program; see the file COPYING.  If
not, see

      ## <http://www.gnu.org/licenses/>.


      ## -*- texinfo -*-

      ## @deftypefn {Function File} {}@var{Jj} =
__calcjacobian
(@var{net},@var{Im},@var{Nn},@var{Aa},@var{vE})

      ## This function calculates the jacobian matrix. It's
used inside the

      ## Levenberg-Marquardt algorithm of the neural
network toolbox.
```

```
      ## PLEASE DO NOT USE IT ELSEWEHRE, it proparly will
not work!
      ## @end deftypefn

      ## Author: Michel D. Schmid


      function [Jj] = __calcjacobian(net,Im,Nn,Aa,vE)

        ## comment:
        ## - return value Jj is jacobi matrix
         ##    for this calculation, see "Neural Network
Design; Hagan, Demuth & Beale page 12-45"


        ## check range of input arguments
        error(nargchk(5,5,nargin))

        ## get signals from inside the network
        bias  = net.b;

        ## calculate some help matrices
        mInputWeight = net.IW{1} * Im;
        nLayers = net.numLayers;
        for i=2:nLayers
          mLayerWeight{i,1} = net.LW{i,i-1} * Aa{i-1,1};
        endfor

         ## calculate number of columns and rows in jacobi
matrix
        ## firstly, number of columns
        a = ones(nLayers+1,1); # +1 is for the input
        a(1) = net.inputs{1}.size;
```

```
        for iLayers = 1:nLayers
          a(iLayers+1) = net.layers{iLayers}.size;
        endfor
        nColumnsJacobiMatrix = 0;
        for iLayers = 1:nLayers
                    nColumnsJacobiMatrix    =    (a(iLayers)
+1)*a(iLayers+1) + nColumnsJacobiMatrix;
        endfor
        ## secondly, number of rows
        ve = vE{nLayers,1};
        nRowsJacobiMatrix = length(ve(:));


                            ##           FIRST           STEP
-----------------------------------------------------
          ##  calculate  the  neuron  outputs  without  the
transfer function
         ## - n1_1 = W^1*a_1^0+b^1: the ^x factor defined
the xth train data set
        ##   the _x factor defines the layer
        ## **********  this datas should be hold in Nn
        ## **********  should be calculated in "__calcperf"
        ## **********  so Nn{1} means hidden layer
         ## **********  so Nn{2} means second hidden layer
or output layer
        ## **********  and so on ...
                       ##        END        FIRST        STEP
-----------------------------------------------

        ## now we can rerange the signals ... this will be
done only for
        ## matrix calculation ...
        [nRowsError nColumnsError] = size(ve);
         errorSize = size(ve(:),1); # this will calculate,
if only one row
```

```
         # of errors exist... in other words... two rows
will be reranged to
        # one row with the same number of elements.
         rerangeIndex = floor([0:(errorSize-1)]/nRowsError)
+1;
        nLayers = net.numLayers;

        for i = 1:nLayers
          Nn{i,1} = Nn{i,1}(:,rerangeIndex);
          Aa{i,1} = Aa{i,1}(:,rerangeIndex);
          [nRows nColumns] = size(Nn{i,1});
          bTemp = bias{i,1};
          bias{i,1} = repmat(bTemp,1,nColumns);
          bias{i,1} = bias{i,1}(:,rerangeIndex);
        endfor
        mInputWeight = mInputWeight(:,rerangeIndex);
        for i=2:nLayers
                  mLayerWeight{i,1}   =   mLayerWeight{i,1}
(:,rerangeIndex);
        endfor
        Im = Im(:,rerangeIndex);

        ## define how the errors are connected
        ## ATTENTION! this happens in row order...
        numTargets = net.outputs{end}.size;
        mIdentity = -eye(numTargets);
        cols = size(mIdentity,2);
                   mIdentity      =      mIdentity(:,rem(0:
(cols*nColumnsError-1),cols)+1);
        errorConnect = cell(net.numLayers,1);
        startPos = 0;
        for i=net.numLayers
          targSize = net.layers{i}.size;
```

```
                                        errorConnect{i}        =
mIdentity(startPos+[1:targSize],:);
          startPos = startPos + targSize;
        endfor

                              ##           SECOND           STEP
-----------------------------------------------
        ## define and calculate the derivative matrix dF
         ## - this is "done" by the two first derivative
functions
        ##   of the transfer functions
        ##   e.g. __dpureline, __dtansig, __dlogsig and so
on ...

