ROBERTO AMILTON BERNARDES SÓRIA

RECONHECIMENTO AUTOMÁTICO DO LOCUTOR USANDO PRÉ-PROCESSAMENTO EM SONS NASALIZADOS COM DIVERSOS CLASSIFICADORES NEURAIS

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Mestre em Engenharia.

São Paulo
2001

ROBERTO AMILTON BERNARDES SÓRIA

RECONHECIMENTO AUTOMÁTICO DO LOCUTOR USANDO PRÉ-PROCESSAMENTO EM SONS NASALIZADOS COM DIVERSOS CLASSIFICADORES NEURAIS

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Mestre em Engenharia.

Área de Concentração:
Sistemas Eletrônicos

Orientador:
Prof. Dr. Euvaldo F. Cabral Jr.

São Paulo
2001

Dedico este trabalho aos meus pais em agradecimento ao profundo incentivo e dedicação que recebi durante toda a minha vida.

À minha esposa Luciana pelo apoio e compreensão.

AGRADECIMENTOS

Em primeiro lugar agradeço a Deus pela oportunidade de ter acesso ao conhecimento.

Ao professor Dr. Euvaldo F. Cabral Jr. pelo constante incentivo e motivação, sem o qual não poderia terminar este trabalho.

Aos colegas do grupo ReNeArt – Redes Neurais Artificiais que ao longo destes anos contribuíram para a elaboração desta obra.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

DCT	- *Discrete Cosine Transform*
DFT	- *Discrete Fourier Transform*
DTW	- *Dynamic Time Warping*
FFT	- *Fast Fourier Transform*
LM	- *LearnMatrix*
LPC	- *Linear Predictive Coding*
LTW	- *Linear Time Warping*
MFC^3	- *Mel-Frequency Cepstral Coefficients Correlations*
MFCC	- *Mel-Frequency Cepstral Coefficients*
MLP	- *Multi-Layer Perceptron*
RAL	- Reconhecimento Automático do Locutor
RBF	- *Radial Basis Function*
RNA	- Rede Neural Artificial
SOFF	- *Self-Organizing Feature Finder*
UML	- *Unified Modeling Language*

RESUMO

Este trabalho avalia o reconhecimento do locutor utilizando diferentes tipos de redes neurais artificiais e um pré-processamento baseado nas correlações dos coeficientes mel-cepstrais.

Primeiramente são mostradas as bases para o aprendizado das redes neurais e em seguida a importante teoria das redes é exposta. As redes utilizadas neste trabalho são de dois tipos diferentes. O *Multi-Layer Perceptron* (MLP), a *LearnMatrix* (LM) e a *Radial Basis Function* (RBF) são redes supervisionadas, enquanto a *Self-Organizing Feature Finder* (SOFF) é não supervisionada. Estas redes são comparadas na tarefa de reconhecimento do locutor.

O pré-processamento do sinal de voz que utiliza as correlações dos coeficientes mel-cepstrais, chamados de MFC^3, é mostrado e avaliado. A viabilidade da utilização destes coeficientes é reconhecida e os resultados obtidos apontam para o MLP junto com os MFC^3 como sendo uma combinação que permite obter taxas elevadas na tarefa de reconhecimento do locutor. No entanto, os resultados mostrados para a rede binária *LearnMatrix* a definem como sendo uma ferramenta poderosa na avaliação prévia do sinal de voz.

ABSTRACT

This work evaluates different types of artificial neural networks in a speaker recognition task and a front-end based on mel-frequency cepstral coefficients correlations.

After the artificial neural networks fundamentals are presented, each neural network is explained. The networks that are used in this work are from two types. The Multi-Layer Perceptron (MLP), the LearnMatrix (LM) and the Radial Basis Function (RBF) are supervised networks while the Self-Organizing Feature Finder (SOFF) is self-organizing. These networks are compared in a speaker recognition task.

The front-end processing using the mel-frequency cepstral coefficients correlations, called MFC^3, is presented and evaluated. The use of these coefficients has been found promising and the results show that the combination of the MLP and the MFC^3 allow achieving high recognition rates. However, the results shown for the LearnMatrix network make us believe that this network may be a powerful tool for a previous speech data evaluation. The ability of dimensionality reduction of the SOFF paradigm is also discussed.

1 Introdução

1.1 Reconhecimento Automático do Locutor (RAL)

O reconhecimento do locutor é um termo genérico que se refere a qualquer atividade de discriminação de pessoas baseada nas características da voz. Duas tarefas distintas bastante difundidas podem ser extraídas desta atividade genérica. Estas duas tarefas são a verificação e a identificação automática do locutor. Elas usam análises e técnicas de decisão similares. A diferença entre a verificação e a identificação é simples. A tarefa de um sistema de verificação é de decidir se um sinal de voz pertence a um locutor de referência ou não. Portanto existem duas possibilidades: ou o sinal de voz é aceito como pertencente ao locutor de referência ou o sinal é rejeitado como sendo pertencente a um impostor. No processo da verificação, o padrão de teste é comparado ao padrão de referência e verifica-se se o sinal de voz do locutor de teste corresponde a quem ele diz ser. O sistema faz uma decisão binária (sim ou não) para verificar se o locutor é um impostor ou não. No outro lado, a tarefa do sistema de identificação do locutor é de classificar um sinal de voz como sendo pertencente a um locutor dentro de um conjunto de N locutores. Neste caso existem N possibilidades. Um sistema deste tipo é classificado com sendo *Closed Set* – Conjunto Fechado, pois não existe a possibilidade de se ter um sinal de voz não pertencente a nenhum dos locutores do conjunto. Num sistema *Open Set* – Conjunto Aberto, existe a possibilidade de que o sinal desconhecido não pertença a nenhum do locutores do conjunto. Portanto, o número de possibilidades aumenta para N+1.

É de se esperar que a taxa de reconhecimento para a verificação será melhor do que para a identificação. O tamanho da população de locutores é um ponto crítico na performance

da identificação do locutor. A taxa de erro se aproxima de 1 para populações indefinidamente grandes. No entanto, a taxa de reconhecimento da verificação do locutor fica praticamente inalterada pelo tamanho da população de locutores. A verificação do locutor precisa levar em conta um fator importante que é uma calibração estatística das características da verificação para julgar se um sinal de voz pertence ou não a um locutor, isto é, se um sinal é próximo o suficiente do padrão de referência do locutor.

O sistema de identificação do locutor em conjunto aberto é uma combinação da identificação e da verificação criando um sistema bem mais complexo.

1.2 Objetivos

A voz é um dos principais meios da comunicação humana. Para o Homem, a fala é um processo natural e simples. No entanto, a tarefa de fazer o computador entender simples palavras tem se mostrado extremamente complexa e difícil. Isto faz com que o reconhecimento de voz continue sendo um grande desafio de pesquisa. Este trabalho tem por objetivo principal apresentar um estudo sobre a utilização de redes neurais artificiais como classificadores no problema do reconhecimento automático do locutor, baseado nos coeficientes mel-cepstrais do sinal de voz, principalmente em sons nasalizados. Este trabalho analisa a eficácia de diferentes redes neurais no Reconhecimento Automático do Locutor.

Quatro paradigmas neurais são utilizados, sendo eles o *Multi-Layer Perceptron, LearnMatrix, Self-Organizing Feature Finder* e *Radial Basis Function*. O sinal de voz é tratado e os *Mel-Frequency Cepstral Coefficients* são extraídos, dando origem às suas correlações chamadas de MFC^3.

Um dos objetivos deste trabalho também é criar um modelo neural orientado a objeto de tal forma que o estudo das redes neurais artificiais possa ser facilitado através de um *software* escrito em JAVA e modelado de uma forma bastante simples utilizando UML (*Unified Modeling Language*) [29], [30], [16].

1.3 Justificativa

As redes neurais artificiais têm sido amplamente utilizadas nas mais variadas aplicações como por exemplo: processamento digital de sinais, na área de controle adaptativo, como classificadores em diversas aplicações e aproximadores de funções.

A utilização das redes neurais é justificada através de suas estruturas simples, de uma base teórica bem estabelecida e rapidez de aprendizado da rede. Apesar das redes neurais terem sido bastante utlizadas notadamente no Reconhecimento Automático do Locutor, as redes do tipo *LearnMatrix* e SOFF não têm sido muito utilizadas e têm sido muito pouco divulgadas.

Este trabalho encontra sua justificativa na necessidade de explorar diferentes paradigmas neurais e diferentes pré-processamentos do sinal de voz quando aplicados ao RAL.

1.4 Organização do trabalho

Os demais capítulos desta dissertação estão organizados da seguinte forma:

- **Capítulo 2:** apresenta um histórico resumido das Redes Neurais Artificiais. Os paradigmas neurais utilizados neste trabalho (MLP, RBF, SOFF e *LearnMatrix*) são apresentados.

- **Capítulo 3:** descreve o pré-processamento e as características utilizadas do sinal de voz.
- **Capítulo 4:** este capítulo descreve a base de dados utilizada e mostra os experimentos realizados e os resultados obtidos.
- **Capítulo 5:** os resultados são descritos e discutidos.
- **Capítulo 6:** as conclusões e comentários finais são apresentados e algumas sugestões para futuros trabalhos são propostas.
- **Anexo:** as publicações que surgiram a partir deste trabalho são listadas.
- **Apêndice:** este apêndice apresenta o modelo neural orientado a objeto, utilizando UML e o código fonte escrito em JAVA.

2 Redes Neurais Artificiais (RNA)

2.1 Descrição histórica das redes neurais artificiais

O ponto inicial na história das redes neurais artificiais é tido como sendo o trabalho pioneiro de McCulloch e Pitts em 1943 [43]. McCulloch era um psiquiatra e especialista em anatomia cerebral por experiência própria. Ele estudou durante 20 anos as representações de eventos no sistema nervoso. Pitts era um matemático genial. Em 1942 os dois pesquisadores se juntaram e em 1943, publicaram um trabalho clássico em que eles descrevem um cálculo lógico sobre as redes neurais. Esta teoria serviu de base para os experimentos de Von Neumann que idealizou elementos derivados dos elementos neurais de McCulloch e Pitts na construção do EDVAC (*Electronic Discrete Variable Automatic Computer*) que gerou o ENIAC [1] *(Electronic Numerical Integrator and Computer*). O maior desenvolvimento teórico que veio a seguir foi a publicação do livro *The Organization of Behavior* por Hebb [24], no qual uma regra fisiológica de aprendizagem foi introduzida pela primeira vez. Hebb propôs que a conectividade no cérebro humano está constantemente mudando à medida que o organismo aprende diferentes tarefas. A famosa lei de Hebb sobre a aprendizagem que diz que a sinapse entre dois neurônios é fortalecida através da repetida ativação de um neurônio pelo outro, veio logo a seguir. O livro de Hebb foi uma fonte de inspiração para o desenvolvimento de modelos computacionais da aprendizagem e de sistemas adaptativos. Em 1954, Minsky [45] publicou sua tese de doutorado que tratava das "Redes Neurais". Também neste mesmo ano a idéia de filtros adaptativos não lineares foi introduzida por Gabor. Nesses mesmos anos as memórias associativas foram também pesquisadas. Em 1961,

Karl Steinbuch introduziu a *LearnMatrix* [70] que era uma matriz plana constituindo uma rede de interruptores ligando sensores e geradores de tensão. Outro trabalho importante no desenvolvimento das memórias associativas foi o de Kohonen, Anderson e Nakano publicado em 1972 [33] que introduziu a memória de matriz de correlações.

Após uns 15 anos da publicação do trabalho de McCulloch e Pitts, uma nova visão a respeito do problema do reconhecimento de padrões veio com Rosenblatt em 1958 [60] com a publicação do livro intitulado *Perceptron* onde ele demonstra pela primeira vez o teorema da convergência do perceptron. Em 1960, Widrow e Hoff [73] apresentaram o algoritmo dos mínimos quadrados com o qual criaram o Adaline (*Adaptive Linear Element*). A diferença entre o perceptron e o adaline reside no algoritmo de aprendizagem. Dois anos depois Widrow [72] apresentou o Madaline (Multiple-adaline) que foi umas das primeiras redes neurais com várias camadas com elementos adaptativos. Em seguida veio o livro de Minsky e Papert [46] (1969) que mostrou a limitação de um perceptron de uma camada. Por causa deste trabalho os ânimos foram desmantelados e os interesses nas redes neurais artificiais diminuíram nos anos 70. No entanto, uma atividade importante que apareceu nestes mesmos anos foram os *Self-Organizing Maps* usando aprendizagem competitiva. Em 1980, Grossberg introduziu a rede ART [21] (*Adaptive Resonance Theory*) baseada no fenômeno de amplificação e prolongamento da atividade neural chamado de ressonância adaptativa. Em 1982, Hopfield [27] usou a idéia de uma função de energia para formular uma rede recorrente com conexões sinápticas simétricas que se tornaram as redes de Hopfield. Uma outra publicação de peso em 1982 foi o trabalho sobre *self-organizing maps* de Kohonen [66]. Em 1986, o desenvolvimento do algoritmo de aprendizagem chamado back-propagation foi reportado por Rumelhart, Hinton e Williams [61]. No mesmo ano Rumelhart e McClelland [62] publicaram o livro sobre processamentos paralelos distribuídos PDP (*Parallel Disstributed Processing*) que

influenciou grandemente o uso da aprendizagem por retro-propagação para treinar os MLP (*Multi-Layer Perceptrons*). Broomhead e Lowe [6] descreveram em 1988 um procedimento para projetar redes de várias camadas usando *radial basis functions* (RBF) que mostraram ser uma alternativa aos MLPs.

Talvez, os trabalhos de Hopfield [27] em 1982 e Rumelhart e McLelland [62] em 1986 tenham sido aqueles que mais influenciaram a volta do interesse por redes neurais nos anos 80. As redes neurais percorreram um longo caminho desde McCulloch e Pitts e estabeleceram raízes profundas nas ciências neurais, psicologia, matemática, ciências físicas e engenharia.

2.2 Modelos Lineares

Um modelo linear para uma função y(x) toma a forma

$$f(x) = \sum_{j=1}^{m} w_j h_j(x) \tag{2.1}$$

O modelo *f* é expresso como uma combinação linear de um conjunto de *m* funções fixas (freqüentemente chamadas de funções base, em analogia ao conceito de um vetor sendo composto de uma combinação linear de vetores base). A escolha dos parâmetros *w* para o coeficiente das combinações lineares e a função *h* para as funções base reflete nosso interesse em redes neurais que têm pesos e unidades ocultas.

A flexibilidade de *f* e sua capacidade de se ajustar a várias funções diferentes deriva da liberdade de escolher diferentes valores para os pesos. As funções de base assim como os parâmetros que elas podem conter são fixos. No caso em que as funções de base mudam durante o processo de aprendizagem, o modelo é não-linear.

Qualquer conjunto de funções pode ser usado como conjunto de base embora seja desejável que as funções sejam bem comportadas (deriváveis). Entretanto, modelos contendo só funções de base derivados de uma classe particular tem um interesse especial. A estatística clássica está repleta de modelos lineares cujas funções de base são polinômios. As combinações de ondas sinoidais (séries Fourier),

$$h_j(x) = sin\left(\frac{2\pi j(x-\theta_j)}{m}\right) \tag{2.2}$$

são freqüentemente usadas em aplicações de processamento de sinal. Funções logísticas, do tipo

$$h(x) = \frac{1}{1 + e^{(b^T x - b_0)}} \tag{2.3}$$

são populares em redes neuras artificiais, particularmente em perceptrons de multi-camadas (MLPs).

Um exemplo familiar de polinômio muito simples é a linha reta

$$f(x) = ax + b \tag{2.4}$$

que é um modelo linear cujas duas funções de base são

$$h_1(x) = 1 \tag{2.5}$$

$$h_2(x) = x \tag{2.6}$$

e cujos pesos são $w1 = a$ e $w2 = b$. Este é um modelo muito simples e não é flexível o suficiente para ser usado em aprendizagem supervisionada.

Os modelos lineares são mais simples a serem analisados matematicamente. Em particular, se os problemas de aprendizagem supervisionada são resolvidos pelos mínimos quadrados, então é possível derivar e resolver um conjunto de equações para encontrar o conjunto de pesos a partir do conjunto de treinamento. O mesmo não se aplica a modelos não-lineares, tal como os MLPs, que requerem um processo numérico iterativo para a otimização.

2.3 Aprendizagem supervisionada

Um problema em estatística com aplicações em muitas áreas é a estimativa de uma função a partir de alguns pares de entrada/saída com nenhum ou pouco conhecimento da função. Este problema é tão comum que tem nomes diferentes nas várias disciplinas (por exemplo regressão não-paramétrica, aproximação de função, identificação de sistema).

Na disciplina de redes neurais artificiais, este problema é chamado de apredizagem supervisionada. A função é aprendida a partir dos exemplos fornecidos. O conjunto de exemplos, ou de treinamento contém elementos que consistem em valores casados de variáveis independentes (entrada) e variáveis dependentes (saída). Por exemplo, a variável independente na relação funcional

$$y = f(x) \tag{2.7}$$

é x (um vetor) e a variável dependente é y (um escalar). O valor de y depende, através da função f, de cada um dos componentes do vetor variável

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{2.8}$$

O conjunto de treino, composto de p pares (indexados a partir de 1 até p), é representado por

$$T = \{(x_i, \hat{y}_i)\}i^{p} = 1 \tag{2.9}$$

A razão para o acento circunflexo sobre a letra *y* (convenção que indica uma estimativa ou valor incerto) é que os valores de saída do conjunto de treino normalmente são corrompidos por ruído. Em outras palavras, o valor correto para o par de x_i, a saber y_i, é desconhecido. O conjunto de treinamento só especifica qual $\hat{y}_i$ é igual a y_i acrescido de um certo valor desconhecido de ruído.

2.4 Multi-Layer Perceptron (MLP)

2.4.1 Arquitetura do MLP

O MLP é uma rede do tipo *feed-forward* e consiste em várias camadas de unidades de processamento conforme mostrado na figura abaixo. Existe um camada de entrada *F* por onde são apresentados os padrões a serem classificados, uma camada interna *H* e uma camada de saída *L* que indica o resultado da classificação. Os pesos *W* conectam camadas adjacentes, mas não existe conexões dentro da mesma camada.

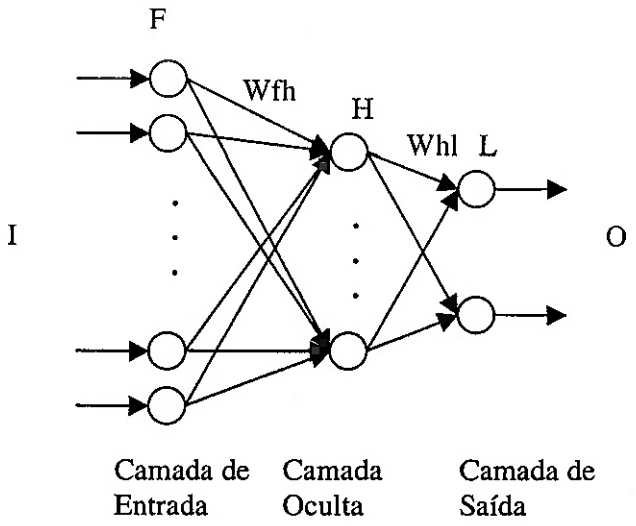


FIGURA 2.4.1 - MLP COM UMA ÚNICA CAMADA INTERNA

A função de transferência de cada unidade de processamento (neurônio) nas camadas interna e de saída pode ser dada por:

$$f(v) = \frac{1}{1+e^{-v}} \tag{2.10}$$

Onde v corresponde a uma combinação ponderada pelos pesos da saída de todos os neurônios da camada anterior. Assim sendo, essa combinação ponderada para a camada interna é dada por:

$$v_j = \sum_i w_{ij} x_i \tag{2.11}$$

Para a camada de saída, v é dado por:

$$v_k = \sum_j w_{jk} x_j \tag{2.12}$$

O treinamento do MLP é realizado de tal forma que seja necessário ter de antemão pares de entradas e saídas desejadas. Começando com um conjunto de pesos gerados de forma aleatória, as saídas são calculadas para um determinado padrão de entrada. Cada saída é comparada com a saída desejada e o erro resultante é usado para ajustar os pesos de tal forma a reduzir o erro total. Este procedimento é repetido várias vezes para cada par de entrada/saída pertencente ao conjunto de treinamento até que o erro seja minimizado.

2.4.2 Notação mnemônica

A notação mnemônica especial para o MLP [CABRAL 98] foi utilizada . Sua descrição está a seguir.

As camadas são representadas por uma letra maiúscula. A tabela a seguir mostra esta notação.

Notação	Descrição
F (*first*)	Primeira camada, (NF) neurônios.
H(1)	Primeira camada oculta, (NH1) neurônios.
H(2)	Segunda camada oculta, (NH2) neurônios.
H(J)	J-ésima camada oculta, (NHJ) neurônios.
H(HD)	última camada oculta, (NHD) neurônios.
L (*Last*)	Última camada, (NL) neurônios.