        ## calculate the sensitivity matrix tildeS
        ## start at the end layer, this means of course the
output layer,
        ## the transfer function is selectable

        ## for calculating the last layer
        ## this should happen like following:
        ## tildeSx = -dFx(n_x^x)
        ## use mIdentity to calculate the number of targets
correctly
        ## for all other layers, use instead:
        ## tildeSx(-1) = dF1(n_x^(x-1))(W^x)' * tildeSx;

         for iLayers = nLayers:-1:1 # this will count from
the last
                                    # layer to the first
layer ...
          n = Nn{iLayers}; # nLayers holds the value of the
last layer...
          ## which transfer function should be used?
          if (iLayers==nLayers)
```

```
            switch(net.layers{iLayers}.transferFcn)
              case "purelin"
                tildeSxTemp = purelinderiv(n);
              case 'tansig'
                tildeSxTemp = tansigderiv(n);
              otherwise
                    error(["transfer function argument: "
net.layers{iLayers}.transferFcn  " is not valid!"])
            endswitch
            tildeSx{iLayers,1} = tildeSxTemp .* mIdentity;
            n = bias{nLayers,1};
            switch(net.layers{iLayers}.transferFcn)
              case "purelin"
                tildeSbxTemp = purelinderiv(n);
              case "tansig"
                tildeSbxTemp = tansigderiv(n);
              otherwise
                    error(["transfer function argument: "
net.layers{iLayers}.transferFcn  " is not valid!"])
            endswitch
                  tildeSbx{iLayers,1}  =  tildeSbxTemp  .*
mIdentity;
          endif

          if (iLayers<nLayers)
            dFx = ones(size(n));
                   switch(net.layers{iLayers}.transferFcn)
######## new lines ...
              case "purelin"
            dFx = purelinderiv(n);
              case "tansig"         ######## new lines ...
            dFx = tansigderiv(n);
       otherwise     ######## new lines ...
```

```
              error(["transfer    function    argument:    "
net.layers{iLayers}.transferFcn           "     is     not
valid!"])######## new lines ...
             endswitch ############ new lines ....
         LWtranspose = net.LW{iLayers+1,iLayers};
            if iLayers<(nLayers-1)
                                              mIdentity     =
-ones(net.layers{iLayers}.size,size(mIdentity,2));
            endif

            mTest = tildeSx{iLayers+1,1};
            LWtranspose = LWtranspose' * mTest;
            tildeSx{iLayers,1} = dFx .* LWtranspose;
            tildeSxTemp = dFx .* LWtranspose;
                                   tildeSbx{iLayers,1}     =
ones(size(dFx)).*tildeSxTemp;
          endif

        endfor #  if iLayers = nLayers:-1:1
                      ##        END        SECOND        STEP
-----------------------------------------------

                             ##           THIRD           STEP
-----------------------------------------------------
         ## some problems occur if we have more than only
one target... so how
        ## does the jacobi matrix looks like?

         ## each target will cause an extra row in the
jacobi matrix, for
        ## each training set..  this means, 2 targets -->
double of rows in the
        ## jacobi matrix ... 3 targets --> three times the
number of rows like
        ## with one target and so on.
```

```
        ## now calculate jacobi matrix
        ## to do this, define first the transposed of it
        ## this makes it easier to calculate on the "batch"
way, means all inputs
        ## at the same time...
          ##  and  it  makes  it  easier  to  use  the  matrix
calculation way..

                                        JjTrans               =
zeros(nRowsJacobiMatrix,nColumnsJacobiMatrix)';           #
transposed jacobi matrix


        ## Weight Gradients
        for i=1:net.numLayers
          if i==1
            newInputs = Im;
            newTemps =  tildeSx{i,1};
                                              gIW{i,1}       =
copyRows(newTemps,net.inputs{i}.size)                     .*
copyRowsInt(newInputs,net.layers{i}.size);
          endif
          if i>1
            Ad = cell2mat(Aa(i-1,1)');
            newInputs = Ad;
            newTemps = tildeSx{i,1};
                                              gLW{i,1}       =
copyRows(newTemps,net.layers{i-1}.size)                   .*
copyRowsInt(newInputs,net.layers{i}.size);
          endif
        endfor


        for i=1:net.numLayers
          [nRows, nColumns] = size(Im);
          if (i==1)
```

```
             nWeightElements = a(i)*a(i+1); # n inputs * n
hidden neurons
                  JjTrans(1:nWeightElements,:)  =    gIW{i}
(1:nWeightElements,:);
            nWeightBias = a(i+1);
            start = nWeightElements;
                    JjTrans(start+1:start+nWeightBias,:)  =
tildeSbx{i,1};
            start = start+nWeightBias;
          endif
          if (i>1)
               nLayerElements  =  a(i)*a(i+1);  #  n  hidden
neurons * n output neurons
              JjTrans(start+1:start+nLayerElements,:)=gLW{i}
(1:nLayerElements,:);
            start = start +  nLayerElements;
            nLayerBias = a(i+1);
                    JjTrans(start+1:start+nLayerBias,:)  =
tildeSbx{i,1};
            start = start + nLayerBias;
          endif
        endfor
        Jj = JjTrans';
                      ##        END        THIRD        STEP
-----------------------------------------------


      #=================================================
===
      #
      # additional functions
      #
      #=================================================
===
```

```
        function k = copyRows(k,m)
          # make copies of the ROWS of Aa matrix

          mRows = size(k,1);
          k = k(rem(0:(mRows*m-1),mRows)+1,:);
        endfunction

      #
-----------------------------------------------------

        function k = copyRowsInt(k,m)
          # make copies of the ROWS of matrix with elements
INTERLEAVED

          mRows = size(k,1);
          k = k(floor([0:(mRows*m-1)]/m)+1,:);
        endfunction

      #
============================================================
===========
      #
      # END additional functions
      #
      #
============================================================
===========

      endfunction

      ======================
      ./@network/network.m
      function  object  =  network(numInputs,  numLayers,
biasConnect, inputConnect,...
```

```
                        layerConnect, outputConnect)

    % name: Neural Network object name
    object.name = 'Neural Network';

    % efficiency: not used, included due to MATLAB
compatiblity
    object.efficiency = create_efficiency_struct();

    % userdata: a struct to allow users to add custom
information to net object
    object.userdata = struct('note','user custom data');

    % numInputs: defines the number of input vectors the
network receives
    object.numInputs = numInputs;

    % numLayers: defines the number of layers the network
has
    object.numLayers = numLayers;

    % numOutputs: read-only, must always be equal to the
number of 1's in
    %             outputConnect
    object.numOutputs = sum(outputConnect);

    % numInputDelays: not used, included due to MATLAB
compatibility
    object.numInputDelays = [];

    % numLayerDelays: not used, included due to MATLAB
compatibility
    object.numLayerDelays = [];
```

```
      % numFeedbackDelays: not used, included due to MATLAB
compatibility
      object.numFeedbackDelays = [];


      % sampleTime: not used, included due to MATLAB
compatibility
      object.sampleTime = [];


      % biasConnect: determines whether the bias is
connected to the ith layer
      object.biasConnect = biasConnect; % numLayers x 1
boolean vector


      % inputConnect: determines whether the inputs are
connected to the ith layer
      object.inputConnect = inputConnect; % numLayers x
numInputs boolean vector


      % layerConnect: determine whether the ith layer
receives a weight connection
      %               from the jth layer
      object.layerConnect = layerConnect; % numLayers x
numLayers bool matrix


      % outputConnect: determine whether the output is
connected to the ith layer
      object.outputConnect = outputConnect; % 1 x numLayers
boolean vector