Os vetores de entrada são definidos, um para cada camada, inclusive para a camada de entrada, cujo formato genérico é:

$$\vec{I}_p^{(camada)} = \left(i_{1,p}^{(camada)}, i_{2,p}^{(camada)}, \ldots, i_{\substack{\text{n. neurônios} \\ \text{da camada},p}}^{(camada)} \right)$$

Notação	Descrição
$\vec{I}_p^{(f)} = \left(i_{1,p}^{(f)}, i_{2,p}^{(f)}, \ldots, i_{NF,p}^{(f)} \right)$	p-ésimo vetor de entrada da camada de entrada
$\vec{I}_p^{(hJ)} = \left(i_{1,p}^{(hJ)}, i_{2,p}^{(hJ)}, \ldots, i_{NHJ,p}^{(hJ)} \right)$	p-ésimo vetor de entrada da J-ésima camada oculta.
$\vec{I}_p^{(l)} = \left(i_{1,p}^{(l)}, i_{2,p}^{(l)}, \ldots, i_{NL,p}^{(l)} \right)$	p-ésimo vetor de entrada da última camada.

Os vetores de saída são definidos, um para cada camada:

$$\vec{O}_p^{(camada)} = \left(o_{1,p}^{(camada)}, o_{2,p}^{(camada)}, \ldots, o_{\substack{\text{n. neurônios} \\ \text{da camada},p}}^{(camada)} \right)$$

Notação	Descrição
$\vec{O}_p^{(f)} = \left(o_{1,p}^{(f)}, o_{2,p}^{(f)}, \ldots, o_{NF,p}^{(f)} \right)$	p-ésimo vetor de saída da primeira camada.
$\vec{O}_p^{(hJ)} = \left(o_{1,p}^{(hJ)}, o_{2,p}^{(hJ)}, \ldots, o_{NHJ,p}^{(hJ)} \right)$	p-ésimo vetor de saída da J-ésima camada oculta.
$\vec{O}_p^{(l)} = \left(o_{1,p}^{(l)}, o_{2,p}^{(l)}, \ldots, o_{NL,p}^{(l)} \right)$	p-ésimo vetor de saída da camada de saída.

O *Target* $\vec{T}_p$ é a saída desejada da rede para o seu padrão de entrada correspondente.

$$\vec{T}_p = \left(t_{1,p}, t_{2,p}, \ldots, t_{NL,p} \right)$$

A dimensão do vetor de *target* deve ser a mesma do vetor de saída da rede, para que se possa calcular o erro.

O vetor de *bias* é definido da seguinte maneira:

$$\vec{B}^{(camada)} = \left(b_1^{(camada)}, b_2^{(camada)}, \ldots, b_{\substack{\text{n. neurônios} \\ \text{da camada}}}^{(camada)} \right)$$

Notação	Descrição
$\vec{B}^{(f)} = \left(b_1^{(f)}, b_2^{(f)}, \ldots, b_{NF}^{(f)}\right)$	p-ésimo vetor de *bias* da primeira camada.
$\vec{B}^{(hJ)} = \left(b_1^{(hJ)}, b_2^{(hJ)}, \ldots, b_{NHJ}^{(hJ)}\right)$	p-ésimo vetor de *bias* da j-ésima camada oculta.
$\vec{B}^{(l)} = \left(b_1^{(l)}, b_2^{(l)}, \ldots, b_{NL}^{(l)}\right)$	p-ésimo vetor de *bias* da última camada.

As matrizes de pesos correspondem às diversas conexões entre as camadas. Seu formato genérico é:

$$w_{\substack{\text{neuronio} \\ \text{origem}}, \substack{\text{neuronio} \\ \text{destino}}, p}^{\begin{pmatrix} camada\ \text{camada} \\ origem\ ' \text{destino} \end{pmatrix}}$$

Notação	Descrição
$[W_p^{(f,h1)}]=\begin{bmatrix} W_{1,1,p}^{(f,h)} & W_{1,2,p}^{(f,h)} & \cdots & W_{1,NH1,p}^{(f,h)} \\ W_{2,1,p}^{(f,h)} & W_{2,2,p}^{(f,h)} & \cdots & W_{2,NH1,p}^{(f,h)} \\ \vdots & \vdots & \ddots & \vdots \\ W_{NF,1,p}^{(f,h)} & W_{NF,2,p}^{(f,h)} & \cdots & W_{NF,NH1,p}^{(f,h)} \end{bmatrix}$	Matriz de pesos

A função neural é definida como NF_{camada}

Notação	Descrição
NF_{H1}	função neural associada à primeira camada oculta.

2.4.3 O algoritmo de treinamento *Back-Propagation*

O algoritmo de treinamento chamado de *Back-Propagation* é um algoritmo baseado no gradiente descendente projetado para minimizar o erro quadrático entre a saída atual da rede e a saída desejada. Este algoritmo de aprendizado é constituído de um passo *forward* e de um *backward*.

Inicialmente é necessário inicializar os pesos com números pseudo-randômicos de pequeno valor. Em seguida, apresenta-se à rede o vetor de entrada p e especifica-se o vetor de saída desejado, chamado de *Target T*. Calcula-se então as saídas O na etapa *feed-forward* usando a função de transferência *sigmoidal*.

A etapa *forward* consiste em calcular a saída da rede para um determinado padrão de entrada, *p*. De modo geral a entrada da rede é igual à saída da primeira camada,

portanto, $\vec{I}_p^{(f)} = \vec{O}_p^{(f)}$. A entrada da primeira camada oculta é descrita pela equação abaixo:

$$i_{h,p}^{(h1)} = \sum_{f=1}^{NF} \left(w_{f,h,p}^{(f,h1)} \cdot o_{f,p}^{(f)} \right) + b_h^{(h1)} \tag{2.13}$$

onde $h \in [1..NH1]$.

Muitas vezes, o *bias*, $b_h^{(h1)}$ é substituído pela equação :

$$b_h^{(h1)} = w_{0,h,p}^{(f,h1)} \cdot o_{0,p}^{(f)} \tag{2.14}$$

onde $o_{0,p}^{(f)} = 1$. Portanto, temos:

$$i_{h,p}^{(h1)} = \sum_{f=0}^{NF} \left(w_{f,h,p}^{(f,h1)} \cdot o_{f,p}^{(f)} \right) \tag{2.15}$$

Finalmente, temos a saída da primeira camada oculta:

$$o_{h,p}^{(h1)} = \begin{cases} NF_{H1}\left(i_{h,p}^{(h1)}\right) & , \text{para } h \in [1..NH1] \\ 1 & , \text{para } h = 0 \end{cases} \tag{2.16}$$

Segundo a convenção mnemônica para o MLP, a camada oculta é representada genericamente por HJ. Com isso, a entrada da camada é dada por:

$$i_{h,p}^{(hJ)} = \sum_{k=0}^{NH(J-1)} \left(w_{k,h,p}^{(hJ-1,hJ)} \cdot o_{k,p}^{(hJ-1)} \right) \tag{2.17}$$

para $h \in [1..NHJ]$, e a saída é dada por:

$$o_{h,p}^{(hJ)} = \begin{cases} NF_{HJ}\left(i_{h,p}^{(hJ)}\right) & , \text{para } h \in [1..NHJ] \\ 1 & , \text{para } h = 0 \end{cases} \tag{2.18}$$

A entrada da última camada oculta (HD) é dada pela equação abaixo.

$$i_{h,p}^{(hD)} = \sum_{k=0}^{NH(D-1)} \left(w_{k,h,p}^{(hD-1,hD)} \cdot o_{k,p}^{(hD-1)} \right) \tag{2.19}$$

para $h \in [1..NHD]$, e a saída é dada por:

$$o_{h,p}^{(hD)} = \begin{cases} NF_{HD}\left(i_{h,p}^{(hD)}\right) & \text{, para } h \in [1..NHD] \\ 1 & \text{, para } h = 0 \end{cases} \tag{2.20}$$

A entrada da última camada é escrita pela equação abaixo:

$$i_{l,p}^{(l)} = \sum_{h=0}^{NHD} \left(w_{h,l,p}^{(hD,l)} \cdot o_{h,p}^{(hD)} \right) \tag{2.21}$$

para $l \in [1..NL]$, e a saída da rede para o padrão $\vec{I}_p^{(f)}$ é dada por:

$$o_{l,p}^{(l)} = NF_L\left(i_{l,p}^{(l)}\right) \quad \text{, para } l \in [1..NL] \tag{2.22}$$

Na etapa *backward*, define-se um erro $\vec{E}_p$ correspondente a uma comparação entre a saída da rede, $\vec{O}_p^{(l)}$, e o *target*, $\vec{T}_p$.

$$\vec{E}_p = \left(e_{1,p}, e_{2,p}, \ldots, e_{NL,p}\right) = f_E\left(\vec{T}_p, \vec{O}_p\right) \tag{2.23}$$

Usualmente f_E é definida como sendo:

$$e_{l,p} = \left(t_{l,p} - o_{l,p}^{(l)}\right) \tag{2.24}$$

para $l \in [1..NL]$.

O método *Generalised Delta Rule* (*GDR*) proposto por Rumelhart [61] para minimizar o erro, é uma generalização do método *Delta Rule* (*Widrow-Hoff rule*) proposto em [72], e utiliza a seguinte função de erro:

$$f_E^{(RHW)}\left(\vec{E}_p\right) = k\sum_{l=1}^{NL} e_{l,p}^2 \tag{2.25}$$

onde o índice *RHW* [61] se deve ao fato de ter sido proposto por Rumelhart, Hinton e Willians; k é uma constante.

O método *Generalised Delta Rule* utiliza o método do gradiente descendente para determinar em qual direção devem ser feitas as alterações nos pesos de modo a minimizar a função de erro $f_E^{(RHW)}\left(\vec{E}_p\right)$.

A seguir são apresentadas as equações de ajustes dos pesos que conectam estas camadas no caso de uma rede de 3 camadas. A primeira equação é utilizada no ajuste dos pesos entre a primeira camada e a camada oculta. A equação seguinte define o ajuste dos pesos entre a camada oculta e a última camada, e a última equação define o ajuste dos pesos entre a primeira camada e a camada oculta.

$$\delta_{l,p}^{(h,l)} = -\left(t_l - o_{l,p}^{(l)}\right)\cdot N_L'\left(i_{l,p}^{(l)}\right) \tag{2.26}$$

$$\Delta w_{h,l,p}^{(h,l)} = \eta\cdot\left(t_l - o_{l,p}^{(l)}\right)\cdot N_L'\left(i_{l,p}^{(l)}\right)\cdot o_{h,p}^{(h)} \tag{2.27}$$

$$\Delta w_{f,h,p}^{(f,h)} = -\eta \cdot o_{f,p}^{(f)} \cdot N_H'\left(i_{h,p}^{(h)}\right) \cdot \sum_{l=1}^{NL}\left(\delta_{l,p}^{(h,l)} \cdot w_{h,l,p}^{(h,l)}\right) \tag{2.28}$$

2.4.4 Aplicações do MLP

As redes neurais do tipo *Multi-Layer Perceptrons*, treinadas com o algoritmo *back-propagation* estão sendo aplicadas com sucesso em várias áreas diferentes. Essas áreas incluem, por exemplo:

- Reconhecimento de voz [47].
- OCR – *Optical Character Recognition* [64].
- Controle [32].
- Detecção e classificação de objetos em radares [23].
- Detecção e classificação em sonares [35].
- Modelamento do controle dos movimentos dos olhos [58].

2.5 Radial Basis Function (RBF)

2.5.1 Introdução

A rede do tipo *Radial Basis Function* ou Função de Base Radial aborda o problema de aproximação de curvas em um espaço multi-dimensional. Baseado nisto, a aprendizagem da rede pode ser vista como sendo o problema de encontrar uma superfície multi-dimensional que se molde aos padrões de treinamento. A estrutura da rede apresenta uma camada oculta que fornece um conjunto de funções que definem centros para os padrões de entrada quando estes são convertidos ao espaço da camada oculta. Estas funções são chamadas de funções de base radial. As funções de base radial foram primeiramente utilizadas em problemas de interpolação. Os primeiros a utilizarem as funções de base radial no contexto de redes neurais foram Broomhead e Lowe em 1988 [6]. Uma rede do tipo RBF é composta de três camadas de nós completamente diferentes. A primeira é simplesmente composta de nós que repassam o vetor de entrada. A segunda é uma camada oculta de uma dimensão suficientemente alta. A terceira camada fornece a resposta da rede. A RBF pode ser utilizada para resolver um problema complexo de classificação de padrões, através da transformação dos padrões de uma maneira não linear em um espaço de alta dimensão. A justificativa para esta transformação é o teorema de Cover para a separação de padrões [13]. Este teorema mostra que num problema de classificação de padrões, um espaço de alta dimensão é mais favorável para separar linearmente do que um espaço de duas dimensões.

2.5.2 Arquitetura da RBF

A rede do tipo *Radial Basis Function*, pode ser vista como um modelo de rede *feed-forward* com uma única camada oculta de nós que calculam a distância entre o padrão de entrada e localizações fixas no espaço de padrões. Definindo a operação ||...|| como sendo a distância medida em RN, temos o cálculo da primeira camada de nós, da seguinte forma:

$$O_{li} = R_i(\alpha_{li}) = R_i(\|I - C_i\|) \tag{2.29}$$

Onde I representa o padrão de entrada e O_{li} a saída do i-ésimo nó da primeira camada, associado à uma localização de referência *Ci* e função de ativação $R_i(\alpha_{li})$. A distância Euclidiana é usada para calcular ||I-Ci||, da seguinte forma:

$$\|I - C_i\|^2 = (I - C_i) \cdot M \cdot (I - C_i)^T \tag{2.30}$$

onde M é a matriz unidade, mas pode ser também ponderada por exemplo como a matriz inversa de co-variância do conjunto de padrões. Os nós de saída formam uma soma ponderada das saídas dos nós da primeira camada e um limiar dado por Tj. A função geral da rede é dada por:

$$O_j = F(\alpha_{2j}) = F\left(\sum_i W_{ji} R_i(\alpha_{li}) - T_j\right) \tag{2.31}$$

Onde O_j é a j-ésima saída da rede e $F(\alpha_{2j})$ é a função de ativação *sigmoidal* dada por:

$$F(\alpha_{2j}) = \frac{1}{1+\exp(-\alpha_{2j})} \tag{2.32}$$

A rede do tipo função de base radial (RBF) pode ser interpretada de formas variadas dependendo da escolha da função de ativação da camada oculta, centros de referência e algoritmo de treinamento. Ao escolher $R_i(\alpha_{li})$ como a função exponencial:

$$R_i(\alpha_{li}) = \exp(-\alpha^2{}_{li}) \tag{2.33}$$

resulta numa rede que implementa a soma de aproximações *Gaussianas* de distribuições. Ao especificar os pesos W da camada de saída como sendo as classes de probabilidade a priori, a rede probabilística de Specht pode ser obtida [69].

2.5.3 Funções de Base Radial

As funções de base radial são uma classe especial de funções. Sua característica principal é que sua resposta diminui (ou aumenta) monotônicamente com relação à distância a um ponto central. O centro, a distância escalar, e a forma da função radial são parâmetros do modelo.

Uma função de base radial típica é a *Gaussiana* que, no caso de uma entrada escalar *x*, é dada por:

$$h(x) = \exp\left(-\frac{(x-c)^2}{r^2}\right) \tag{2.34}$$

Seus parâmetros são seu centro c e seu raio r. A figura 2.5.1, a seguir, ilustra uma RBF *Gaussiana* com centro $c = 0$ e raio $r = 1$.

A RBF *Gaussiana* diminui monotônicamente com a distância ao centro. Em contraste, uma RBF multiquadrada que, no caso de uma entrada escalar x, é

$$h(x) = \frac{\sqrt{r^2 + (x-c)^2}}{r} \tag{2.35}$$

aumenta monotônicamente com a distância ao centro. As RBFs *Gaussianas* são localizadas (elas dão uma resposta significativa somente numa vizinhaça próxima ao centro) e são normalmente mais usadas do que as RBFs do tipo multiquadradas que têm uma resposta global. As RBFs *Gaussianas* também são mais biologicamente plausíveis porque sua resposta é finita.

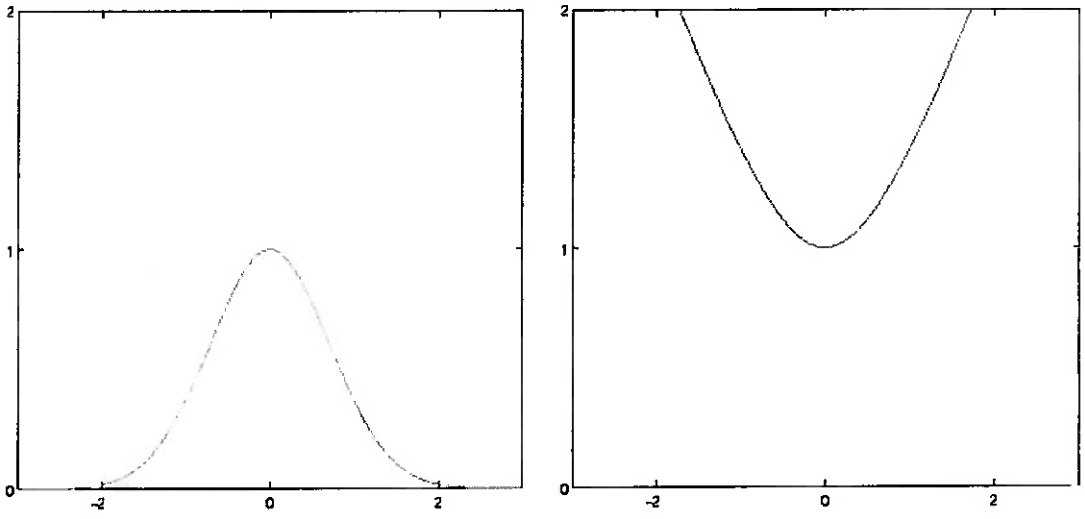


FIGURA 2.5.1 - RBF *GAUSSIANA* (ESQUERDA) E MULTIQUADRADA (DIREITA).

A fórmula mais genérica para qualquer função de base radial é:

$$h(x) = \phi\left((x-c)^T R^{-1}(x-c)\right) \tag{2.36}$$

onde ϕ é a função usada (*Gaussiana*, multiquadrada, etc...), c é o centro e $\boldsymbol{R}$ é a métrica.

O termo

$$\left((x-c)^T R^{-1}(x-c)\right) \tag{2.37}$$

é a distância entre a entrada x e o centro C usando a métrica defina por $\boldsymbol{R}$. Há muitos tipo de funções utilizadas. A tabela abaixo apresenta alguns deles.

Tabela 2.5.1 - Funções de base radial mais comuns

Gaussiana	Multiquadrada	Inversa da Multiquadrada	Cauchy
$\phi(z) = e^{-z}$	$\phi(z) = (1+z)^{\frac{1}{2}}$	$\phi(z) = (1+z)^{-\frac{1}{2}}$	$\phi(z) = (1+z)^{-1}$

Muitas vezes a métrica é a Euclidiana. Neste caso, $\mathbf{R} = r^2\mathbf{I}$ para algum raio escalar r e a equação acima fica simplificada da seguinte forma:

$$h(x) = \phi\left(\frac{(x-c)^T(x-c)}{r^2}\right) \tag{2.38}$$

Uma simplificação ainda maior é no caso de um espaço de uma dimensão, em cujo caso temos:

$$h(x) = \phi\left(\frac{(x-c)^2}{r^2}\right) \tag{2.39}$$

A figura abaixo ilustra esta função, para $c = 0$ e $r = 1$, para as RBFs citadas acima.

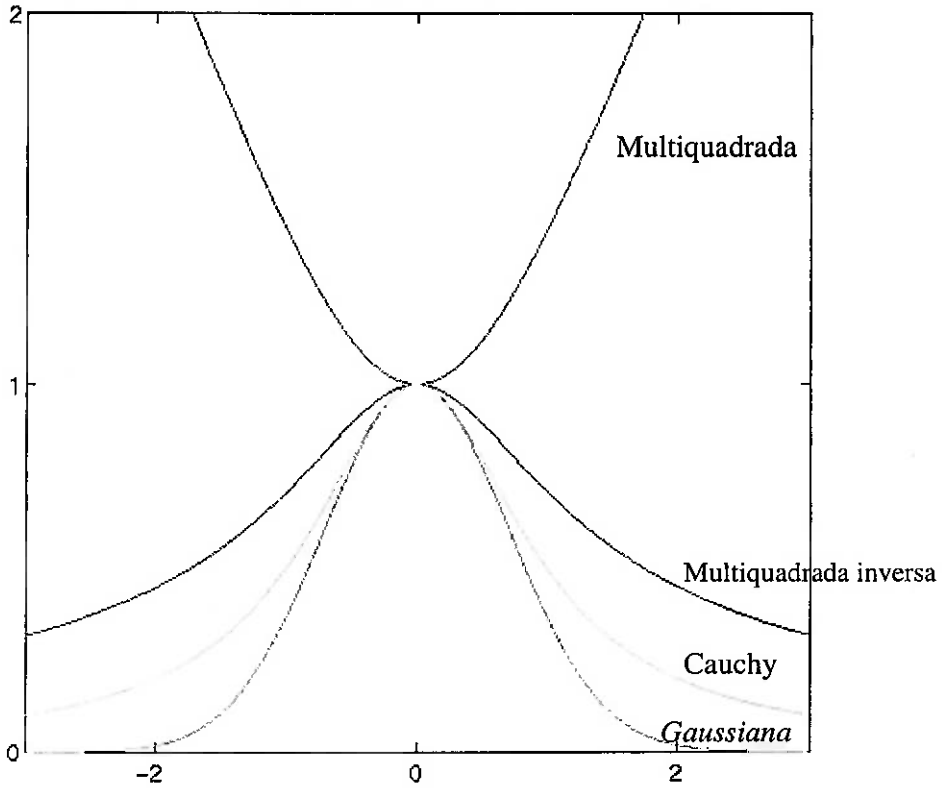


FIGURA 2.5.2 - RBFS COM RAIO UNITÁRIO E CENTRO NA ORIGEM

2.5.4 Redes Neurais do tipo Radial Basis Function

As funções de base radial são simplesmente uma classe de funções. Em princípio, elas poderiam ser empregadas em qualquer tipo de modelo (linear ou não-linear) e qualquer tipo de rede (uma camada ou multi-camada). Entretanto, desde que Broomhead e Lowe [6] publicaram seu trabalho, as RBFs tradicionalmente foram associadas com funções de base radial numa única camada tal como mostrado na figura embaixo.