      % TODO: create a method that receives the size (rows
and columns), a class name
      % and a logical index and returns a matrix of that
size, filled with objects of
      % that type only in the spots indicated by the
logical index
      %      e.g.                    m                    =
```

```
create_object_matrix(nRows,nColumns,className,logicalIndex
)

      % inputs: numInputs x 1 cell array of 'input' objects
      inputs = cell(numInputs,1);
      for i=1:numInputs
       inputs{i} = input;
      end
      object.inputs = inputs;

      % layers: numLayers x 1 cell array of 'layer' objects
      layers = cell(numLayers,1);
      for i=1:numLayers
       layers{i} = layer;
      end
      object.layers = layers;

      % outputs: numOutputs x 1 cell array of 'output'
objects
      outputs = cell(numLayers,1);
      for i=1:numLayers
       if outputConnect(i)
          outputs{i} = output;
       end
      end
      object.outputs = outputs;

      % biases: numLayers x 1 cell array of 'bias' objects
      biases = cell(numLayers,1);
      for i=1:numLayers
       if biasConnect(i)
          biases{i} = bias;
       end
```

```
    end
    object.biases = biases;

    % inputWeights: numLayers x numInputs cell array of
'weight' objects
    inputWeights = cell(numLayers, numInputs);
    for i=1:prod(size(inputWeights))
     if inputConnect(i) == 1
         inputWeights{i} = weight;
     end
    end
    object.inputWeights = inputWeights;

    % layerWeights: numLayer x numLayer cell arrya of
'weight' objects
    layerWeights = cell(numLayers, numLayers);
    for i=1:prod(size(layerWeights))
     if layerConnect(i) == 1
         layerWeights{i} = weight;
     end
    end
    object.layerWeights = layerWeights;

    % adaptFcn: not used, included due to MATLAB
compatibility
    object.adaptFcn = '';

    % adaptParam: not use, included due to MATLAB
compatibility
    object.adaptParam = 'No Neural Function Parameters';

    % derivFcn: derivate function to be used to calculate
gradients and Jacobians
    object.derivFcn = 'defaultderiv';
```

```
      % divideFcn: defines the data division function to be
used
      object.divideFcn = '';

      % divideParam: struct containing the parameters to be
used by the 'divideFcn'
      object.divideParam = '';

      % divideMode: not used, included due to MATLAB
compatibility
      object.divideMode = '';

      % initFcn: defines the network weight initialization
function
      object.initFcn = 'initLay';

      % performFcn: defines the performance function used
during training
      object.performFcn = [];

      % performParam: struct containing the parameters to
be used by the 'performFcn'
      object.performParam    =    'No    Neural    Function
Parameters';

      % plotFcns: not used, included due to MATLAB
compatibility
      object.plotFcns = cell(1,0);

      % trainFcn: defines the function to be used to train
the network
      object.trainFcn = [];

      % trainParam: struct containing the parameters to be
```

```
used by the 'trainFcn'
      object.trainParam = 'No Neural Function Parameters';

      % IW: numLayers x numInputs cell array of weight
matrices
      object.IW = cell(numLayers, numInputs);

      % LW: numLayers x numLayers cell array of weight
matrices
      object.LW = cell(numLayers, numLayers);

      % b: numLayers x 1 cell array of weight vectors
      object.b = cell(numLayers,1);

      % numWeightElements: number of weight and bias
elements in the network
      object.numWeightElements    =    sum([numel(object.IW)
numel(object.LW) numel(object.b)]);

      object = class(object, 'network');
      end

      ======================
      ./@network/__is_initialized__.m
      function isit = __is_initialized__(net)
      empty = @(x) isempty(x);
      isit = all(cellfun(empty,net.IW)) || ...
             all(cellfun(empty,net.LW)) || ...
             all(cellfun(empty,net.b));
      isit = ~isit;
      end

      ======================
```

```
./@network/create_efficiency_struct.m
function eff = create_efficiency_struct()

eff.cacheDelayedInputs = true;
eff.flattenTime = true;
eff.memoryReduction = 1;

end

======================
./@network/train.m
function [net,tr,Y,E,Pf,Af] = train(net, P, T, Pi,
Ai)
% P stands for 'patterns', the net inputs
% T stands for 'targets'
% initial checks
if isempty(net.trainFcn)
 error('@network/train:   no   training   algorithm
defined');
end

if isempty(net.performFcn)
 error('@network/train:   no   performance   function
defined');
end