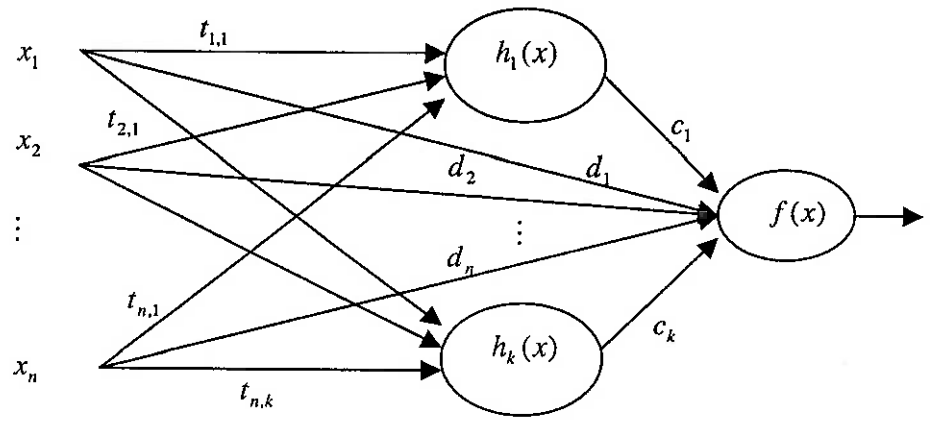


FIGURA 2.5.3 - A REDE NEURAL DO TIPO FUNÇÃO DE BASE RADIAL TRADICIONAL.

Cada um dos n vetores de entrada interagem da entrada para a saída (*feed-forward*) com k funções de base radial cujas saídas são linearmente combinadas com pesos para formar a saída da rede $f(x)$.

Uma rede RBF pode ser não-linear se as funções de base mudarem de posição ou de tamanho, ou se houver mais de uma camada oculta. Neste trabalho, focalizaremos em redes de uma única camada oculta com funções que estão fixas em posição e tamanho.

2.5.5 Fundamentos das redes do tipo RBF

O princípio das funções de base radial deriva da teoria de aproximação de funções. Dados **N** pares $(\vec{x}_i, y_i)$ $(\vec{x} \in \Re^n, y \in \Re)$ procuramos por uma função f da forma:

$$f(\vec{x}) = \sum_{i=1}^{K} c_i h\left(\left|\vec{x} - \vec{t}_i\right|\right) \tag{2.40}$$

Onde h é a função de base radial e $\vec{t}_i$ são os **K** centros que têm que ser selecionados.

Os coeficientes c_i são desconhecidos no momento e têm que ser calculados. $\vec{x}_i$ e $\vec{t}_i$ são elementos de um espaço vetorial n-dimensional.

A função h é aplicada à distância Euclidiana entre cada centro $\vec{t}_i$ e o argumento dado $\vec{x}$. Freqüentemente a função h é a função *Gaussiana* que tem seu máximo numa distância zero de seu centro. Neste caso, valores de $\vec{x}$ que sejam iguais a um centro $\vec{t}$ produzem um valor de saída igual a 1, enquanto a saída torna-se praticamente zero para distâncias maiores.

A função f deve ser uma aproximação dos N pares $(\vec{x}_i, y_i)$ dados e portanto deve reduzir a seguinte função de erro H:

$$H[f] = \sum_{i=1}^{N} \left(y_i - f(\vec{x}_i)\right)^2 + \lambda \|Pf\|^2 \tag{2.41}$$

A primeira parte da definição de **H** (a somatória) é a condição que minimiza o erro total da aproximação, i.e. que faz f aproximar os **N** pontos dados. A segunda parte de H

($\|\boldsymbol{Pf}\|2$) é um estabilizador que força $\boldsymbol{f}$ a tornar-se tão suave quanto possível. O fator λ determina a influência deste estabilizador.

Sob certas condições é possível mostrar que um conjunto de coeficientes c_i pode ser calculado de modo a minimizar **H**. Este cálculo depende dos centros $\vec{t}_i$ que têm que ser escolhidos de antemão.

Introduzindo os seguintes vetores e matrizes

$$\begin{aligned} \vec{c} &= (c_1, \ \ldots, \ c_K)^T \\ \vec{y} &= \{y_1, \ \ldots, \ y_N\}^T \end{aligned} \tag{2.42}$$

$$\begin{aligned} G &= \begin{pmatrix} h(|\vec{x}_1 - \vec{t}_1|) & \cdots & h(|\vec{x}_1 - \vec{t}_K|) \\ \vdots & \ddots & \vdots \\ h(|\vec{x}_N - \vec{t}_1|) & \cdots & h(|\vec{x}_N - \vec{t}_K|) \end{pmatrix} \\ G_\Delta &= \begin{pmatrix} h(|\vec{t}_1 - \vec{t}_1|) & \cdots & h(|\vec{t}_1 - \vec{t}_K|) \\ \vdots & \ddots & \vdots \\ h(|\vec{t}_K - \vec{t}_1|) & \cdots & h(|\vec{t}_K - \vec{t}_K|) \end{pmatrix} \end{aligned} \tag{2.43}$$

o conjunto de parâmetros desconhecidos c_i pode ser calculado da seguinte maneira:

$$\vec{c} = \left(G^T \cdot G + \lambda G_\Delta\right)^{-1} \cdot G^T \cdot \vec{y} \tag{2.44}$$

Igualando λ a 0, esta fórmula torna-se um sistema linear que segue diretamente das condições de uma interpolação exata do problema dado:

$$f(\vec{x}_j) = \sum_{i=1}^{K} c_i h\left(\left|\vec{x}_j - \vec{t}_i\right|\right)^{t} = y_j \qquad (2.45)$$
$$j = 1, \cdots, N$$

O método das funções de base radial pode ser representado facilmente por uma rede neural de três camadas do tipo *feed-forward*. A camada de entrada consiste em n unidades que representam os elementos do vetor de entrada $\vec{x}$. Os **K** componentes da soma na definição de f são representados pelas unidades na camada oculta. Os pesos entre a entrada e a camada oculta contêm os elementos dos vetores $\vec{t}_i$. As unidades ocultas calculam a distância Euclidiana entre o padrão de entrada e o vetor que é representado pelos pesos levando à unidade oculta. A ativação das unidades ocultas é calculada aplicando-se a distância Euclidiana à função h. A figura abaixo mostra a arquitetura das unidades ocultas.

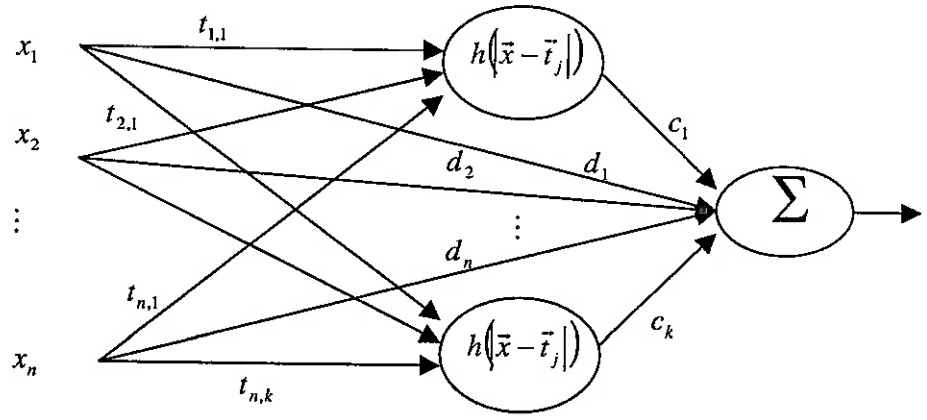


Figura 2.5.4 - Unidades contendo a função de base radial

Neste caso, o único neurônio de saída recebe sua entrada de todos os neurônios escondidos. As conexões levando ao neurônio de saída contêm os coeficientes c_i. A ativação do neurônio de saída é determinada pela soma ponderada de suas entradas.

A arquitetura previamente descrita de uma rede neural que realiza aproximações a partir de funções de base radial, pode ser expandida com algumas características úteis: mais de um neurônio de saída é possível, o que permite a aproximação de várias funções f ao redor do mesmo conjunto de centros $\vec{t}_i$. A ativação das unidades de saída pode ser calculada usando uma função não-linear σ (por exemplo a função *sigmoid*al). Um bias dos neurônios de saída e uma conexão direta entre a entrada e a camada de saída podem ser usados para melhorar a qualidade da aproximação. O bias das unidades ocultas pode ser usado para modificar a característica da função h. Em soma uma rede neural é capaz de representar o seguinte conjunto de aproximações:

$$o_k(\vec{x}) = \sigma\left(\sum_{j=1}^{K} c_{j,k} h\left(|\vec{x} - \vec{t}_j|, p_j\right) + \sum_{i=1}^{n} d_{i,k} x_i + b_k \right) = \sigma(f_k(\vec{x})) \qquad (2.46)$$
$$k = 1, \cdots, m$$

Esta equação descreve o comportamento de uma rede do tipo *feed-forward* totalmente conectada com n entradas, k neurônios ocultos e m neurônios de saída. $O_k(\vec{x})$ é a ativação do neurônio k de saída para a entrada $\vec{x} = x_1, x_2, ..., x_n$. Os coeficientes $c_{j,k}$ representam os pesos entre a camada oculta e a camada de saída. As conexões de curto entre a entrada e a saída são dadas por $d_{i,k}$. b_k é o bias das unidades de saída e P_j é o bias dos neurônios ocultos que determina a característica da função h. A função de ativação dos neurônios de saída é representada por σ.

A grande vantagem do modelo das funções de base radial é a possibilidade de calcular diretamente os coeficientes $c_{j,k}$ (i.e. os pesos entre camada oculta e a de saída) e o bias b_k. Este cálculo requer uma escolha conveniente de centros $\vec{t}_j$ (isto é, os pesos entre a entrada e a camada oculta). Por causa dos pesos das conexões entre a camada de entrada

e a de saída que não podem ser calculados diretamente, deve haver um procedimento especial de treino para redes neurais que usam as funções de base radial.

O procedimento de treinamento tenta reduzir o erro **E** usando o gradiente descendente. É recomendado usar diferentes taxas de aprendizagem para conjuntos diferentes de parâmetros de treino. As equações a seguir contêm toda a informação necessária para procedimento de treino:

$$E = \sum_{k=1}^{m}\left(\sum_{i=1}^{N}\left(y_{i.k} - o_k(\vec{x}_i)\right)^2 \right) \tag{2.47}$$

$$\Delta \vec{t}_j = -\eta_1 \frac{\partial E}{\partial \vec{t}_j} \tag{2.48}$$

$$\Delta \vec{p}_j = -\eta_2 \frac{\partial E}{\partial \vec{p}_j} \tag{2.49}$$

$$\Delta c_{j,k} = -\eta_3 \frac{\partial E}{\partial c_{j,k}} \tag{2.50}$$

$$\Delta d_{i,k} = -\eta_3 \frac{\partial E}{\partial d_{i,k}} \tag{2.51}$$

$$\Delta b_k = -\eta_3 \frac{\partial E}{\partial b_k} \tag{2.52}$$

É freqüentemente útil um coeficiente de *momentum*. Este coeficiente aumenta a taxa de aprendizagem. A equação a seguir descreve o efeito de um coeficiente de movimento no

treinamento de um parâmetro qualquer g dependente do parâmetro adicional μ. Δg_{t+1} é a mudança de g durante o tempo t+1 enquanto Δg_t é a variação durante o tempo t:

$$\Delta g_{t+1} = -\eta \frac{\partial E}{\partial g} + \mu \Delta g_t \tag{2.53}$$

Outra melhora útil no procedimento de treino é a definição de um erro mínimo permitido para os neurônios na camada de saída. Isto previne a rede de receber um treinamento além do necessário, fazendo com que esta perca a capacidade de generalização.

2.6 Self Organizing Feature Finder (SOFF)

2.6.1 Descrição da SOFF

Muitos padrões como os sinais de voz apresentam uma grande variação em suas representações. Sabemos que para um sistema de reconhecimento de voz ser robusto, é necessário que este seja capaz de escolher e separar os padrões e criar seu próprio espaço de *features*, na fase de treinamento. Essa necessidade fez com que sistemas de reconhecimento *self-organized* (auto-organizáveis) aparecessem, entre os quais os modelos de Markov [54], [38]. A capacidade das redes neurais artificiais de criar esses padrões de forma apropriada, as fez se tornarem cada vez mais importantes alternativas às técnicas convencionais de reconhecimento de voz. Este capítulo apresenta uma rede *self-organized* que aprende de forma automática as características espectrais do sinal de voz.

A SOFF é uma rede *self-organized* que aprende características espectrais das amostras de voz, e é capaz de criar um espaço de *features* de forma automática. A técnica baseia-se na aprendizagem das características espectrais da voz, isto é, na extração dos padrões de voz, e no reconhecimento desses padrões. A rede apresenta um modo de treinamento, quando é realizada a aprendizagem dos padrões, e um modo de reconhecimento. No modo de treinamento, a rede SOFF cria um conjunto de "detetores de padrões" chamados de *detetores de características*, que são ativados em certas regiões do espaço espectral que apresentam características invariantes. Cada *detetor de característica* representa um padrão característico do sinal. A regra de aprendizagem é seguida de tal forma a convergir para um estado de estabilidade. No modo de reconhecimento, cada *detetor de característica* calcula uma medida normalizada,

indicando a presença ou não de um padrão da mesma classe do que o padrão de referência correspondente àquele *detetor de característica*. As saídas de todos os *detetores de características* são colocadas em um vetor que é apresentado a uma segunda camada de *detetores de características* da SOFF. A saída dessa segunda camada é apresentada a uma rede neural do tipo MLP. Originalmente, a decisão era feita através de DTW [65] (*Dynamic Time Warping*) mas estudos prévios mostraram que o MLP permite uma decisão mais robusta. A segunda camada da SOFF tem por objetivo tornar a saída da SOFF menos sensivel a pequenas variações na entrada e ao mesmo tempo diminuir a dimensão do espaço de saída. A rede *Self-Organizing Feature Finder* possui algumas semelhanças com outras duas redes. Essas redes são o Mapa de Kohonen [35] e a ART-1 (*Adaptive Resonance Theory*) [47]. A semelhança do Mapa de Kohonen com a SOFF reside no fato de que um número pré-definido de padrões é usado para aproximar a distribuição de probabilidade do sinal de entrada. No entanto, a necessidade de se ter um número fixo de padrões restringe sua capacidade de auto-organização, criando uma topologia também fixa, o que difere da SOFF. A ART-1, cujas entradas são vetores binários, se assemelha à SOFF no tocante à quantidade indeterminada de padrões que são criados sempre que um limiar de semelhança é alcançado. À seguir encontram-se os algoritmos de apredizagem e reconhecimento da SOFF e em seguida a arquitetura da rede.

2.6.2 Algoritmos de Reconhecimento e de Aprendizagem

Durante a fase de reconhecimento, quando um padrão é apresentado à primeira camada da rede, cada *detetor de característica* desta camada apresenta como saída um número que representa a similitude do padrão de entrada com aquele embutido no *detetor*

de característica. As saídas de todos os *detetores de características* são conectadas formando um vetor que é apresentado à segunda camada.

Um *detetor de característica*, isto é um padrão característico do sinal de voz, pode ser representado de forma genérica por:

$$\Im[\mathrm{W(n)}, \mathrm{LIM\_SUP}, \mathrm{LIM\_INF}, \mathrm{N}] \tag{2.54}$$

onde W(n) é um vetor N-dimensional indexado pelo tempo e LIM_SUP e LIM_INF são respectivamente os limites superior e inferior usados durante a fase de aprendizagem.

Cada *detetor de característica* realiza o cômputo da similitude entre seu padrão de referência "residente", W(n), e o padrão de entrada X(n+1). Esta medida, $\eta(n)$, é feita através do co-seno do ângulo entre X(n+1) e W(n).

$$\eta(n) = \frac{W^T(n) \cdot X(n+1)}{\|X(n+1)\| \cdot \|W(n)\|} \tag{2.55}$$

onde $\eta(n) \in [-1,+1]$. O grau de similitude entre X(n+1) e W(n) pode ser interpretado como a similitude entre suas respectivas direções no espaço.

Em cada camada, a aprendizagem é realizada pela apresentação dos padrões de treinamento ao conjunto já existente de *detetores de características*. Esse conjunto de *detetores de características* pode ser vazio se esta for a primeira vez que a rede é treinada. A aprendizagem se dá segundo dois passos principais:

(a) Localiza-se o *detetor de característica,* $\Im$, com o máximo valor de saída da camada, $\Im^*$.

(b.1) Se $\eta(n)^*$ > LIM_SUP, $W^*(n)$ representa o padrão de entrada correspondente e $W^*(n)$ é então reforçado pelo produto com um número, k, k > 1. Nota-se que isto não muda o valor de $\eta(n)^*$. Isto simplesmente faz com que $W^*(n)$ não seja perturbado de sua direção e sim em sua amplitude.

(b.2) Senão, se $\eta(n)^*$ > LIM_INF, $W^*(n)$ é ajustado por uma regra de tal forma a aumentar $\eta(n)^*$. Esta regra é dada por:

$$W^*(n+1) = f(\eta^*(n)) \cdot X(n+1) + \gamma \cdot W^*(n) \tag{2.56}$$

com f(z) > 0, $\forall z \in [0,1], \gamma \in [0,1]$, e η (n) como na equação anterior.

Em [37], mostrou-se que para a função f(z)=0.1z/(1-z), a velocidade de convergência da regra de aprendizagem era otimizada.

(b.3) Senão, se $\eta(n)^*$ < LIM_INF, o padrão representado por X(n+1) não pertence ao conjunto conhecido de padrões. Um novo *detetor de característica* é criado com W(n+1) = X(n+1).

2.6.3 Arquitetura da SOFF

O sinal de voz passa por um pré processamento onde são extraídos os vetores de características da voz, como os *MFC*[3]. Esse vetores são então apresentados à SOFF onde

eles são transformados em representações internas pelas camadas da SOFF. Finalmente, a saída da última camada da rede é apresentada ao MLP para a identificação do padrão.

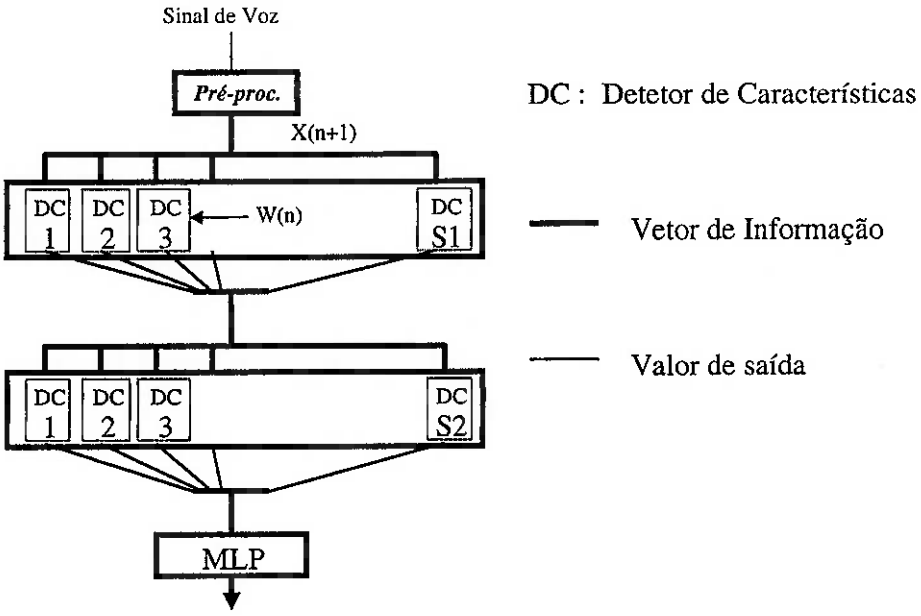


FIGURA 2.6.1 - ARQUITETURA DA SOFF COM S1 DETETORES DE CARACTERÍSTICAS NA PRIMEIRA CAMADA E S2 NA SEGUNDA CAMADA.

2.7 A rede binária de Steinbuck - *LearnMatrix*

A rede *LearnMatrix* foi inventada em 1958 por Karl Steinbuch que lançou um dos primeiros livros sobre neurocomputação [70]. Seu objetivo era produzir uma rede que utilizasse uma versão binária da lei de Hebb [24] para realizar associações entre pares de padrões binários. Esta idéia originou-se na famosa experiência do fisiologista russo I. P. Pavlov no início do século. Nesta descoberta, um cão foi condicionado a salivar ao som de uma campainha, isto é, uma sinal que não apresentava nenhum significado anterior. Pavlov descobriu que se a hora da alimentação do cão era acompanhada de um sinal sonoro, procedimento realizado durante um longo período de tempo, a emissão deste som sem ser acompanhado pela alimentação, produzia salivação, o que se tornou então um reflexo condicionado ao som da campainha. Esses reflexos condicionados deram origem às conecções condicionadas encontradas nos circuitos propostos por Steinbuch, compostos por resistores e ligações. A *LearnMatrix* é uma ferramenta poderosa por causa de sua capacidade de análise e sua simplicidade que permitem realizar associações de forma muito eficiente. O presente capítulo é organizado da seguinte forma: a seção 2.7.1 apresenta uma introdução à lei de Hebb, e na seção 2.7.2, tem-se a descrição funcional e estrutural da *LearnMatrix*. A *LearnMatrix* é comparada ao *Multi-Layer Perceptron* (MLP), no caso específico de reconhecimento automático do locutor.

2.7.1 A lei de Hebb

Em 1949 o psicólogo canadense Donald Hebb publicou um livro intitulado *The organization of behavior*, no qual ele propõe um possível mecanismo de aprendizagem no cérebro. Basicamente, a idéia é que quando a entrada de um neurônio, via uma sinapse, faz com que este imediatamente emita um pulso, então a eficácia desta entrada,

em termos da habilidade da célula em produzir pulsos, deve ser de alguma forma reforçada. Trabalhos posteriores no campo da neurofisiologia mostraram que a idéia proposta por Hebb estava aproximadamente correta. No entanto, é conhecido que existem outros mecanismos de aprendizagem biológica. Para exemplificar a lei de Hebb é discutido a seguir o associador linear. A entrada do associador linear é um vetor x, de dimensão n. A saída correspondente, y', de dimensão m, é dada por:

$$y' = Wx \tag{2.57}$$

onde W é uma matriz $m \times n$ de pesos. A idéia básica de um associador linear é que a rede deve aprender L pares de vetores de entrada/saída (x_1, y_1), (x_2, y_2),...,(x_L, y_L). Quando um vetor x_k é apresentado à entrada da rede, esta deveria ter como saída y_k. O problema de fazer o associador linear associar x_k a y_k, está relacionado com o problema de encontrar a matriz W que realiza esta tarefa. A lei de aprendizagem de Hebb, inserida dentro do contexto da neurocomputação, pode ser expressa por:

$$w_{ij}^{new} = w_{ij}^{old} + y_{ki} \cdot x_{kj} \tag{2.58}$$

Assume-se que antes de apresentar qualquer par de vetores (x_k, y_k) à rede, os pesos w_{ij} são inicializados com valor nulo. O processo de treinamento muda a matriz de pesos W de seu estado inicial para um estado final através de um peso adicional causado pela aplicação da lei de Hebb L vezes. O estado final da matriz de pesos é dado por:

$$W = y_1 x_1^T + y_2 x_2^T + ... + y_L x_L^T \tag{2.59}$$

Com isso, L deve ser menor ou igual a n, a dimensão dos vetores x, e o máximo número de pares de vetores que podem ser associados é limitado a L=*n*. Esta restrição se deve ao fato de que o número máximo de vetores ortogonais linearmente independentes num espaço n-dimensional é n. Isso torna a lei de Hebb bastante limitada em sua capacidade, mas ela pode ser aplicada em várias situações.

2.7.2 A *LearnMatrix*

A figura abaixo apresenta a *LearnMatrix* da forma concebida originalmente por Steinbuch. Os vetores de entradas binárias x são introduzidos como um conjunto de tensões {d}. Os valores das condutâncias devem ser tais que: $g_{ij} = w_{ij}$.

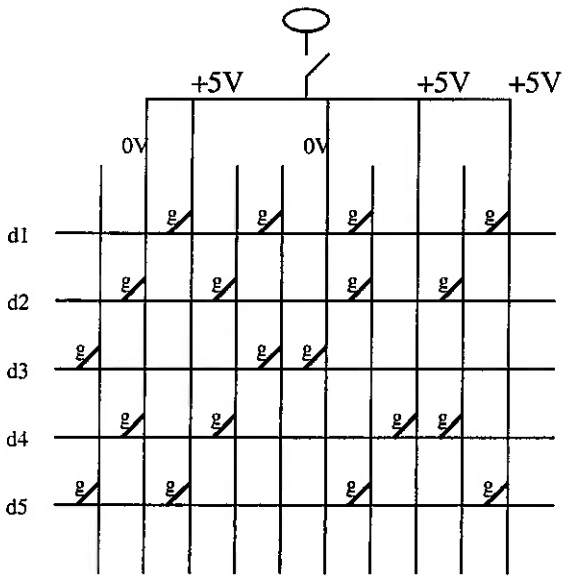


FIGURA 2.7.1 - *LEARNMATRIX* ORIGINALMENTE CONCEBIDA POR STEINBUCH

A estrutura da *LearnMatrix* está apresentada nas figuras a seguir. Dois conjuntos de condutores se interceptam e formam uma estrutura matricial cujos pontos de intercessão se comportam como conecções condicionadas. As linhas verticais conduzem os padrões binários {f}, chamados de *features*. As linhas horizontais conduzem vetores binários, {d}, chamados de indicações *(designations)*. Essas indicações possuem um único bit igual a "1", enquanto todos os outros são "0". Existem dois modos de operação da *LearnMatrix*. O *learning mode*, mostrado na figura 2.7.2, representa a fase de aprendizagem da rede. Inicialmente, todos os pesos, simulados pelos nós da matriz, são zerados. Nesta fase, um padrão pertencente a uma classe C { $f^{(c)}$ } e seu respectivo vetor de indicações { $d^{(c)}$ } são apresentados à rede. Os pesos são então condicionados segundo uma forma booleana da lei de aprendizagem de Hebb, mostrada a seguir:

$$w_{ij} = \begin{cases} 1 & se \quad w_{ij}^{old} = 1 \\ 1 & se \quad f_j d_i = 1 \\ 0 & caso \quad cont. \end{cases} \tag{2.60}$$

sendo que a indicação que representa a classe C é da forma: $d^{(c)}{}_i$ =0..010..0. Por exemplo, no caso de três classes de padrões que se queira representar, o conjunto de indicações{d} pode ser da forma:

$$\begin{aligned} d^{(1)} &= 1\ 0\ 0 \\ d^{(2)} &= 0\ 1\ 0 \\ d^{(3)} &= 0\ 0\ 1 \end{aligned} \tag{2.61}$$

Esta regra de aprendizagem é equivalente a simplesmente copiar o padrão de entrada {f} nos pesos correspondentes à linha horizontal associada ao bit "1" da indicação {d} da referida classe do padrão de entrada {f}. De outra forma, quando $d^{(c)}{}_i = 1$, então, faz-se $w_{ij} = f^{(c)}{}_j$. Pode-se ver então que nesta fase, a aprendizagem se dá igualando-se os pesos correspondentes à linha onde $d^{(c)}{}_i = 1$, aos padrões de referência.

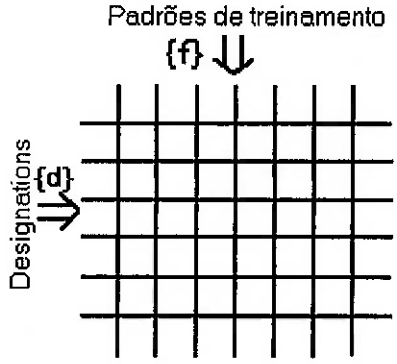


FIGURA 2.7.2 - LEARNING MODE: MODO DE APRENDIZAGEM NO QUAL A PARTIR DA APLICAÇÃO DE UMA INDICAÇÃO {D} E DE UM PADRÃO DE REFERÊNCIA {F}, AS CONEXÕES CONDICIONADAS, ISTO É OS PESOS, SE TORNAM CÓPIAS DO PADRÃO {F}

Quando a fase de aprendizagem é concluída, o *skilled mode* ou *knowing phase* que é a fase quando a rede está apta para realizar associações, pode ser iniciada. Nesta fase, as associações podem ocorrer de duas formas diferentes. A primeira forma, inserida dentro do contexto de um reconhecimento automático de locutor, é a identificação do locutor, apresentada na figura 2.7.3. Quando um vetor de entrada {f} é apresentado à rede, esta emite como saída a indicação mais similar, representando um determinado locutor. Este processo pode ser exemplificado pelas equações a seguir, onde $\oplus$ representa o "ou-exclusivo":

$$d'_i = \begin{cases} 1 & se \quad \sum_j f_j \oplus w_{ij} = \min_k [\sum_j f_j \oplus w_{kj}, \forall k] \\ 0 & caso \quad cont. \end{cases} \tag{2.62}$$

Esta equação significa que o padrão {f} é obtido através de uma comparação entre {f} e {w}. O vetor {d} terá um bit "1" na linha correspondente ao menor número de erros (ou maior número de acertos) entre {f} e {w}, sinalizando a indicação mais similar a {f}.

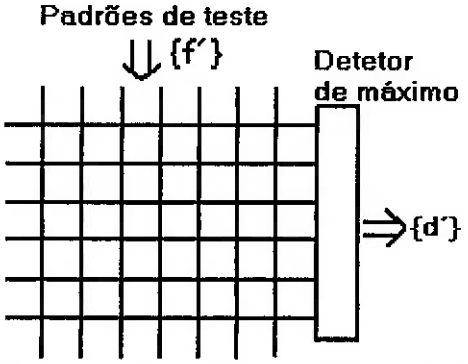


FIGURA 2.7.3 - SKILLED MODE: MODO DE IDENTIFICAÇÃO DO LOCUTOR. UM PADRÃO {F'} É APRESENTADO À REDE E A INDICAÇÃO {D'}, MAIS SIMILAR, É COLOCADA COMO IDENTIDADE DO PADRÃO {F'}.

A segunda forma da *knowing phase,* é a verificação do locutor, apresentada na figura 2.7.4. Quando um vetor {f'} e uma indicação {d} são apresentados à rede, esta realiza comparações entre o vetor {f'} e o padrão de referência {f} correspondente à indicação apresentada {d}. Se {f'} e {f} forem similares, em relação a certos limiares obtidos experimentalmente, o padrão {f'} é verificado e associado ao vetor {d}, senão ele é rejeitado. Este processo pode ser equacionado como segue:

$$\left[\{d_i\},\{f'\}\right]=\begin{cases} verificado \quad se \quad \sum_j f'_j \oplus w_{ij} < Limiar \\ rejeitado \qquad\quad caso \quad contrario. \end{cases} \tag{2.63}$$

Nesta equação, o índice i representa a linha horizontal indicada pelo único bit "1" da indicação {d}. Uma "locução" é então verificada se o número de diferenças entre {f'} e

{f}, isto é, o número de erros, for menor que um certo limiar obtido experimentalmente. Este limiar deve ser um compromisso entre a quantidade de falsa aceitação e falsa rejeição. No caso de um sistema de verificação de locutor, a falsa aceitação é muito mais indesejável do que a falsa rejeição. Este limiar de modo geral é único para todos os locutores.

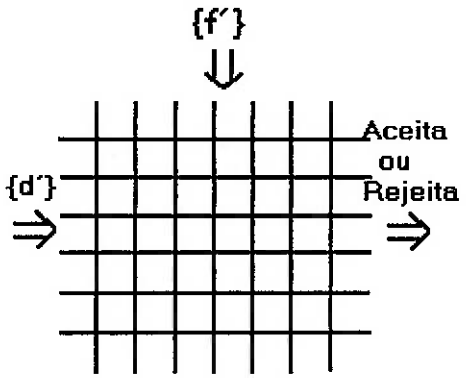


FIGURA 2.7.4 - SKILLED MODE: MODO DE VERIFICAÇÃO DO LOCUTOR.

No modo de verificação do locutor, um padrão de teste {f'} e uma indicação {d'} são apresentadas à rede. Esta, verifica o número de erros entre {f'} e {f} correspondente a {d'}. Se a quantidade de erros for menor que um limiar, então a locução é verificada. Senão, ela é rejeitada.

Em resumo, a *LearnMatrix* determina o padrão mais similar ao conjunto de padrões de entrada, realizando uma detecção de máxima semelhança.

3. Pré-processamento

3.1 Mel Frequency Cepstral Coefficients (MFCC)

Os MFCCs possibilitam uma representação paramétrica da voz. Foi mostrado [44] que esses coeficientes apresentam uma taxa de reconhecimento de palavras muito superior à dos coeficientes LPC (*Linear Prediction Coefficients*) e seus derivados (coeficientes de reflexão e cepstrais) e aos LFCCs (*Linear Frequency Cesptral Coefficients*). Os coeficientes mel-cepstrais têm sido bastante utilizados em reconhecimento de voz por apresentarem informações transicionais e instantâneas.

À seguir encontra-se o diagrama em blocos do sistema que calcula os coeficientes cepstrais baseados na escala Mel.

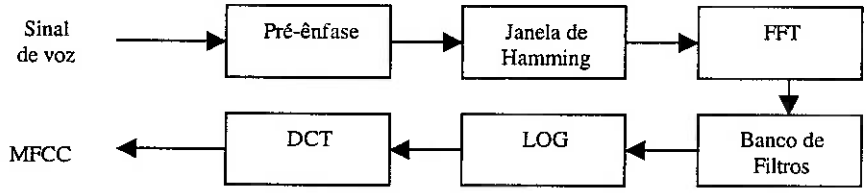


FIGURA 3.1.1 - PROCESSO DE EXTRAÇÃO DO MFCCS

Muitos reconhecedores realizam a pré-ênfase do sinal antes da parametrização. A pré-ênfase faz com que tanto as baixas freqüências como as altas recebam pesos iguais antes da parametrização. Isto é muito desejado para o reconhecimento acurado de consoantes, onde muitas informações espectrais sobre as articulações ocorrem em regiões de freqüência acima de 1KHz .

A pré-ênfase é realizada pela subtração de cada elemento do vetor de entrada pelo elemento anterior multiplicado pelo fator *alfa*

$$y(n) = s(n) - alfa \cdot s(n-1) \tag{3.1}$$

onde y(n) é o sinal resultante no tempo n, s(n) é o sinal original no tempo *n* e *alfa* é um coeficiente.

O sinal passa então por uma janela de Hamming, para evitar distorções espectrais nas bordas da janela. O aspecto da janela de Hamming está mostrado na figura a seguir.

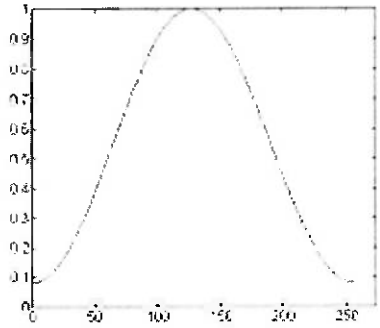


FIGURA 3.1.2 - JANELA DE HAMMING

A equação que define a janela de Hamming é:

$$h(n) = 0.54 - 0.46 \cdot \cos(\frac{2\pi n}{N-1}) \tag{3.2}$$

O banco de filtros permite que o espectro de magnitude do sinal de voz (dado pela magnitude da FFT), seja *frequency-warped* para seguir a escala mel, e *amplitude-warped* numa escala logarítmica. O espectro de magnitude é então passado pelo banco de filtros na escala normal de freqüência e a energia de cada filtro é calculada. Em

seguida, é tomado o logarítmo da energia de cada filtro, para desconvoluír em freqüência os dois sinais (excitação periódica e resposta do aparelho fonador) multiplicados no domínio do tempo. As duas figuras abaixo representam o banco de filtros na escala MEL e na escala normal de freqüência.

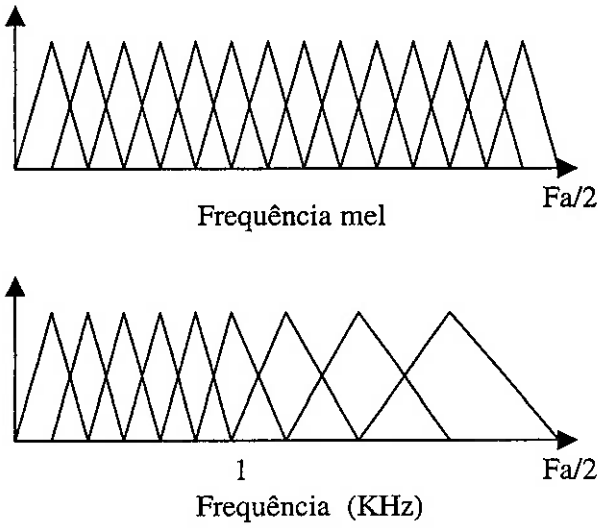


FIGURA 3.1.3 - BANCOS DE FILTROS NA ESCALA MEL (PARTE SUPERIOR) E BANCO DE FILTROS NA ESCALA NATURAL DE FREQÜÊNCIAS (PARTE INFERIOR)

A figura 3.1.4 abaixo mostra a relação entre a freqüência mel e a freqüência natural. Esta relação é dada por:

$$f_{mel} = 2595 \cdot \log_{10}(1 + \frac{f}{700}) \tag{3.3}$$

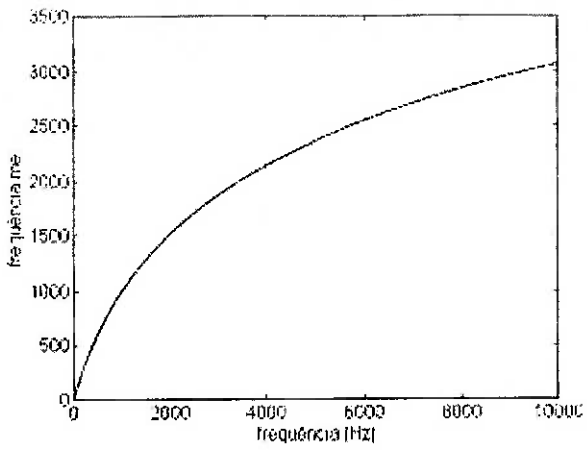


FIGURA 3.1.4 - RELAÇÃO ENTRE A FREQÜÊNCIA MEL E A FREQÜÊNCIA NATURAL

Os primeiros coeficientes da transformada discreta do co-seno das energias logarítmicas dos filtros representam os *Mel Frequency Cepstral Coefficients*, MFCCs.

$$G_x(0) = \frac{\sqrt{2}}{M} \sum_{m=0}^{M-1} X(m) \tag{3.4}$$

A DCT [48] é definida como:

$$G_x(k) = \frac{2}{M} \sum_{m=0}^{M-1} X(m) \cos \frac{(2m+1)k\pi}{2M} \tag{3.5}$$

Onde K=1,2,...,(M-1), X(m), m=0,1,...,(M-1) é o sinal de entrada, e M é o número de filtros usados.

A DCT2 é escolhida para o cálculo dos MFCCs por que ela se aproxima muito da transformada de Karhunen-Loève (KLT) que é tida como uma transformada otimizada em termos de medidas de desempenho como a distribuição de variância.

3.2 Mel Frequency Cepstral Coefficients Corelations (MFC^3)

Os coeficientes chamados de *Mel Frequency Cepstral Coefficients Correlations*, MFC^3, representam as correlações dos MFCCs e foram propostos por Sória e Cabral [SORI96a] e têm sido utilizados para a identificação do locutor. As correlações são definidas segundo as equações abaixo. (Coeficiente de correlação de Pearson (r))

$$r_{xy} = \frac{\frac{\sum (X - \bar{X}) \cdot (Y - \bar{Y})}{n-1}}{S_x \cdot S_y} \tag{3.6}$$

onde rxy é o coeficiente de correlação de Pearson para duas variáveis **X** e**Y** de dimensão n. Sx e Sy são os desvios padrão para **X** e **Y** respectivamente.

As duas equações abaixo mostram apenas outras maneiras de escrever o coeficiente de correlação de Pearson.

$$r = \frac{1}{1-n} \sum \left(\frac{x - \bar{x}}{S_x} \right) \left(\frac{y - \bar{y}}{S_x} \right) \tag{3.7}$$

A equação abaixo mostra a definição do coeficiente de correlação expandido para mostrar como o desvio padrão é calculado para as variáveis X e Y.

$$r = \frac{\sum xy - \frac{1}{n}\left(\sum x\right)\left(\sum y\right)}{(n-1)S_x S_y} \tag{3.8}$$

$$r_{xy} = \frac{\frac{\sum (X-\bar{X})\cdot(Y-\bar{Y})}{n-1}}{\sqrt{\frac{\sum (X-\bar{X})^2}{n-1}\cdot\frac{\sum (Y-\bar{Y})^2}{n-1}}} \tag{3.9}$$

A tabela a seguir mostra um exemplo das correlações entre vetores compostos por cada um dos 10 primeiros MFCCs extraídos ao longo de um sinal de voz. VMFCC0 corresponde ao vetor composto do primeiro coeficientes MFCC extraído ao longo do sinal de voz. A metade superior da matriz corresponde às correlações e a metade inferior corresponde à matriz das variâncias. Essas matrizes são simétricas.