% Pi and Ai are ignored
% and also are Pf and Af
Pf = [];
Af = [];

% TODO: check also if the sizes of weights matrices
are coherent
```

```
      if ~__is_initialized__(net)
       net = init(net,P,T);
      end

      trainingRecord = initial_training_record(net);
      [trainSet,         valSet,         testSet]         =
__divide_datasets__(net,P,T,trainingRecord);

      trainFcn = net.trainFcn;

      % code to make it nnet compatible
      [nRowsInputs, nColumnsInputs] = size(P);
      Im = ones(nRowsInputs,nColumnsInputs).*P;

      nLayers = net.numLayers;
      Tt = cell();
      Tt{nLayers,1} = T;

      % create a struct to contain Im and Tt, so that there
are fewer arguments
      data.Im = Im;
      data.Tt = Tt;
      % end of compatbilty code

      % train
      [net,tr]                                                =
feval(trainFcn,net,trainingRecord,trainSet,valSet,data);

      % calculate outputs and errors
      Y = sim(net,P);
      E = Y - T;

      end
```

```
function tr = initial_training_record(net)
 tr.trainFcn = net.trainFcn;
 tr.trainParam = net.trainParam;
 tr.performFcn = net.performFcn;
 tr.performParam = net.performParam;
 tr.derivFcn = net.derivFcn;
 tr.divideFcn = net.divideFcn;
 tr.divideMode = net.divideMode;
 tr.divideParam = net.divideParam;
 tr.trainInd = [];
 tr.valInd = [];
 tr.testInd = [];
 tr.stop = 'Training stop reason';
 tr.num_epochs = NaN;
 tr.trainMask = cell(0);
 tr.valMask = cell(0);
 tr.testMask = cell(0);
 tr.best_epoch = NaN;
 tr.goal = 0;
 tr.states = cell(0);
 % 'epoch' to 'val_fail' will be 1 x num_epochs+1
vectors containing training
 % information of values over each epoch (including
epoch zero)
 tr.epoch = [];
 tr.time = [];
 tr.perf = [];
 tr.vperf = [];
 tr.tperf = [];
 tr.mu = [];
 tr.gradient = [];
 tr.val_fail = [];
```

```
 tr.best_perf = Inf;
 tr.best_vperf = Inf;
 tr.best_tperf = Inf;
end

======================
./@network/purelin.m
function out = purelin(in)
 out = in;
end

======================
./@network/__checknetstruct.m
## Copyright (C) 2006 Michel D. Schmid <michaelschmid@users.sourceforge.net>
##
##
## This program is free software; you can redistribute it and/or modify it
## under the terms of the GNU General Public License as published by
## the Free Software Foundation; either version 2, or (at your option)
## any later version.
##
## This program is distributed in the hope that it will be useful, but
## WITHOUT ANY WARRANTY; without even the implied warranty of
## MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
## General Public License for more details.
##
## You should have received a copy of the GNU General Public License
```

```
      ## along with this program; see the file COPYING.  If
not, see
      ## <http://www.gnu.org/licenses/>.

      ## -*- texinfo -*-
      ## @deftypefn {Function File} {}[@var{isTrue}] =
__checknetstruct (@var{net})
      ## This function will check if a valid structure
seems to be a neural network
      ## structure
      ##
      ## @noindent
      ##
      ## left side arguments:
      ## @noindent
      ##
      ## right side arguments:
      ## @noindent
      ##
      ##
      ## @noindent
      ## are equivalent.
      ## @end deftypefn

      ## @seealso{newff,prestd,trastd}

      ## Author: Michel D. Schmid


      function isTrue = __checknetstruct(net)
      isTrue = isa(net,'network');
      endfunction
```

```
      ======================
      ./@network/tansig.m
      function out = tansig(in)
       out = 2 ./ (1 + exp(-2*in)) - 1;