Tabela 3.2.1 - Matriz de correlações e variância dos 10 primeiros MFCCs

	VMFCC0	VMFCC1	VMFCC2	VMFCC3	VMFCC4	VMFCC5	VMFCC6	VMFCC7	VMFCC8	VMFCC9
VMFCC0		-0.549	-0.856	-0.932	0.860	0.774	0.846	0.927	-0.878	-0.733
VMFCC1	0.162		0.797	0.397	-0.652	-0.137	-0.620	-0.623	0.693	0.732
VMFCC2	0.069	0.251		0.809	-0.855	-0.577	-0.912	-0.929	0.936	0.820
VMFCC3	0.073	0.150	0.137		-0.753	-0.803	-0.823	-0.878	0.873	0.692
VMFCC4	0.140	0.136	0.145	0.283		0.617	0.889	0.919	-0.873	-0.579
VMFCC5	0.122	0.235	0.178	0.106	0.405		0.710	0.768	-0.596	-0.433
VMFCC6	0.291	0.394	0.229	0.221	0.347	0.310		0.970	-0.873	-0.626
VMFCC7	0.056	0.209	0.119	0.034	0.207	0.091	0.297		-0.893	-0.665
VMFCC8	0.076	0.196	0.206	0.143	0.364	0.191	0.478	0.160		0.828
VMFCC9	0.061	0.254	0.188	0.179	0.212	0.134	0.460	0.074	0.084	

A metade superior da matriz (acima da diagonal principal) apresenta as correlações entre cada vetor de coeficientes MFCC e os demais vetores de coeficientes (MFCC0 a MFCC9). As correlações entre um vetor de coeficientes e ele próprio não é mostrada na tabela por ser igual a um, portanto a diagonal principal está em branco. A primeira linha, mostrada abaixo, apresenta a correlação entre o primeiro vetor de coeficientes, VMFCC0, com os demais, e indica que o coeficiente de correlação entre VMFCC0 e VMFCC1 é igual a –0.549. Este valor é calculado conforme as equações mostradas anteriormente e corresponde à correlação entre dois vetores de MFCCs de igual tamanho. Este valor, -0.549, é o primeiro MFC^3.

Tabela 3.2.2 - Correlações entre o primeiro vetor de coeficientes, VMFCC0, com os demais.

	VMFCC0	VMFCC1	VMFCC2	VMFCC3	VMFCC4	VMFCC5	VMFCC6	VMFCC7	VMFCC8	VMFCC9
VMFCC0	1.0	-0.549	-0.856	-0.932	0.860	0.774	0.846	0.927	-0.878	-0.733

A metade inferior da matriz (abaixo da diagonal principal) apresenta as variâncias entre cada vetor de coeficientes MFCC e os demais vetores de coeficientes (VMFCC0 a VMFCC9). A primeira linha, mostrada a seguir, apresenta a variância do primeiro vetor de coeficientes, VMFCC0, com os demais.

Tabela 3.2.3 - Variância do primeiro coeficiente, MFCC0, com os demais.

	VMFCC0
VMFCC0	0.0
VMFCC1	0.162
VMFCC2	0.069
VMFCC3	0.073
VMFCC4	0.140
VMFCC5	0.122
VMFCC6	0.291
VMFCC7	0.056
VMFCC8	0.076
MFCC9	0.061

Nota-se que algumas correlações apresentam valores em módulo bastante próximos de um. Verificou-se experimentalmente que estes coeficientes MFC^3 próximos de um se mantiveram estáveis para diferentes locuções do mesmo locutor. Algumas variâncias correspondentes às correlações mais altas se apresentaram próximas de zero. Isto nos levou a crer que os coeficientes MFC^3 poderiam ser utilizados no reconhecimento do locutor já que eles pareceram bastante estáveis para diferentes locutores. Extraindo da tabela mostrada, encontramos um vetor de MFC^3 composto por 45 elementos.

-0.549	**-0.856**	**-0.932**	**0.860**	**0.774**	**0.846**	**0.927**	**-0.878**	**-0.733**
	0.797	**0.397**	**-0.652**	**-0.137**	**-0.620**	**-0.623**	**0.693**	**0.732**
		0.809	**-0.855**	**-0.577**	**-0.912**	**-0.929**	**0.936**	**0.820**
			-0.753	**-0.803**	**-0.823**	**-0.878**	**0.873**	**0.692**
				0.617	**0.889**	**0.919**	**-0.873**	**-0.579**
					0.710	**0.768**	**-0.596**	**-0.433**
						0.970	**-0.873**	**-0.626**
							-0.893	**-0.665**
								0.828

4. Implementação e Testes

4.1 Base de dados utilizada

Os parâmetros utilizados no pré-processamento para representar o locutor foram as correlações dos MFCCs (*Mel-Frequency Cepstral Coefficients)*, chamadas de MFC^3 [SORI96-1,2,3] (MFCC *Correlations*), extraídas de segmentos de voz de curta duração. De modo geral, segmentos de 100ms a 150ms foram utilizados para o reconhecimento. Os MFC^3 têm se mostrado muito estáveis em determinados segmentos de voz, como por exemplo em sons nasalizados. Um estudo foi realizado no sentido de encontrar quais os segmentos de voz que proporcionam as maiores taxas de reconhecimento. Para tal, várias frases foneticamente balanceadas foram selecionadas [3], entre as quais a frase "Amanhã Ligo de Novo" foi escolhida para os testes. Esta frase foi dividida em 80 segmentos de 120ms cada, com uma sobreposição variando entre 100ms e 108ms. Para cada um desses 80 segmentos, os MFC^3 foram calculados com janelas de 23ms e sobreposição de 22ms. Para cada segmento, o número de MFCCs foi variado entre 12 e 15. Com isso, uma matriz simétrica de correlações foi obtida cujo número de elementos independentes variou entre 66 e 105. Um estudo prévio mostrou que os segmentos de número 1 e 40, correspondendo respectivamente aos sons /am/ e /de/ da frase "Amanhã Ligo de Novo", mostraram ser aqueles que levam à maior taxa de reconhecimento. Portanto, estes dois segmentos foram usados nos experimentos. Dez locutores participaram das sessões de gravação, sendo cinco do sexo masculino e cinco do sexo feminino. Cada locutor gravou dez repetições da frase mencionada anteriormente numa mesma sessão. No total 100 locuções foram gravadas. Essas locuções foram divididas em dois conjuntos, um de treinamento e outro de teste. As gravações foram realizadas numa sala bastante quieta. A

aquisição das amostras aconteceu com uma freqüência de amostragem de 11KHz, num computador do tipo PC com placa de som SoundBlaster e microfone direcional.

4.2 Experimentos utilizando o MLP

Neste experimento, uma única rede do tipo MLP foi utilizada para identificar o locutor. Muitas vezes, encontra-se na literatura o uso de uma rede por locutor, prática bastante eficiente para a verificação do locutor. Nos experimentos realizados com uma única rede, havia um neurônio na camada de saída da rede para cada locutor.

O *MLP* utilizado possui três camadas totalmente conectadas. A camada de entrada possui uma dimensão igual à dimensão do vetor de entrada. No caso dos MFC^3, a melhor configuração foi encontrada para 66 elementos, portanto a dimensão da camada de entrada adotada foi de 66 neurônios. A camada oculta teve o número de neurônios variado de forma a otimizar o resultado da rede. A camada de saída foi composta por um número fixo de neurônios, correspondente ao número de locutores.

Os neurônios na camada de entrada apenas repassam os vetores de entrada para os neurônios da camada oculta. Os neurônios nas camadas oculta e de saída possuem a função *sigmoidal*, descrita pela equação a seguir, como função neural. Os parâmetros *a, b, c* e *d* puderam ser variados para adequar a função aos padrões de entrada.

$$y = \frac{a}{1 + e^{(-c\,x)+d}} - b \tag{4.1}$$

onde: x é a entrada do neurônio e y sua saída.

Esta função neural é facilmente derivável, o que é necessário para a implementação do algoritmo *back-propagation.*

Um coeficiente de aprendizado (*learning rate*) e um momento foram utilizados, permitindo com que os mesmos sejam variados ao longo do treinamento. A equação abaixo mostra a forma destes coeficientes.

$$coef = a\, e^{-bt}\, t^{-cn} \tag{4.2}$$

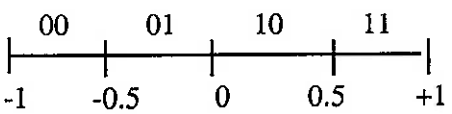
onde: coef é o valor calculado e t é o número da iteração atual.

A rede foi treinada utilizando um conjunto de treinamento e *targets* respectivos. As saídas da rede foram calculadas e os pesos adaptados através do algorítmo *back-propagation*.

4.3 Experimentos utilizando a *LearnMatrix*

Na avaliação da *LearnMatrix* foram utilizados 5 locutores do mesmo sexo, que repetiram 10 vezes a frase apresentada. Cinco locuções foram utilizadas no treinamento e cinco no teste.

Uma vez realizada a extração dos MFC^3, foi necessário quantizar e codificar as correlações, pois estas variam entre -1 e +1. Dois modos simples de codificação foram aplicados. Inicialmente, codificou-se os MFC^3 em 1 bit. Isto é, as correlações positivas foram atribuídas ao bit "0", enquanto as negativas ao bit "1". Num segundo experimento, codificou-se as correlações em quatro níveis quânticos, 2 bits, da forma:



Na fase de aprendizagem da rede, *learning mode*, a rede foi treinada com a codificação binária dos vetores média dos MFC^3. Isto quer dizer que haviam cinco indicações e a *LearnMatrix* era 5×66 no caso de 1 bit ou 5×132, no caso de 2 bits. Nestes experimentos foram utilizados 12 MFCCs que correspondem a 66 elementos independentes na matriz simétrica de correlações. O teste foi realizado apresentando à rede o conjunto de treinamento, já que este foi realizado com o vetor média, e o conjunto de teste. A *LearnMatrix* foi comparada a uma rede neural artificial do tipo MLP com uma camada interna. O MLP foi treinado com os vetores média da mesma forma que a *LearnMatrix*, só que com os valores originais dos MFC^3. O processo de identificação do locutor foi realizado ao longo da frase "Amanhã ligo de novo" e o resultado está apresentado na figura 4.3.1. A curva de cima representa a evolução da trajetória da taxa de reconhecimento ao longo da frase para o MLP, e a curva de baixo o mesmo para a *LearnMatrix*.

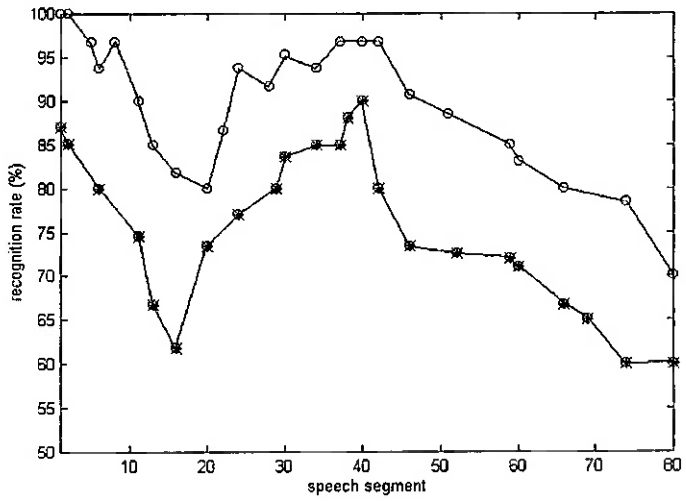


FIGURA 4.3.1 - TRAJETÓRIA DA TAXA DE RECONHECIMENTO AO LONGO DA FRASE "AMANHÃ LIGO DE NOVO" PARA O MLP (TRAJETÓRIA SUPERIOR) E PARA A *LEARNMATRIX* (TRAJETÓRIA INFERIOR), COM CODIFICAÇÃO EM 1 BIT.

Como pode-se ver na figura 4.3.1, os segmentos 1 e 40 apresentam a maior taxa de acerto na identificação. Por isso, esses dois segmentos foram utilizados para comparar a codificação por 1 bit com a de 2 bits. Os resultados encontram-se na tabela 4.3.1.

Tabela 4.3.1 - Comparação entre a codificação em 1 bit e em 2 bits usando a *LearnMatrix* e o MLP, para os segmentos 1 e 40.

	Conjunto de treinamento		Conjunto de teste	
segmento	01	40	01	40
1 bit	83.4%	86.7%	86.7%	93.4%
2 bits	96.7%	93.4%	93.4%	93.4%
MLP	100%	100%	100%	93.4%

Esses resultados mostram que as taxas obtidas com a *LearnMatrix* com codificação de 2 bits são bem próximas àquelas obtidas com o MLP.

4.4 Experimentos utilizando a SOFF

A rede SOFF foi aplicada ao reconhecimento automático do locutor. Os parâmetros utilizados no pré-processamento para representar o locutor foram as correlações dos MFCCs extraídas de segmentos de voz de curta duração. De modo geral, segmentos de 100ms a 150ms foram utilizados para o reconhecimento. Os testes foram realizados em cima do conjunto de dez locutores de ambos os sexos. As amostras foram separadas em dois conjuntos, um de teste e outro de treinamento.

O conjunto de treinamento dos MFC^3 foi apresentado à primeira camada da rede. Dez *detetores de características* foram inicializados com os vetores média dos MFC^3. Inicialmente, existia um *detetor de características* para cada locutor. Os limiares LIM_SUP e LIM_INF foram variados de tal forma que a habilidade da SOFF em criar

novos *detetores de características* fosse julgada com relação à taxa de reconhecimento. O limiar LIM_INF controla o número de *detetores de características* criados. Quanto maior o valor de LIM_INF, maior se torna o número de *detetores de características* criados. A quantidade de *detetores de características* na segunda camada corresponde à dimensão de saída da SOFF e à quantidade de neurônios na camada de entrada do MLP. Após o treinamento da rede, cada vetor de teste foi apresentado à entrada da primeira camada da rede. A saída da SOFF, um vetor de dimensão reduzida, foi apresentado à entrada do MLP. A quantidade de neurônios na camada interna do MLP foi variada de tal forma a se obter a melhor taxa de reconhecimento. As características da voz utilizadas nos testes foram as 66 correlações independentes dos 12 primeiros coeficientes mel-cepstrais. Os resultados estão apresentados na tabela 4.4.1. A melhor taxa de acerto foi obtida quando da criação de 38 outros *detetores de características* além dos 10 criados na inicialização, totalizando 48 *detetores de características*. Deve se notar que para um total de 18 *detetores de características* e 18 neurônios na camada interna do MLP, a taxa de acerto ficou elevada, o que mostra que uma otimização da SOFF pode levar a uma rede MLP pequena. A redução da dimensão do vetor de entrada do MLP neste caso ficou em aproximadamente 72,7%, o que representa uma taxa de redução bastante elevada.

Tabela 4.4.1 - Resultados da identificação do locutor para o conjunto de dez locutores

Nº de MFCCs	Dimensão na camada de entrada da SOFF (Nº de MFC^3)	Dimensão na camada de saída da SOFF	Nº de neurônios na camada interna do MLP	Taxa de Identificação para o conjunto de teste (%)
12	66	48	40	100
12	66	18	18	96,3
12	66	18	15	96,3
12	66	18	10	88,9
12	66	10	10	88,9

4.5 Experimentos utilizando a RBF

Contrariamente ao MLP, foi adotada uma única rede para cada um dos 10 locutores. No processo de reconhecimento do locutor utilizando uma rede por locutor, existem dois tipos de classes a serem separadas, a classe do locutor relativo à rede e a classe correspondente ao restante da população de locutores. A *RBF* utilizada é composta de três camadas. A camada de entrada possui 66 neurônios pois esta é a dimensão dos vetores de entrada dos MFC^3. O número de neurônios da camada oculta variou a partir do número de classes. A camada de saída é composta de um único neurônio representando o locutor associado à rede.

A função neural adotada na camada de entrada é a função linear unitária. Desta forma a primeira camada apenas repassa os valores de entrada adiante. A função *Gaussiana* foi utilizada na camada oculta, conforme equação a seguir.

$$y = e^{-\frac{\|x-c\|^2}{2\sigma^2}} \tag{4.3}$$

onde y é a saída do neurônio, x é a entrada do neurônio, c é o centro da função Gaussiana correspondente e σ corresponde à abertura (raio) da função *Gaussiana*.

O algoritmo *LBG* (*Linde, Buzo and Gray*) [39] foi utilizado para determinar a localização dos centros das *Gaussianas*. As aberturas das *Gaussianas* foram configuradas de forma fixa para cada uma das classes existentes. O valor da abertura corresponde à variância da matriz de distâncias dos centros obtidos das funções de base.

A função neural da camada de saída foi a função *sigmoidal* descrita pela equação a seguir.

$$y = \frac{a}{1 + e^{(-c\,x\,)+d}} - b \tag{4.4}$$

onde: x é a entrada do neurônio e y sua saída. Os valores dos coeficientes da função anterior foram os seguintes: a=1.0, b = 0.0, c=1.5 e d=-2.5.

A rede foi treinada e o erro para cada uma das saídas foi calculado. Os pesos entre a camada oculta e a camada de saída foram adaptados segundo a regra de treinamento do *Perceptron*.

O algoritmo LBG [39] foi utilizado para criar um *codebook* para cada locutor a ser identificado pelo sistema neural baseado no conjunto de treinamento de cada locutor e um *codebook* associado à população em geral de locutores. Os centroides correspondentes à população de locutores em geral é portanto utilizado em todos os modelos dos locutores. As tabelas 4.5.1 e 4.5.2 apresentam as configurações das redes do tipo RBF utilizadas nos experimentos e os melhores resultados.

Tabela 4.5.1 – Configuração das redes do tipo RBF utilizadas nos testes.

Nº de neurônios na camada de entrada	Nº de RBFs na camada oculta		Nº de neurônios na última camada
	Nº de centroides comuns a todos os locutores	Nº de centroides de cada locutor	
66	512	128	1
66	512	64	1
66	512	32	1
66	0	32	1
66	0	5	1
66	0	1 (média)	1

Tabela 4.5.2 – Melhores resultados obtidos nos testes das redes do tipo RBF, com uma rede por locutor.

Neurônios na camada de entrada	Nº de RBFs na camada oculta		Neurônios na última camada	Taxa de acerto (conjunto treino)	Taxa de acerto (conjunto teste)	Falsa rejeição	Falsa aceitação
	Centroides comuns a todos os locutores	Centroides de cada locutor					
66	512	64	1	100%	80%	15%	5%

5. Descrição dos resultados

Como pôde-se ver, a *LearnMatrix* é uma rede puramente associativa, extremamente simples em relação ao MLP e cujos resultados podem ser próximos aos do MLP. A *LearnMatrix* apresenta muitas vantagens em relação a uma rede neural mais complexa, como a do tipo MLP. Enquanto pode-se levar muito tempo para treinar uma rede neural, o processo de aprendizagem da *LearnMatrix* é praticamente imediato, já que na *learning phase* a rede simplesmente copia os padrões de referência nos respectivos pesos. A figura 4.3.1 mostra que a trajetória da taxa de reconhecimento para a rede binária tem a mesma forma do que a do MLP. Com isso, a *LearnMatrix* se torna muito interessante na tarefa de uma análise prévia da taxa de acerto para diferentes segmentos de voz, já que o resultado final com a rede binária pode ser obtido muito rapidamente.

A SOFF, uma rede ainda pouco utilizada mas que apresenta características muito importantes, foi apresentada e testada no reconhecimento do locutor. A característica mais interessante da SOFF é sua habilidade em aprender de forma automática e não supervisionada padrões espectrais a partir de um conjunto de treinamento.

A rede *Self-Organizing Feature Finder* apresentou bons resultados no reconhecimento do locutor para um conjunto fechado de locutores. A SOFF mostrou-se muito útil na redução da dimensão dos vetores de características da voz. A redução em 72.7% da dimensão de entrada levou a uma taxa de 96.3% de reconhecimento, o que faz da SOFF uma ferramenta poderosa na redução da dimensão mantendo uma boa taxa de reconhecimento.

Os experimentos realizados com a RBF mostraram que o ajuste da rede é bastante complexo e os resultados se apresentaram inferiores aos das outras redes. O número de centroides representando cada locutor foi variado de 1 a 128 mas a única configuração que apresentou uma taxa de acerto razoável para o conjunto de teste foi para 64 centroides. As outras configurações, apesar de apresentarem taxa de acerto perto de 100% para o conjunto de treinamento revelaram uma taxa de acerto muito baixa para o conjunto de teste. Para a configuração apresentada na tabela 4.5.2, as taxas de falsa rejeição e falsa aceitação ficaram bastante elevadas, no entanto a falsa aceitação que é mais indesejável ficou três vezes menor que a falsa rejeição.

A tabela 5.1 mostra um resumo dos principais resultados obtidos nos experimentos.

Tabela 5.1 – Resumo dos resultados na identificação do locutor para o segmento nº 1

Rede Neural	Nº de Locutores	Nº de MFCCs	Dimensão da camada de entrada (Nº de MFC^3)	Dimensão da camada de saída	Taxa de Identificaçã para o conjunto de teste (%)
SOFF	10	12	66	10	100
RBF	10	12	66	1	80
MLP	5	12	66	10	100
LM	5	12	66	10	93.4

6. Conclusões

Este trabalho que trata da aplicação de quatro diferentes tipos de redes neurais ao reconhecimento automático do locutor utilizando as correlações entre os MFCCs mostra algumas conclusões importantes.