       % check for values gone to infinity
       i = find(!finite(out));
       out(i) = sign(in(i));
      end

      ======================
      ./@network/sim.m
      function [output,ilfAndOutputs] = sim(net,in)
      % this structure will store induced local fields
(ilf) and outputs of each layer
      if nargout() == 2
       ilfAndOutputs = cell(2,net.numLayers);
       ilfIndex = 1;
       outputIndex = 2;
      end

      % expects input series as columns
      % expects each input sample as row vectors
      % TODO: supress + broadcast operation warning

      % find which layer is connected to the input
      layerIndex = find(net.inputConnect); % will be 1

      % get the appropriate weight and first layer transfer
functions
      weightFcn = net.inputWeights{layerIndex}.weightFcn;
      netInputFcn = net.layers{1}.netInputFcn;
      transferFcn = net.layers{1}.transferFcn;
```

```
      % get the appropriate weights
      weights = net.IW{layerIndex};
      b = net.b;

      % calculate the induced local field (v) of the first
layer (HAYKIN, YYYY)
      preInducedLocalField = feval(weightFcn, weights, in);
      inducedLocalField = feval(netInputFcn, ...
                          {[preInducedLocalField],
[b{layerIndex}]});

      % calculate the output (y) of the first layer
      output = feval(transferFcn, inducedLocalField);

      % store values for training
      if nargout() == 2 % i.e. if someone is interested in
the intermediate values
       ilfAndOutputs{ilfIndex,layerIndex}                  =
inducedLocalField;
       ilfAndOutputs{outputIndex,layerIndex} = output;
      end

      % get indexes of layers who receive synapses; also
get internal values
      [connectionReceivingIndexes connectionSendingIndexes]
= find(net.layerConnect);
      indexes         =         [connectionReceivingIndexes
connectionSendingIndexes];
      layerWeights = net.LW;

      % TODO: propagation in network with out of order
connected layers
      %       a few example of non-functional layerConnect
matrices
```

```
      %       [0 0 0; 1 0 0; 1 1 0] and [0 0 0; 0 0 1; 1 0
0]
      for layerIndex = 2:net.numLayers
       % only propagate trough the connected layers
       if ismember(layerIndex,connectionReceivingIndexes)
          % get the sender layer index
          senderIndex = find(indexes(:,1) == layerIndex);
          % get the net.layerWeights index
          weightIndex = layerIndex * senderIndex;

          % get layer functions
          weightFcn                                        =
net.layerWeights{weightIndex}.weightFcn;
          netInputFcn                                      =
net.layers{layerIndex}.netInputFcn;
          transferFcn                                      =
net.layers{layerIndex}.transferFcn;

          % combine inputs and weights (u)
          preInducedLocalField                             =
feval(weightFcn,layerWeights{weightIndex},...
                                    output);
          % add bias values (v)
          inducedLocalField      =      feval(netInputFcn,
{[preInducedLocalField], [b{layerIndex}]});
          % apply transfer function (y)
          output = feval(transferFcn, inducedLocalField);

          % store values for training
          if nargout() == 2
             ilfAndOutputs{ilfIndex,layerIndex}            =
inducedLocalField;
             ilfAndOutputs{outputIndex,layerIndex}         =
output;
          end
```

```
 end
end
end

======================
./@network/dividerand.m
function          [trainInd,valInd,testInd]          =
dividerand(Q,trainRatio,valRatio,testRatio)
% if requested, return the default parameters
if strcmp(Q,'pdefaults')
 trainInd = __default_divide_parameters__();
 valInd = [];
 testInd = [];
 return
end

indexes = randperm(Q);
[trainInd,          valInd,          testInd]          =
__divideindexes__(indexes,trainRatio,valRatio,testRatio);

end

======================
./@network/init.m
function ret = init(net, P, T)
layers = net.layers;
for i=1:net.numLayers-1
 if layers{i}.size == 0
    error('@netowrk/init.m:   all,   but   the   last,
layers must have size defined')
 end
end
inputSize = size(P,1);
```