- Os MFC3 foram aplicados com êxito e representam uma técnica de pré-processamento bastante interessante já que esta técnica elimina a necessidade de alinhamento temporal.
- O MLP mostrou-se um paradigma neural muito poderoso, permitindo taxas de acerto elevadas.
- A rede binária de Steinbuck, *LearnMatrix*, constitui uma *ferramenta* neural importante para a prévia determinação da taxa de reconhecimento com outros paradigmas neurais mais poderosos, tais como o MLP, a RBF ou a SOFF. A *LearnMatrix* possibilita de forma muito rápida a determinação do comportamento de determinado segmento de voz no reconhecimento do locutor. A forma de codificação das amostras de voz deve ser estudada e variada para se obter os resultados esperados.
- A rede SOFF, destaca-se das outras por ser auto-organizável. Ela apresentou resultados muito próximos do MLP e mostrou-se capaz de reduzir bastante a dimensão do espaço de entrada. O desempenho da SOFF está ligado diretamente à escolha correta dos limiares que determinam a criação de novos *detetores de características* e ao reforço daqueles já criados.
- As RBFs mostraram ter um desempenho inferior ao do MLP, no entanto com um tempo de treinamento menor. A dificuldade em se encontrar um modelo que

efetivamente maximize a taxa de acerto no reconhecimento do locutor está ligada ao posicionamento adequado dos centros das funções de base radial, à definição da região de ativação desses centros e à escolha adequada dos *codebooks* que representam os locutores. No entanto, as RBF constituem um paradigma neural cada vez mais utilizado e pesquisado e que pode ser aplicado com sucesso ao reconhecimento do locutor.

- A metodologia de orientação a objeto para o desenvolvimento de redes neurais utilizando UML e JAVA foi utilizada com sucesso e mostrou ser bastante importante para o desenvolvimento de sistemas neurais mais complexos e flexíveis. O tempo de desenvolvimento mostrou-se bem menor do que para outras linguagens de programação e sua natureza já orientada a objeto permitiu criar um sistema facilmente expansível a outros modelos neurais.

À seguir, encontram-se sugestões para o prosseguimento dos trabalhos.

- Aplicar os MFC3 em outras bases de dados com caraterísticas diferentes
- Realizar uma comparação efetiva entre os MFC3 e os MFCCs
- Utilizar as MTIs – *Minimal Temporal Information* [71], para a comparação com os MFC3 nos diferentes modelos neurais. As MTIs são conjuntos de estruturas temporais formadas por segmentos consecutivos obtidos a partir dos MFCCs.
- Aplicar uma codificação mais complexa à *LearnMatrix* e compará-la com os outros modelos neurais.
- Com relação à SOFF devem ser investigados outras métricas de comparação incorporadas aos *detetores de características*.

- Os métodos de treinamento que também determinam o posicionamento dos centros e as regiões de ativação de forma adaptiva devem ser investigados para a RBF.

ANEXO – Publicações

[SORI96a] SÓRIA, R. A. B., CABRAL, E. F. Jr, *Speaker Recognition With Artificial Neural Networks and Mel-Frequency Cepstral Coefficients Correlations*. Proceedings VIII European Signal Processing Conference - EUSIPCO'96, Trieste, Italy, September 10-13, 1996.

[SORI96b] SÓRIA, R. A. B., CABRAL, E. F. Jr, *Combining Neural Networks Paradigms and Mel Frequency Cepstral Coefficients Correlations in a Speaker Recognition Task*. Proc. VII Annual International Conference on Signal Processing Applications and Technology - ICSPAT '96, Boston, Massachusetts, USA, 7-10 October, 1996,

[SORI96c] SÓRIA, R. A. B., CABRAL, E. F. Jr, *Comparison of Different Neural Paradigms in a Speaker Recognition Task Using Mel-Frequency Cepstral Coefficients Correlations*. VII Simpósio Brasileiro de Microondas e Optoeletrônica e XIV Simpósio Brasileiro de Telecomunicações - TELEMO'96, Curitiba, 22-25 de Julho de 1996, Vol. 2, p. 521-526

[SORI99] CABRAL JR., E. F.; SÓRIA R. A B., *Redes Neuras Artificiais – Um Curso Teórico e Prático para Engenheiros e Cientistas*, Edição dos autores, São Paulo. Jun/1999, ISBN 85-900933-1-X, p. 67-78, p. 267-274.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] – ASPRAY W., BURKS A., **Papers on John Von Neumann on Computing and Computer Theory**, Charles Babbage Institute Reprint Series for The History of Computing, Vol. 12 Cambridge, MA: MIT Press, 1986.

[2] - ATAL, B.S. Automatic recognition of speakers from their voices. **Proceedings of the IEEE**, v.64, n.4, p. 460-475, Apr. 1976.

[3] - BEZERRA, M.R, CABRAL JR, E.F. **Determinação de frases de aplicação forense para o projeto NESPER e tese de mestrado IME/94 com base em estudos fonéticos**. São Paulo, EPUSP, 1994. (Boletim Técnico da USP. Departamento de Engenharia Eletrônica, BT/PEE94-01).

[4] - BISHOP C. Improving the generalisation properties of radial basis function neural networks. **Neural Computation**, 3(4): p. 579-588, 1991.

[5] - BREIMAN L., FRIEDMAN J. Predicting multivariate responses in multiple linear regression. **Technical report**, Department of Statistics, University of California, Berkeley, 1994.

[6] - BROOMHEAD D.S. and LOWE D. Multivariate functional interpolation and adaptive networks. **Complex Systems**, 2, p. 321-355, 1988.

[7] - CABRAL JR., E. F. **Redes Neurais Artificiais – Um Curso Teórico e Prático para Engenheiros e Cientistas**. Edição dos autores, São Paulo, Jun/1999. ISBN 85-900933-1-X.

[8] - CABRAL JR., E.F.; TATTERSAL, G.D. Trace-segmentation of isolated utterances for speech recognition. In: INTERNATIONAL CONFERENCE ON ACOUSTICS, SPEECH, AND SIGNAL PROCESSING, Detroit, 1995. **Proceedings**. Piscataway, IEEE, 1995. v.1, p. 365-8

[9] - CASSELMAN L.F., FREEMAN D.F., KERRINGAND.A., LANE S.E., MILLSTROM N.H., NICHOLS W.G., A Neural Network-Based Passive Sonar Detection and Classification Design with a Low False Alarm Rate, **IEEE Conf. On Neural Networks** for Ocean Engineering, p. 49-55, WA, DC., 1991.

[10] - CHEN S., COWAN C.F.N., GRANT P.M. Orthogonal least squares learning for radial basis function networks. **IEEE Transactions on Neural Networks**, 2(2): p. 302-309, 1991.

[11] - COHEN M., FRANCO H., MORGAN N., RUMELHART D. and ABRASH V., Context-Dependent Multiple Distribution Phonetic Modeling with MLPs. In advances in **Neural Information Processing Systems**, p. 649-657, San Mateo, CA: Morgan Kaufmann, 1993

[12] - CORSI, P. Speaker recognition: a survey. In: **PROCEEDINGS** OF THE NATO ADVANCED STUDY INSTITUTE, Bonas, 1981. Automatic Speech Analysis and Recognition, Dordrecht, D.Reidel Publishing company, 1982. p. 277-308

[13] - COVER, T.M. Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. **IEEE Transactions on Electronic Computers**, v.14, n.3, p. 326-334, June 1965.

[14] - DELLER JR., J.R. et al. **Discrete-time processing of speech signals**. New York, MacMillan, 1993.

[15] - DODDINGTON, G. R., Speaker Recognition – Identifying People by their Voices, **Proc. IEEE**, vol. 73, N°11, Nov. 1985

[16] - FOWLER, M. **UML Distilled - Applying The Standard Object Modeling Language**. Addison Wesley, Jan. 1997

[17] - FURUI, S. Cepstral analysis technique for automatic speaker verification. IEEE **Transactions on Acoustics, Speech and Signal Processing**. v.29, n.2, p. 254-272, Apr. 1981.

[18] - FURUI, S. **Digital speech processing, synthesis, and recognition**. New York, Marcel Dekker, 1989.

[19] - GEMAN S., BIENENSTCK E., DOURSAT R. Neural networks and the bias/variance dilemma. **Neural Computation**, 4(1): p. 1-58, 1992.

[20] - GISH, H.; SCHMIDT, M. Text-independent speaker identification. **IEEE Signal Processing Magazine**, v.11, n.4, p. 18-32, Oct. 1994.

[21] - GROSSBERG, S. How does a brain build a cognitive code? **Psychological Review**. n.87, p. 1-51, 1980.

[22] - HAYKIN, S. **Neural Networks-A comprehensive Foundation**, IEEE Press, 1994.

[23] - HAYKIN S., DENG C., Classification of Radar Clutter Using Neural Networks, **IEEE Trans. on Neural Networks** 2, p. 589-600, 1991.

[24] - HEBB, D.O. **The organization of behavior: a neuropsychological theory**. New York, John Wiley, 1949.

[25] - HECHT-NIELSEN R., **Neurocomputing**, Addison-Wesley Publishing Company, 1990.

[26] - HERTZ J., KROUGH A., PALMER R.G. **Introduction to the Theory of Neural Computation**. Addison Wesley, Redwood City, CA, 1991.

[27] - HOPFIELD, J.J. Neural networks and physical systems with emergent collective computational abilities. **Proceedings of the national Academy of Sciences of the U.S.A.**, n.79, p. 2554-2558, 1982.

[28] - HORN R.A., JOHNSON C.R.. **Matrix Analysis**. Cambridge University Press, Cambridge, UK, 1985.

[29] - JACOBSON, I.; BOOCH, G.; RUMBAUGH, J. **The Unified Software Development Process**. Addison Wesley, Jan. 1999

[30] - JACOBSON, I.; BOOCH, G.; RUMBAUGH, J. **The Unified Modeling Language User Guide**. Addison Wesley, Jan. 1999

[31] - JANKOWSKI Jr., C.R.; VO, H.H.; LIPPMANN, R.P. A comparison of signal processing front ends for automatic word recognition. **IEEE Transactions on Speech and Audio Processing**, v.3, n.4, p. 286-293, July 1995.

[32] - JORDAN M.I., JACOBS R.A. Learning to Control an Unstable System with Forward Modeling, In **Advances in Neural Processing Systems**, p. 324-331. San Mateo, CA:Morgan Kaufmann, 1990.

[33] - KOHONEN T., Correlation Matrix Memories, **IEEE Trans. On Computers** C-21, p. 353-359,1972.

[34] - KOHONEN, T. Self-organized formation of topologically correct feature maps. **Biological Cybernetics**. n.43, p.59-69, 1982.

[35] - KOHONEN T., **Self Organization and associative Memories**, (Springer-Verlag, Berlin, 1988).

[36] - LEE, H.S, HAHN, M. Development of a Real-Time Endpoint Detection Algorithm. In: INTERNATIONAL CONFERENCE ON SIGNAL PROCESSING APPLICATIONS AND TECHNOLOGY, Santa Clara, 1993. **Proceedings**. Newton, DSP Associates, 1993. v.2, p. 1547-52

[37] - LERNER S. Z., DELLER J. R. Jr., Speech Recognition by Self-Organizing Feature Finder, **International Journal of Neural Systems**, Vol. 2, Nos. 1 & 2, p. 55-78, 1991.

[38] - LEVINSON S. E., Structural methods in automatic speech recognition, **Proc IEEE** 73, 1985, p. 1625-1650.

[39] - LINDE, Y.; BUZO, A.; GRAY, R.M. An algorithm for vector quantizer design. **IEEE Transactions on Communications**. v.28, n.1, p. 84-95, Jan. 1980

[40] - LINFORD, P.W.; TATTERSALL, G.D. Non-linear time normalization of utterances for speech recognition using MLP's. **Proceedings of the Institute of Acoustics**, v.12, part 10, p. 291-297, 1990.

[41] - LIPPMANN, R.P. An introduction to computing with neural nets. **IEEE ASSP Magazine**, v.4, n.2, p. 4-22, Apr. 1987.

[42] - LIPPMANN, R.P. Pattern classification using neural networks. **IEEE Communications Magazine**, v.27, n.11, p. 47-64, Nov. 1989.

[43] - McCULLOCH, W.S.; PITTS, W. A logical calculus of ideas immanent in nervous activity. **Bulletin of Mathematical Biophysics**. n.5, p. 115-133, 1943.

[44] - MERMELSTEIN P., DAVIS, B., Comparison of Parametric Representations for Monosyllabic Word Recognition in Continuously Spoken Sentences, **IEEE Trans. on ASSP**, Vol. 28, No. 4, p. 357-366, Aug. 1980.

[45] - MINSKY, M.L. **Theory of neuro-analog reinforcement systems and its application to the brain-model problem**. Princeton, Thesis (PhD) - Princeton University.

[46] - MINSKY, M.L.; PAPERT, S.A. **Perceptrons**. Cambridge, MIT Press, 1969.

[47] - MOORE B, ART-1 and Pattern Clustering, in **Proc. 1988 Connectionists Models Summer School**, eds, San Mateo, CA, 1988, p. 174-185.

[48] - NATARAJAN T. e AHNED N., Discrete Cosine Transform, **IEEE Transactions on Computers**, January 1974.

[49] - O'SHAUGHNESSY, D. Speaker recognition. **IEEE ASSP Magazine**, v.3, p. 4-17, Oct.1986.

[50] - O'SHAUGHNESSY D., **Speech Communication. Human and Machine**, Addison-Wesley Publishing Company, 1987.

[51] - PICONE, J.W. Signal modeling techniques in speech recognition. **Proceedings of the IEEE**, v.81, n.9, p. 1215-1247, Sept. 1993.

[52] - PRESS W.H., TEUKOLSKY S.A., VETTERLING W.T, FLANNERY B.P. **Numerical Recipes in C**. Cambridge University Press, Cambridge, UK, second edition, 1992.

[53] - RABINER, L.R. Applications of voice processing to telecommunications. **Proceedings of the IEEE**, v.82, n.2, p. 197-228, Feb. 1994.

[54] - RABINER L.R., HUANG B. H. , An Introduction to Hidden Markov Models, **IEEE ASSP Magazine3**, 1986, p. 4-16.

[55] - RABINER, L.R.; SCHAFER, R.W. **Digital processing of speech signals**. Englewood Cliffs, Prentice Hall, 1978.

[56] - REYNOLDS J., TARASSENKO L., Isolated Word Recognition With Radial Basis Function Classifier. **2nd International Conference on Artificial Neural Networks**, Bournemouth, UK.

[57] - RITTER, H., MARTINETZ, T., SCHULTEN, K., **Neural Computation and Self-Organizing Maps – An Introduction**, Addison-Wesley Publishing Company, 1992.

[58] - ROBINSON, D. A., Signal Processing by Neural Networks in the Control of Eye Movements, **Computational Neuroscience Symposium**, p. 73-78. Indiana University-Purdue University.

[59] - ROSENBERG, A.E. Automatic speaker verification: a review. **Proceedings of the IEEE**, v.64, n.4, p. 475-486, Apr. 1976.

[60] - ROSENBLATT, F. The perceptron: a probabilistic model for information storage and organization in the brain. **Psychological Review**. n.65, p. 386-408. 1958.

[61] - RUMELHART, D.E, HINTON G. E., WILLIAMS R. J., Learning Representations by Back-Propagation Errors, **Nature** (London) 323, p. 533-536 1986.

[62] - RUMELHART, D.E, McCLELLAND, J.L. **Parallel distributed processing**. San Diego, MIT Press, 1986.

[63] - RUNSTEIN, F.; VIOLARO, F.; NUNES, H.F. Uso de diferentes parâmetros de entrada em u sistema de reconhecimento de fala baseado em redes neurais. In: SIMPÓSIO BRASILEIRO DE TELECOMUNICAÇÕES, 13°, **Proceedings**, Aguas de Lindóia, 1995. Anais. Campinas, Unicamp, 1995. p. 155-160

[64] - SÄCKINGER E., BOSER B.E., JACKEL L.D., LECUN Y. and BROMLEY J., Application of the ANNA Neural Network Ship to High-Speed Character Recognition, **IEEE Trans. On Neural Networks** 3, p. 498-505, 1992.

[65] - SAKOE H., CHIBA C., Dynamic programming algorithm optimization for spoken word recognition, **IEEE Trans. Acoust. Speech and Signal Process**. 27. 1978.

[66] - SAMBUR, M.R. Selection of acoustic features for speaker identification. **IEEE Transactions on Acoustics, Speech and Signal Processing**, v.23, n.2, p. 176-182, Apr. 1975.

[67] - SCHAFER, R.W.; RABINER, L.R. Digital representations of speech signals. **Proceedings of the IEEE**, v.63, n.4, p. 662-677, Apr. 1975.

[68] - SCHAFER, R.W.; RABINER, L.R. System for automatic formant analysis of voiced speech. **The Journal of the Acoustical Society of America**, v.47, n.2, p. 634-648, 1970.

[69] - SPECHT D. F., **IEEE Trans. on Neural Networks**, 1: p. 111-121, 1990.

[70] - STEINBUCH K., **Die Learnmatrix** - The Beginning of Associative Memories, Advanced Neural Computers, p. 21-29, Elsevier Science Publishers B.V. (North-Holland), 1990.

[71] - TIMOSZCZUC, A. P., CABRAL JR., E. F. Reconhecimento automático do locutor com redes neurais artificiais do tipo radial basis function minimal temporal information. **Boletim Técnico da Escola Politécnica da USP. Departamento de Engenharia Eletrônica, BT/PEE/9936. São Paulo,EPUSP, 1999.**

[72] - WIDROW, B. Generalization and Information storage in Networks of adaline Neurons. In **Self Organizing Systems** (M.C. Yovitz. G.T. Jacobi and G. D. Goldstein, eds.) p. 435-461. Washington, D.C.: Sparta,1962.

[73] - WIDROW, B. HOFF JR, M.E. **Adaptive switching circuits**. IRE WESCON. Convention Record, v.4, p. 96-104, 1960.

[74] - WILPON, J.G.; RABINER, L.R. A modified K-means clustering algorithm for use in isolated word recognition. **IEEE Transactions on Acoustics, Speech and Signal Processing.** v.33, n.3, p. 587-594, June 1985.

[75] - WOLF, J.J. Efficient acoustic parameters for speaker recognition. **The Journal of the Acoustical Society of America**, v.51, n.6, p. 2044-2056, 1972.

APÊNDICE – Modelo Orientado a Objeto usando UML e código escrito em JAVA.

Esta seção mostra o modelo do sistema de rede neural utilizado neste trabalho usando UML – *Unified Modeling Language* [29], [30], [16]. Inicialmente são apresentados dois diagramas de classes de *design* mostrando a arquitetura do sistema através de suas classes, generalizações, especializações e associações. Cada classe é especificada pelos seus atributos, representando o estado do objeto, e pelos seus métodos, representando o comportamento. Por exemplo, a classe *RBFNetwork* é uma especialização da classe *MultiLayerNeuralNetwork*, assim como a classe *LearningRate* é uma especialização de *TrainingCoefficients*. Em seguida são apresentados dois diagramas de sequência mostrando dois cenários diferentes. O primeiro é o cenário da etapa *feed-forward* da rede. O segundo mostra a etapa de treinamento da rede. Estes diagramas de seqüência mostram os objetos envolvidos no cenário e as mensagens entre os objetos. O objetivos principal destes diagramas é especificar a seqüência das mensagens Finalmente é apresentado o código fonte escrito em JAVA do sistema neural. A ferramente utilizada para modelar o sistema foi o *Rational Rose* e o *Visual J++* foi utilizado para codificar.