```
      outputSize = size(T,1);

      net.inputs{1}.size = inputSize;
      layers{end}.size = outputSize;

      % IW: numLayers x numInputs cell array of weight
matrices
      % TODO: only supports one input
      idx = find(net.inputConnect);
      inputWeights = cell(size(net.IW));
      inputWeights{idx} = __rand(layers{1}.size,inputSize);
      net.IW = inputWeights;

      % LW: numLayers x numLayers cell array of weight
matrices
      [i,j] = find(net.layerConnect);
      tuples = [i j]; % (i,j) => i receives a connection
from j
      receiverLayerIdx = 1;
      senderLayerIdx = 2;
      layerWeights = cell(size(net.LW));
      for idx=1:rows(tuples)
       layerWeights{tuples(idx,1),tuples(idx,2)}          =
__rand(...
                        layers{tuples(idx,receiverLayerIdx)
}.size,...
                        layers{tuples(idx,senderLayerIdx)}.
size);
      end
      net.LW = layerWeights;

      % fill output objects's sizes
      outputs = net.outputs;
      for i = find(net.outputConnect)
```

```
 outputs{i}.size = layers{i}.size;
end

% b: numLayers x 1 cell array of weight vectors
idx = find(net.biasConnect);
biases = cell(size(net.b));
for i=idx'
 biases{i} = __rand(layers{i}.size,1);
end

net.b = biases;
net.layers = layers;
net.outputs = outputs;

ret = net;
end

function result = __rand(m,n)
 result = unifrnd(-1,1,m,n);
end

======================
./@input/input.m
function object = input()
object.name = 'Input';
object.feedbackOutput = [];
object.processFcns = [];
object.processParams = cell(1,0);
object.processSettings = cell(1,0);
object.processedRange = [];
object.processedSize = 0;
object.range = [];
```

```
      object.size = 0;
      object.userdata  =  struct('note','Put  your  custom
information here.');

      object = class(object, 'input');
      end

      ======================
      ./@input/subsref.m
      function ret = subsref(obj, sub)
      if isempty(sub)
       error('@input/subsref: missing index');
      end

      switch sub.type
      case '.'
       attribute = sub.subs;
       switch attribute
       case 'name'
          ret = obj.name;
       case 'feedbackOutput'
          ret = obj.feedbackOutput;
       case 'processFcns'
          ret = obj.processFcns;
       case 'processParams'
          ret = obj.processParams;
       case 'processSettings'
          ret = obj.processSettings;
       case 'processedRange'
          ret = obj.processedRange;
       case 'processedSize'
          ret = obj.processedSize;
```

```
       case 'range'
          ret = obj.range;
       case 'size'
          ret = obj.size;
       case 'userdata'
          ret = obj.userdata;
       otherwise
          error("@input/subsref:  invalid  property  '%s'",
attribute);
       end
      otherwise
       error("@input/subsref:   invalid   property   '%s'",
attribute);
      end

      end
      ======================
      ./@input/subsasgn.m
      function ret = subsasgn(obj, sub, rhs)
      if isempty(sub)
       error('@input/subsasgn: missing index');
      end

      ret = obj;

      switch sub.type
      case '.'
       attribute = sub.subs;
       switch attribute
       case 'name'
          ret.name = rhs;
       case 'feedbackOutput'
          ret.feedbackOutput = rhs;
```

```
        case 'processFcns'
            ret.processFcns = rhs;
        case 'processParams'
            ret.processParams = rhs;
        case 'processSettings'
            ret.processSettings = rhs;
        case 'processedRange'
            ret.processedRange = rhs;
        case 'processedSize'
            ret.processedSize = rhs;
        case 'range'
            ret.range = rhs;
        case 'size'
            ret.size = rhs;
        case 'userdata'
            ret.userdata = rhs;
        otherwise
            error("@input/subsasgn: invalid property '%s'",
attribute);
        end
      otherwise
        error("@input/subsasgn:  invalid  property  '%s'",
attribute);
      end

      end
      ======================
      ./@input/display.m

      function display(obj)
        disp(sprintf('%s object', class(obj)))
        disp(struct(obj))
      end
```