Diagrama de classes 1

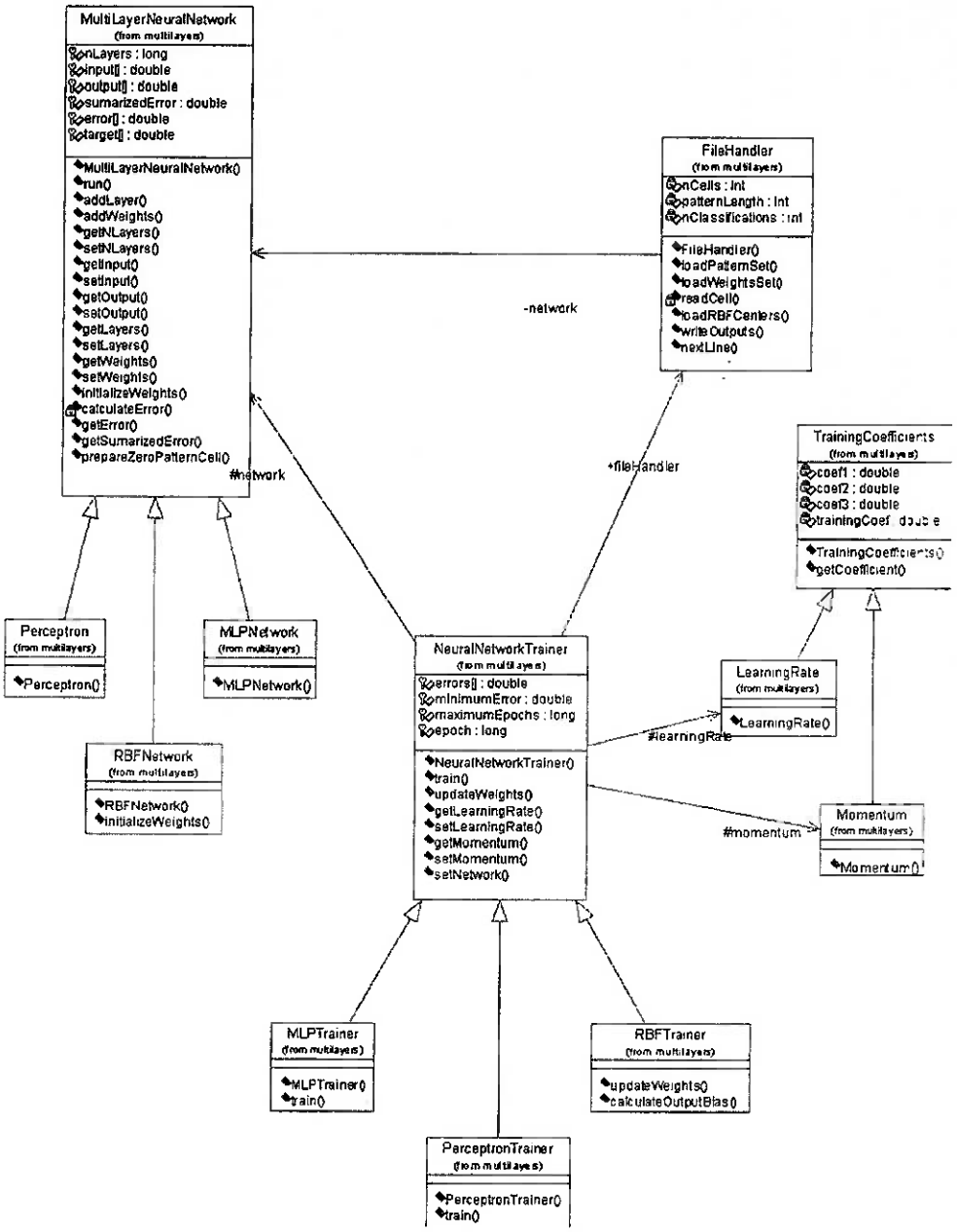


Diagrama de classes 2

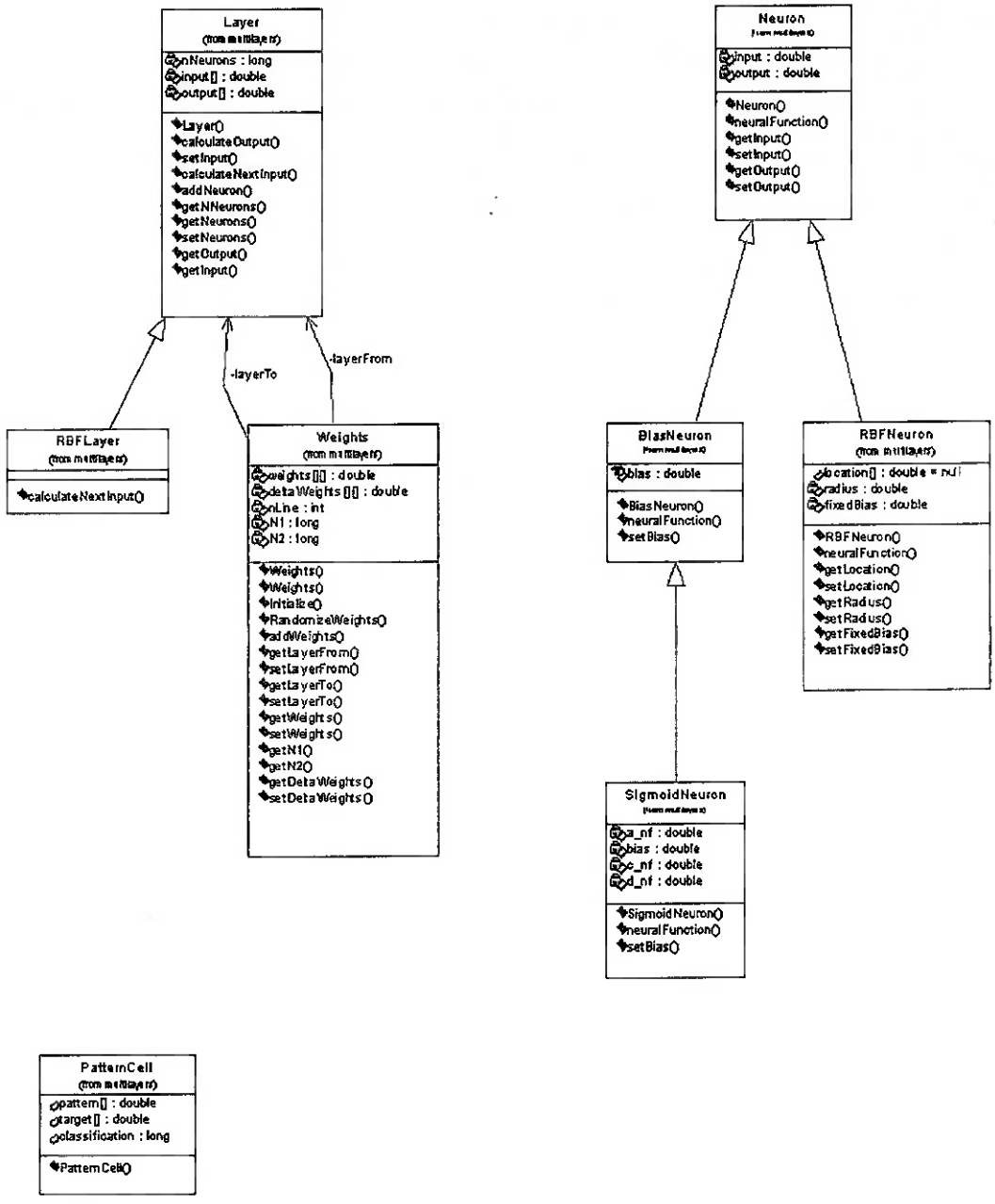


Diagrama de seqüência para a etapa *feed-forward*

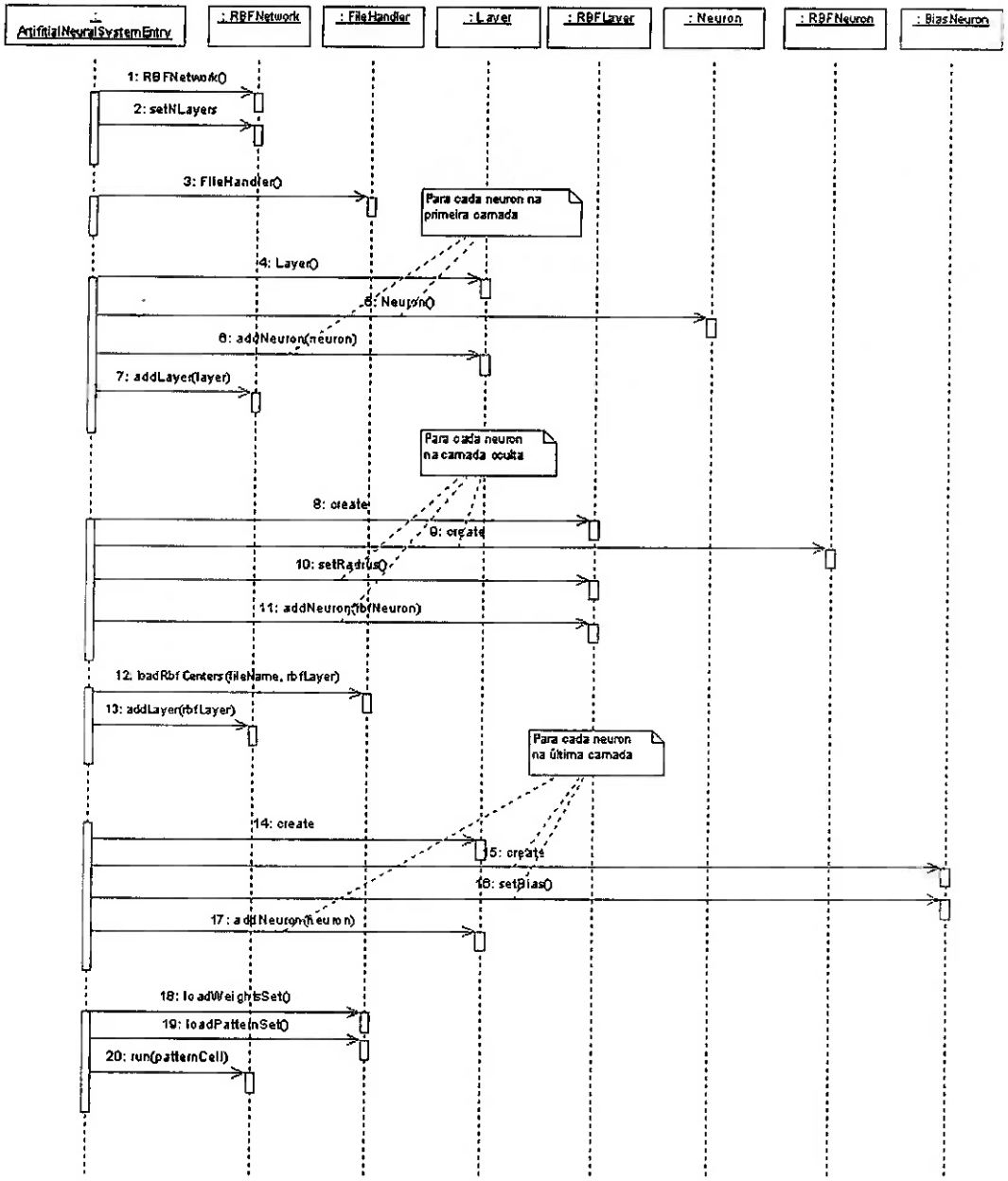
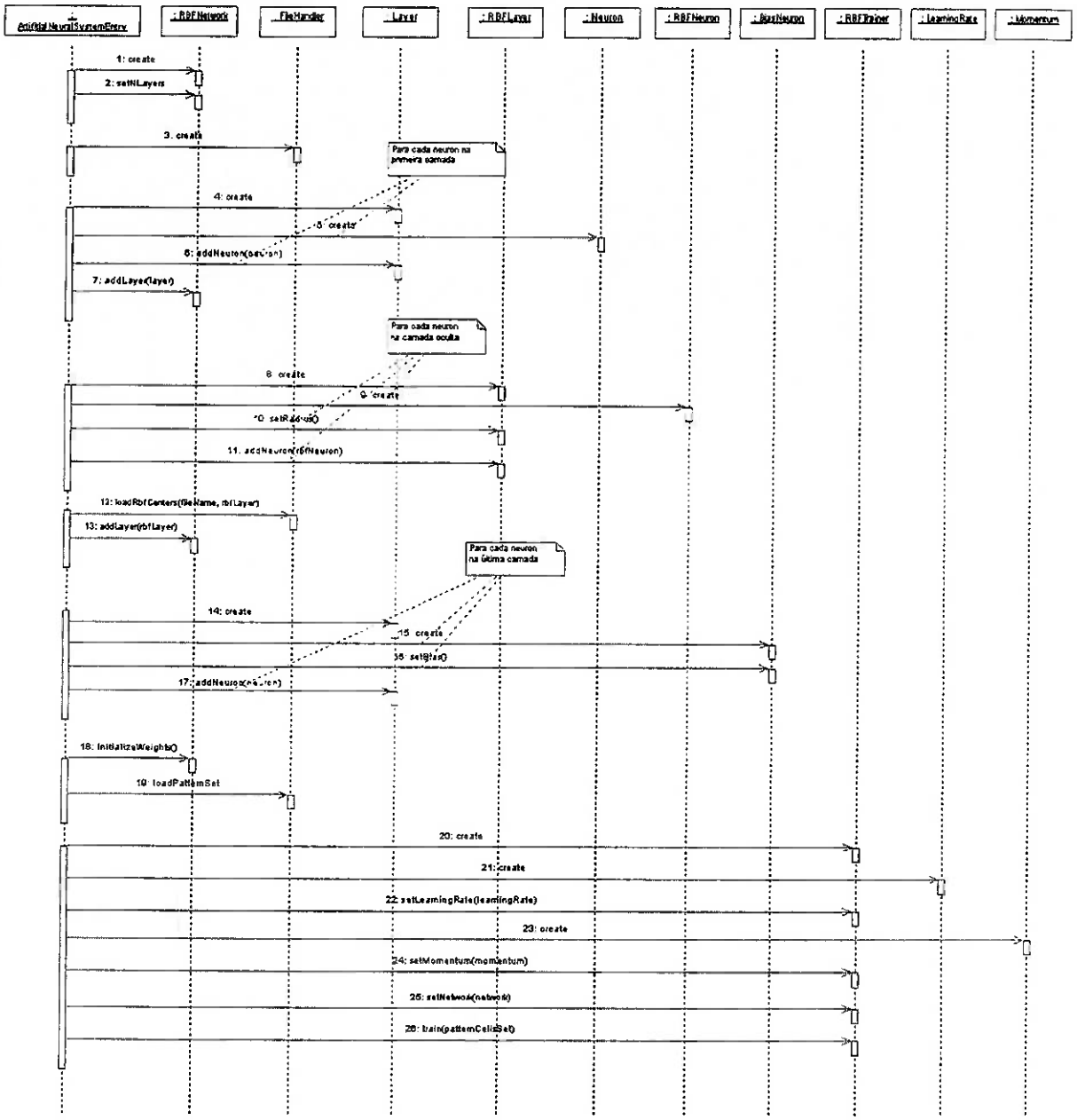


Diagrama de seqüência para a etapa de treinamento



Classe *MultiLayerNeuralNetwork*

```
package neuralnetworks.multilayers;

import java.util.Vector;
import java.util.Enumeration;

public class MultiLayerNeuralNetwork
{
   protected long nLayers;
   protected double[] input;
   protected double[] output;
   protected Vector layers;
   protected Vector weights;
   protected double sumarizedError;
   protected double[] error;
   protected double[] target;

   public MultiLayerNeuralNetwork()
   {
        nLayers                = 0;
        sumarizedError   = 10e10;
        layers                 = new Vector();
        weights                = new Vector();
   }

   /**
   @roseuid 38A5EF9202DE
   */
   public void run(PatternCell pc)
   {
        long count = 0;
        double[] internalInput = null;
        double[] internalOutput = null;
        Layer layer = null;
        Layer nextLayer = null;
        Weights theWeights = null;

        if(nLayers != layers.size())
             return;

        input = pc.pattern;
        target = pc.target;

          Enumeration enumLayers = layers.elements();
          Enumeration enumWeights = weights.elements();
          // First Layer
          if(enumLayers.hasMoreElements()){
               layer = (Layer) enumLayers.nextElement();
               layer.setInput(input);
               internalOutput = layer.calculateOutput();
               theWeights = (Weights)
enumWeights.nextElement();
               nextLayer = (Layer)
enumLayers.nextElement();
               count++;
```

```
                internalInput =
nextLayer.calculateNextInput(internalOutput, theWeights);
                layer = nextLayer;
           }
           // Hidden Layers
           while(enumLayers.hasMoreElements()){
                if(count != nLayers-1){
                     layer.setInput(internalInput);
                     internalOutput =
layer.calculateOutput();
                     theWeights = (Weights)
enumWeights.nextElement();
                     nextLayer = (Layer)
enumLayers.nextElement();
                     count++;
                     internalInput =
nextLayer.calculateNextInput(internalOutput, theWeights);
                     layer = nextLayer;
                }
           }
           // Last Layer
           if( count == nLayers -1) {
                layer.setInput(internalInput);
                output = layer.calculateOutput();
                sumarizedError = calculateError();
           }
           else{
                System.out.println("Erro na contagem das
camadas");
                return;
           }
     }

     /**
     @roseuid 38A619D002BC
     */
     public void addLayer(Layer layer)
     {
          layers.addElement(layer);
     }

     /**
     @roseuid 38A61A0A0140
     */
     public void addWeights(Weights weight)
     {
          weights.addElement(weight);
     }

     /**
* Access method for the nLayers property.
*
* @return   the current value of the nLayers property
     */
     public long getNLayers() {
        return nLayers;}
```

```
    /**
 * Sets the value of the nLayers property.
 *
 * @param aNLayers the new value of the nLayers property
    */
    public void setNLayers(long aNLayers) {
       nLayers = aNLayers;}

    /**
 * Access method for the input property.
 *
 * @return   the current value of the input property
    */
    public double[] getInput() {
       return input;}

    /**
 * Sets the value of the input property.
 *
 * @param aInput the new value of the input property
    */
    public void setInput(double[] aInput) {
       input = aInput;}

    /**
 * Access method for the output property.
 *
 * @return   the current value of the output property
    */
    public double[] getOutput() {
       return output;}

    /**
 * Sets the value of the output property.
 *
 * @param aOutput the new value of the output property
    */
    public void setOutput(double[] aOutput) {
       output = aOutput;}

    /**
 * Access method for the layers property.
 *
 * @return   the current value of the layers property
    */
    public Vector getLayers() {
       return layers;}

    /**
 * Sets the value of the layers property.
 *
 * @param aLayers the new value of the layers property
    */
    public void setLayers(Vector aLayers) {
       layers = aLayers;}

    /**
```

```
* Access method for the weights property.
*
* @return   the current value of the weights property
   */
   public Vector getWeights() {
      return weights;}

   /**
* Sets the value of the weights property.
*
* @param aWeights the new value of the weights property
   */
   public void setWeights(Vector aWeights) {
      weights = aWeights;}


   public void initializeWeights(){
        Layer theLayer, nextLayer;
        long count = 0;
        Enumeration enum = layers.elements();
        theLayer = (Layer) enum.nextElement();
        while(enum.hasMoreElements()){
              nextLayer = (Layer) enum.nextElement();
               Weights w = new
Weights(theLayer.getNNeurons(), nextLayer.getNNeurons());
               w.Initialize(0.0);
               count++;
               w.setLayerFrom(theLayer);
               w.setLayerTo(nextLayer);
               weights.addElement(w);
               theLayer = nextLayer;
        }

   }

   private double calculateError(){
        double err = 0;
        error = new double[output.length];
        for(int i = 0; i < output.length; i++){
             error[i] = output[i] - target[i];
             err += error[i]*error[i];
        }
        return java.lang.Math.sqrt(err)/output.length;
   }

   public double[] getError(){
        return error;
   }

   public double getSumarizedError(){
        return sumarizedError;
   }

   public PatternCell prepareZeroPatternCell(){
        long nInput = 0;
        long nOutput = 0;
```

```
            for(int i = 0; i < nLayers; i++){
                Layer l = (Layer) layers.elementAt(i);
                if(i==0)
                    nInput = l.getNNeurons();
                else if(i == nLayers-1)
                    nOutput = l.getNNeurons();
                else
                    ;
            }
            return new PatternCell((int)nInput,(int) nOutput);
        }

    }
```

Classe *RBFNetwork*

```
package neuralnetworks.multilayers;

import java.util.Vector;
import java.util.Enumeration;

public class RBFNetwork extends MultiLayerNeuralNetwork
{

   public RBFNetwork()
   {
   }

   public void initializeWeights(){
        Layer theLayer, nextLayer;
        long count = 0;
        Enumeration enum = layers.elements();
        theLayer = (Layer) enum.nextElement();
        while(enum.hasMoreElements()){
              nextLayer = (Layer) enum.nextElement();
               Weights w = new
Weights(theLayer.getNNeurons(), nextLayer.getNNeurons());
               if(count == 0)
                    w.Initialize(1.0);
               else
                    w.RandomizeWeights();
               count++;
               w.setLayerFrom(theLayer);
               w.setLayerTo(nextLayer);
               weights.addElement(w);
               theLayer = nextLayer;
        }
  }
```

Classe *Neuron*

```
package neuralnetworks.multilayers;


public class Neuron
{
   private double input;
   private double output;

   public Neuron()
   {
   }

   /**
   @roseuid 38A5F0AF00E9
   */
   public double neuralFunction(double input)
   {
        return input;
   }

   /**
* Access method for the input property.
*
* @return   the current value of the input property
   */
   public double getInput() {
      return input;}

   /**
* Sets the value of the input property.
*
* @param aInput the new value of the input property
   */
   public void setInput(double aInput) {
      input = aInput;}

   /**
* Access method for the output property.
*
* @return   the current value of the output property
   */
   public double getOutput() {
      return output;}

   /**
* Sets the value of the output property.
*
* @param aOutput the new value of the output property
   */
   public void setOutput(double aOutput) {
      output = aOutput;}
}
```

Classe RBFNeuron

```
package neuralnetworks.multilayers;


public class RBFNeuron extends Neuron
{
   public double[] location = null;
   private double radius;
   private double fixedBias;

   public RBFNeuron(long locationLength)
   {
        fixedBias = 0.0;
        radius = 0.35;
        location = new double[(int)locationLength];
   }

   /**
   @roseuid 38A5F0FF0008
   */
   public double neuralFunction(double intput)
   {
        return ( java.lang.Math.exp(-(intput*intput) /
(radius*radius)));
   }

   /**
* Access method for the location property.
*
* @return   the current value of the location property
   */
   public double[] getLocation() {
      return location;}

   /**
* Sets the value of the location property.
*
* @param aLocation the new value of the location property
   */
   public void setLocation(double[] aLocation) {
       location = aLocation;
   }

   /**
* Access method for the radius property.
*
* @return   the current value of the radius property
   */
   public double getRadius() {
      return radius;}

   /**
* Sets the value of the radius property.
*
```

```
* @param aRadius the new value of the radius property
   */
   public void setRadius(double aRadius) {
      radius = aRadius;}

   public double getFixedBias(){
        return fixedBias;
   }

   public void setFixedBias(double aFixedBias){
        fixedBias = aFixedBias;
   }

}
```

Classe *BiasNeuron*

```
package neuralnetworks.multilayers;

public class BiasNeuron extends Neuron
{

   double bias;

   public BiasNeuron()
   {
        bias = 0.0;
   }

   /**
   @roseuid 38A5F0AF00E9
   */
   public double neuralFunction(double input)
   {
        return input + bias;
   }

   public void setBias(double aBias){
        bias = aBias;
   }

}
```

Classe *SigmoidNeuron*

```
package neuralnetworks.multilayers;


public class SigmoidNeuron extends Neuron
{
   private double a, bias, c, d;

   public SigmoidNeuron()
   {
          a = 1.0;
          bias = 0.0;
          c = 1.5;
          d = -2.5;
   }

   /**
   @roseuid 38A5F13302A1
   */
   public double neuralFunction(double intput)
   {
        double out = (a / (1.0 + java.lang.Math.exp( -(c *
intput)+d ))) - bias;
       return super.neuralFunction(out);
   }

   public void setBias(double aBias){
        bias = aBias;
   }
}
```

Classe *Layer*

```
package neuralnetworks.multilayers;

import java.util.Vector;
import java.util.Enumeration;

public class Layer
{
   private long nNeurons;
   private Vector neurons;
   private double[] input;
   private double[] output;

   public Layer()
   {
        nNeurons = 0;
        neurons = new Vector();
   }

   public double[] calculateOutput(){
        Neuron neuron;
        int    count = 0;

        if(input.length != nNeurons)
             return null;

        Enumeration enumNeurons = neurons.elements();
        while(enumNeurons.hasMoreElements()){
             neuron = (Neuron) enumNeurons.nextElement();
             output[count] =
neuron.neuralFunction(input[count]);
             count++;
        }
        return output;
   }

   public void setInput(double[] anInput){
        input = new double[anInput.length];
        input = anInput;
        output = new double[input.length];
   }

   public double[] calculateNextInput(double[] out, Weights
theWeights){
          double [] nextInput = null;
          double [][] w ;

          w = theWeights.getWeights();
          nextInput = new double[w[0].length];
          for(int j = 0; j < w[0].length; j++){
               for(int i = 0; i < w.length; i++)
                    nextInput[j] += out[i]*w[i][j];
          }
          return nextInput;
```

```
    }

    /**
    @roseuid 38A61A5A01FE
    */
    public void addNeuron(Neuron neuron)
    {
         neurons.addElement(neuron);
         nNeurons++;
    }

    /**
* Access method for the nNeurons property.
*
* @return   the current value of the nNeurons property
    */
    public long getNNeurons() {
       return nNeurons;}


    /**
* Access method for the neurons property.
*
* @return   the current value of the neurons property
    */
    public Vector getNeurons() {
       return neurons;}

    /**
* Sets the value of the neurons property.
*
* @param aNeurons the new value of the neurons property
    */
    public void setNeurons(Vector aNeurons) {
       neurons = aNeurons;}

    public double[] getOutput(){
         return output;
    }

    public double[] getInput(){
         return input;
    }
}
```

Classe *RBFLayer*

```
package neuralnetworks.multilayers;

import java.util.Vector;

public class RBFLayer extends Layer
{

     public double[] calculateNextInput(double[] out,
Weights theWeights){
      double [] nextInput = null;
      double [] location = null;
      double [][] w = null;
      RBFNeuron neuron = null;

      Vector neurons = this.getNeurons();
      w = theWeights.getWeights();
      nextInput = new double[w[0].length];

      if(neurons.size() != w[0].length){
           System.out.println("Número de neurons
incompatível com o de pesos!");
           return null;
      }

      for(int j = 0; j < w[0].length; j++){
          neuron = (RBFNeuron) (neurons.elementAt(j));
          location = neuron.getLocation();
          for(int i = 0; i < w.length; i++)
               nextInput[j] += (out[i]*w[i][j] -
location[i]) * (out[i]*w[i][j] - location[i]);
          nextInput[j] = java.lang.Math.sqrt(nextInput[j])
+ neuron.getFixedBias();
      }
      return nextInput;
   }

}
```

Classe *TrainingCoefficients*

```
package neuralnetworks.multilayers;

public class TrainingCoefficients
{
     private double coef1;
     private double coef2;
     private double coef3;
     private double trainingCoef;

     public TrainingCoefficients(double c1, double c2,
double c3){
          coef1 = c1;
          coef2 = c2;
          coef3 = c3;
     }

     public double getCoefficient(double time){
          trainingCoef = coef1 *
                         java.lang.Math.exp(-coef2*time)
*
                         java.lang.Math.exp(-
coef3*java.lang.Math.log(time));
          return trainingCoef;
     }
}
```

Classe *LearningRate*

```
package neuralnetworks.multilayers;

public class LearningRate extends TrainingCoefficients
{
     public LearningRate(double c1, double c2, double c3){
          super(c1, c2, c3);
     }
}
```

Classe *Momentum*

```
package neuralnetworks.multilayers;

public class Momentum extends TrainingCoefficients
{
     public Momentum(double c1, double c2, double c3){
          super(c1, c2, c3);
     }
}
```

Classe *Weights*

```
package neuralnetworks.multilayers;

import java.util.Vector;

public class Weights
{
   private Layer layerFrom;
   private Layer layerTo;
   private double[][] weights;
   private double[][] deltaWeights;
   private int nLine;
   private long N1, N2;

   public Weights()
   {
        layerFrom = null;
        layerTo = null;
        nLine = 0;
   }

   public Weights(long n1, long n2)
   {
        layerFrom = null;
        layerTo = null;
        nLine = 0;
        weights = new double[(int)n1][(int)n2];
        deltaWeights = new double[(int)n1][(int)n2];
        N1 = n1;
        N2 = n2;
   }

   public void Initialize(double value){
        for(int i = 0; i < N1; i++)
             for(int j = 0; j < N2; j++){
                  weights[i][j] = value;
                  deltaWeights[i][j] = 0.0;
             }
   }

   public void RandomizeWeights(){
        for(int i = 0; i < N1; i++)
             for(int j = 0; j < N2; j++){
                  weights[i][j] = java.lang.Math.random()
/ 10;
                  deltaWeights[i][j] = 0.0;
             }
   }

   /**
   @roseuid 38A61A32035C
   */
   public void addWeights(double[] aLineOfWeights)
   {
        weights[nLine] = aLineOfWeights;
```

```
            nLine++;
        }

        /**
     * Access method for the layerFrom property.
     *
     * @return   the current value of the layerFrom property
        */
        public Layer getLayerFrom() {
            return layerFrom;}

        /**
     * Sets the value of the layerFrom property.
     *
     * @param aLayerFrom the new value of the layerFrom property
        */
        public void setLayerFrom(Layer aLayerFrom) {
            layerFrom = aLayerFrom;}

        /**
     * Access method for the layerTo property.
     *
     * @return   the current value of the layerTo property
        */
        public Layer getLayerTo() {
            return layerTo;}

        /**
     * Sets the value of the layerTo property.
     *
     * @param aLayerTo the new value of the layerTo property
        */
        public void setLayerTo(Layer aLayerTo) {
            layerTo = aLayerTo;}

        /**
     * Access method for the weights property.
     *
     * @return   the current value of the weights property
        */
        public double[][] getWeights() {
            return weights;}

        /**
     * Sets the value of the weights property.
     *
     * @param aWeights the new value of the weights property
        */
        public void setWeights(double[][] aMatrixOfWeights) {
            weights = aMatrixOfWeights;}

        public long getN1() {
            return N1;}

        public long getN2() {
            return N2;}
```

```
    public double[][] getDeltaWeights() {
        return deltaWeights;}

    public void setDeltaWeights(double[][] aMatrixOfWeights)
{
        deltaWeights = aMatrixOfWeights;}

}
```

Classe *PatternCell*

```
package neuralnetworks.multilayers;

public class PatternCell
{
     public PatternCell(int patternLength, int
nClassifications){
          pattern = new double[patternLength];
          target = new double[nClassifications];
          classification = -1;
     }

     public double[] pattern;
     public double[] target;
     public long classification;
}
```

Classe *NeuralNetworkTrainer*

```
package neuralnetworks.multilayers;

import java.util.Vector;
import java.util.Enumeration;

public class NeuralNetworkTrainer
{
   protected double[] errors;
   protected double minimumError;
   protected long maximumEpochs;
   protected long epoch;
   protected Vector patternCellsSet;
   protected MultiLayerNeuralNetwork network;
   protected LearningRate learningRate;
   protected Momentum momentum;

   protected FileHandler fileHandler;
   protected String outputFileName;
   protected String weightsFileName;


   public NeuralNetworkTrainer()
   {
        minimumError = 0.001;
        maximumEpochs = 1000;
        epoch = 1;
   }

   /**
   @roseuid 38A6175501EA
   */
   public void train(Vector aPatternCellsSet)
   {
         patternCellsSet = aPatternCellsSet;
          long filePointer = 0;
          while(network.getSumarizedError() > minimumError
&& epoch < maximumEpochs){
               Enumeration enum =
patternCellsSet.elements();
               System.out.println("Epoch: "+epoch +" Error:
"+ network.getSumarizedError());
               while(enum.hasMoreElements()){
                    PatternCell pc =
(PatternCell)enum.nextElement();
                    network.run(pc);
                    try{
                         filePointer =
fileHandler.writeOutputs(outputFileName, filePointer);
                    }
                    catch(Exception e){
                         e.printStackTrace();
                    }
                    updateWeights();
               }
```

```
                epoch++;
                try{
                     filePointer =
fileHandler.nextLine(outputFileName, filePointer);
                }
                catch(Exception e){
                     e.printStackTrace();
                }
           }
           try{
                fileHandler.writeWeights(weightsFileName);
           }catch(Exception e){
                e.printStackTrace();
           }
   }


   // Each extended class has its own algorithm for weights
update.
   public void updateWeights(){
   }



   /**
* Access method for the learningRate property.
*
* @return   the current value of the learningRate property
   */
   public LearningRate getLearningRate() {
      return learningRate;}

   /**
* Sets the value of the learningRate property.
*
* @param aLearningRate the new value of the learningRate
property
   */
   public void setLearningRate(LearningRate aLearningRate)
{
      learningRate = aLearningRate;}

   public Momentum getMomentum() {
      return momentum;}

   public void setMomentum(Momentum aMomentum) {
      momentum = aMomentum;}

   public void setNetwork(MultiLayerNeuralNetwork
aNetwork){
        network = aNetwork;
   }

   public void setFileHandler(FileHandler fh){
        fileHandler = fh;
   }
   public void setOutputFileName(String fileName){
```

```
            outputFileName = fileName;
        }
        public void setWeightsFileName(String fileName){
            weightsFileName = fileName;
        }

    }
```

Classe *RBFTrainer*

```
package neuralnetworks.multilayers;

import java.util.Vector;
import java.util.Enumeration;

public class RBFTrainer extends NeuralNetworkTrainer
{

   public void updateWeights(){
        Enumeration enum =
this.network.getWeights().elements();
        long count = 0;
        Weights w;
        double[][] weightsNew;
        double[][] deltaWeightsNew;
        while(enum.hasMoreElements()){
             w = (Weights) enum.nextElement();
             if(count > 0){
                  weightsNew       = new
double[(int)w.getN1()][(int)w.getN2()];
                  deltaWeightsNew  = new
double[(int)w.getN1()][(int)w.getN2()];
                  for(int i = 0; i < w.getN2(); i++){
                       for(int j = 0; j < w.getN1(); j++){
                            double aux = (
w.getLayerFrom().getOutput()[j] *

this.network.getError()[i] *

this.getLearningRate().getCoefficient(epoch) ) +
                                                    (
w.getDeltaWeights()[j][i] *

this.getMomentum().getCoefficient(epoch) );
                            deltaWeightsNew[j][i] = aux;
                            weightsNew[j][i] =
w.getWeights()[j][i] - deltaWeightsNew[j][i];
                       }
                  }
                  w.setWeights(weightsNew);
                  w.setDeltaWeights(deltaWeightsNew);
             }
             count++;
        }
   }

   public void calculateOutputBias(){
        BiasNeuron neuron;
        PatternCell pc = network.prepareZeroPatternCell();
        network.run(pc);
        Layer lastLayer = (Layer)
network.getLayers().elementAt((int) network.getNLayers()-
1);
```

```
            for(int i = 0; i < lastLayer.getNNeurons(); i++){
                neuron = (BiasNeuron)
lastLayer.getNeurons().elementAt(i);
                neuron.setBias(network.getOutput()[i]);
            }
        }

}
```

Classe *FileHandler*

```
package neuralnetworks.multilayers;

import java.util.Vector;
import java.util.Enumeration;
import java.util.StringTokenizer;
import java.text.DecimalFormat;
import java.io.*;

public class FileHandler
{
     private Vector weightsSet;
     private int    nCells;
     private int    patternLength;
     private int    nClassifications;
     private MultiLayerNeuralNetwork network;

     public FileHandler(MultiLayerNeuralNetwork aNetwork){
          network = aNetwork;
     }

     public Vector loadPatternSet(String fileName) throws
IOException {
          RandomAccessFile    file;
          Vector trainSet = new Vector();

          file = new RandomAccessFile(fileName, "r");

          String s = file.readLine().trim();

          StringTokenizer st = new StringTokenizer(s," ");
          nCells = new
Integer((String)st.nextElement()).intValue();
          patternLength =  new
Integer((String)st.nextElement()).intValue();
          nClassifications = new
Integer((String)st.nextElement()).intValue();

          Layer layer = (Layer)
network.getLayers().firstElement();
          if(patternLength != layer.getNNeurons()){
               System.out.println("O tamanho dos Patterns
não bate com a estrutura da rede!");
               return null;
          }

          for(int i = 0; i < nCells; i++){
               trainSet.addElement(readCell(file));
          }
          file.close();
          return trainSet;
     }

     public void loadWeightsSet(String fileName) throws
IOException {
```

```
            RandomAccessFile    file;
            Weights w = null;
            double[][] newW = null;
            StringTokenizer st;
            Vector networkWeights = new Vector();

            file = new RandomAccessFile(fileName, "r");

            Enumeration enum =
network.getWeights().elements();

            while(enum.hasMoreElements()){
                  w = (Weights) enum.nextElement();
                  newW = new
double[(int)w.getN1()][(int)w.getN2()];
                  for(int i = 0; i < w.getN1(); i++){
                        String s = file.readLine().trim();
                        st = new StringTokenizer(s," ");
                        for(int j = 0; j < w.getN2(); j++)
                              newW[i][j] = new Double ((String)
st.nextElement()).doubleValue();
                  }
                  w.setWeights(newW);
                  networkWeights.addElement(w);
            }
            network.setWeights(networkWeights);
            file.close();
      }


      private PatternCell readCell(RandomAccessFile file)
throws IOException {
            String s;
            StringTokenizer st;
            PatternCell trainCell = new
PatternCell(patternLength,nClassifications);

            s = file.readLine().trim();
            st = new StringTokenizer(s," ");
            for(int i = 0; i < patternLength; i++){
                  trainCell.pattern[i] = new
Double((String)st.nextElement()).doubleValue();
            }

            s = file.readLine().trim();
            st = new StringTokenizer(s," ");
            for(int i = 0; i < nClassifications; i++){
                  trainCell.target[i] = new
Double((String)st.nextElement()).doubleValue();
            }
            trainCell.classification = new
Integer((String)st.nextElement()).intValue();
            return trainCell;
      }

      public void loadRBFCenters(String fileName, RBFLayer
layer) throws IOException {
```

```
            RandomAccessFile    file;
            RBFNeuron neuron;

            file = new RandomAccessFile(fileName, "r");

            String s = file.readLine().trim();

            StringTokenizer st = new StringTokenizer(s," ");
            int nNeurons = new
Integer((String)st.nextElement()).intValue();
            int centerLength =  new
Integer((String)st.nextElement()).intValue();

            if(nNeurons != layer.getNNeurons()){
                System.out.println("O tamanho dos Centros
não bate com a estrutura da rede!");
                return;
            }

            Vector neurons = layer.getNeurons();
            for(int i = 0; i < nNeurons; i++){
                neuron = (RBFNeuron) neurons.elementAt(i);
                s = file.readLine().trim();
                st = new StringTokenizer(s," ");
                for(int j = 0; j < centerLength; j++)
                    neuron.location[j] = new
Double((String)st.nextElement()).doubleValue();
            }
            file.close();
        }

        public long writeOutputs(String fileName, long
filePonter) throws IOException {

            RandomAccessFile    file;
            DecimalFormat format = new
DecimalFormat("###.#####");

            File f = new File(fileName);
            f.delete();

            file = new RandomAccessFile(fileName, "rw");
            file.seek(filePonter);

            double [] out = network.getOutput();
            for(int i = 0; i < out.length; i++)
                file.writeBytes(out[i]+" ");
                //file.writeBytes(format.format(out[i])+"
");
            file.writeBytes("   Error:
"+network.getSumarizedError());
            file.write(0x000d);
            file.write(0x000a);
            return file.getFilePointer();
            //file.close();
        }
```

```
    public long nextLine(String fileName, long filePonter)
throws IOException {
          RandomAccessFile    file;
          file = new RandomAccessFile(fileName, "rw");
          file.seek(filePonter);
          file.write(0x000d);
          file.write(0x000a);
          file.write(0x000d);
          file.write(0x000a);
          return file.getFilePointer();
    }

    public void writeWeights(String fileName) throws
IOException {

          RandomAccessFile    file;
          Weights w = null;

          File f = new File(fileName);
          f.delete();

          file = new RandomAccessFile(fileName, "rw");

          Enumeration enum =
network.getWeights().elements();

          while(enum.hasMoreElements()){
               w = (Weights) enum.nextElement();
               for(int i = 0; i < w.getN1(); i++){
                    for(int j = 0; j < w.getN2(); j++)

    file.writeBytes(w.getWeights()[i][j]+" ");
                    file.write(0x000d);
                    file.write(0x000a);
               }
          }
          file.close();
    }

}
```

Classe *ArtifitialNeuralSystemEntry*

```
package neuralnetworks.multilayers;

import java.util.Vector;
import java.util.Enumeration;

public class ArtifitialNeuralSystemEntry
{
     private final static long RUN   = 1;
     private final static long TRAIN = 2;

     private final static long NLAYERS = 3;

     public static void main(String argv[]){

          // Neuron types
          Neuron neuron;
          RBFNeuron rbfneuron;
          BiasNeuron biasNeuron;
          SigmoidNeuron sigNeuron;

          long action;

          if(argv[0].equals("TRAIN"))
               action = TRAIN;
          else if(argv[0].equals("RUN"))
               action = RUN;
          else {
               System.out.println(" Tem que ser TRAIN ou
RUN!");
               return;
          }
          long firstLayerNeuronsQty =
Long.parseLong(argv[6]);
          long hiddenLayerNeuronsQty =
Long.parseLong(argv[7]);
          long lastLayerNeuronsQty =
Long.parseLong(argv[8]);
          double bias = new Double(argv[9]).doubleValue();
          double lRate = new
Double(argv[10]).doubleValue();

          MultiLayerNeuralNetwork network = new
RBFNetwork();
          network.setNLayers(NLAYERS);

          FileHandler fh = new FileHandler(network);

          // First Layer - Identity neuron
          Layer layer1 = new Layer();
          for(int j = 0; j < firstLayerNeuronsQty; j++){
               neuron = new Neuron();
               layer1.addNeuron(neuron);
          }
          network.addLayer(layer1);
```

```
            // Second Layer - RBF neuron
            RBFLayer rbfLayer = new RBFLayer();
            for(int j = 0; j < hiddenLayerNeuronsQty; j++){
                rbfneuron = new
RBFNeuron(layer1.getNNeurons());
                rbfneuron.setRadius(0.25);
                rbfLayer.addNeuron(rbfneuron);
            }
            try{
                fh.loadRBFCenters(argv[2], rbfLayer);
            }
            catch(Exception e){
                e.printStackTrace();
            }
            network.addLayer(rbfLayer);

            // Third Layer - Sigmoid neuron or Bias neuron
            Layer layer3 = new Layer();
            for(int j = 0; j < lastLayerNeuronsQty; j++){
                //sigNeuron = new SigmoidNeuron();
                //sigNeuron.setBias(bias);
                //layer3.addNeuron(sigNeuron);
                biasNeuron = new BiasNeuron();
                biasNeuron.setBias(bias);
                layer3.addNeuron(biasNeuron);
            }
            network.addLayer(layer3);
            network.initializeWeights();


            // Train? Run?
            try{
                fh.loadRBFCenters(argv[2],(RBFLayer)
network.getLayers().elementAt(1));
                Vector patternCellsSet =
fh.loadPatternSet(argv[1]);
                if(action == RUN){
                    fh.loadWeightsSet(argv[5]);
                    Enumeration enum =
patternCellsSet.elements();
                    long filePointer = 0;
                    while(enum.hasMoreElements()){
                        PatternCell pc =
(PatternCell)enum.nextElement();
                        network.run(pc);
                        filePointer =
fh.writeOutputs(argv[3], filePointer);
                    }
                }
                else if(action == TRAIN){
                    // Setting LearningRate and Momentum
for training
                    RBFTrainer networkTrainer = new
RBFTrainer();
                    LearningRate learningRate = new
LearningRate(lRate, 0.0004, 0.4);
```

```
                    Momentum momentum = new Momentum(0.96,
0.0, 0.0 );

     networkTrainer.setLearningRate(learningRate);
                    networkTrainer.setMomentum(momentum);
                    networkTrainer.setNetwork(network);
                    //networkTrainer.calculateOutputBias();
                    networkTrainer.setFileHandler(fh);

     networkTrainer.setOutputFileName(argv[3]);

     networkTrainer.setWeightsFileName(argv[4]);
                    networkTrainer.train(patternCellsSet);
               }
               else
                    return;
          }
          catch(Exception e){
               e.printStackTrace();
          }
     }

}
```