Universidade Federal do Rio Grande do Norte
Programa de Pós-Graduação em Engenharia Elétrica e de Computação
Redes Neurais (EEC1505)
Prof. Adrião Duarte Doria Neto
Alunos: José Lenival Gomes de França, Raphael Diego Comesanha e Silva, Danilo de Santana Pena.

Lista 3 Exercícios

1. A representação de uma determinada mensagem digital ternária, isto é formada por três bits, forma um cubo cujos vértices correspondem a mesma representação digital. Supondo que ao transmitirmos esta mensagem a mesma seja contaminada por ruído formando em torno de cada vértice uma nuvem esférica de valores aleatórios. O raio da esfera corresponde ao desvio padrão do sinal de ruído. Solucione o problema usando máquinas de vetor de suporte linear. Compare com a solução obtida na lista 2 onde foi usada uma rede de perceptron de Rosemblat com uma camada para atuar como classificador/decodificador. Para solução do problema defina antes um conjunto de treinamento e um conjunto de validação.

2. Implemente a RBF considerando os algoritmos de treinamento para as três situações: (a) centros fixos e escolhidos aleatoriamente, (b) centros escolhidos através da seleção auto-supervisionada (algoritmo K-means) , (c) centros escolhidos através da seleção supervisionada, para as três questões abaixo:

   a) A função lógica $f(x_1, x_2, x_3) = x_1 \oplus x_2 \oplus x_3$

   b)
   $$f(x) = \left[\frac{sin(\pi||x||)}{\pi||x||}\right], \; x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \; |x_1| \leq 10 \text{ e } |x_2| \leq 10$$

   c) $f(x) = x_1^2 + x_2^2 - 2x_1x_2 + x_1 + x_2 - 1$, $|x_1| \leq 10$, $|x_2| \leq 10$

   RESOLUÇÃO:

   As alternativas *a*) e *c*) possuiram erro igual a 0. Na alternativa *b*) não conseguiu-se bons resultados utilizando a *toolbox*, pois a saída da rede resulta em valor NaN.

   Segue o código utilizando a *toolbox* do MATLAB:

```
%% Questao 2
%% a)
clear

% Gerando dados
x1 = [0 1 0 1 0 1 0 1];
x2 = [0 0 1 1 0 0 1 1];
x3 = [0 0 0 0 1 1 1 1];
x = [x1; x2; x3];

f = xor(xor(x1,x2),x3);
padroes = [x; f]';
padroes = repmat(padroes,5,1);
ni = size(padroes,1);
padroes = padroes(randperm(ni),:);

% Treinando rede
net = newrbe(padroes(:,1:3)',padroes(:,4)');
%view(net)
% Simulando rede
y = sim(net,padroes(:,1:3)')';

%% b)
clear

x1 = -10:0.1:10;
x2 = -10:0.1:10;
x = zeros(1,length(x1));
for i=1:201
    x(i) = norm([x1(:,i) x2(:,i)]);
end

f = sin(x.*pi)./(x.*pi);

padroes = [x; f]';

net = newrbe(padroes(:,1)',padroes(:,2)');
y = sim(net,padroes(:,1)')';

%% c)
clear

x1 = -10:0.1:10;
x2 = -10:0.1:10;
x = [x1; x2];

f = x1.^2 + x2.^2 - 2.*x1.*x2 + x1 + x2 - 1;

padroes = [x; f]';

net = newrbe(padroes(:,1:2)',padroes(:,3)');

y = sim(net,padroes(:,1:2)')';
```

3. Considere o problema de classificação de padrões constituído neste caso de 12 padrões. A distribuição dos padrões tem como base um quadrado centrado no ponto (0.5,0.5) e lados iguais a 1. Os pontos (0.5,0.5), (1.0,0.5), (0.5,1.) e (0.0, 0.5) são centros de quatro semicírculos que se interceptam no interior do quadrado originando quatro classes e outras oito classes nas regiões de não interseção. Após gerar aleatoriamente dados que venham formar estas distribuições de dados, selecione um conjunto de treinamento e um conjunto de validação. Solucione o problema usando RBF, SVM e Máquina de Comitê. Verifique o desempenho do classificador usando o conjunto de validação e calculando a matriz de confusão e compare com o obtido na lista anterior usando MLP.

   RESOLUÇÃO:

   Segue abaixo nossa implementação da RBF, com problema para convergir (apenas converge para erro igual zero após algumas tentativas).

```
classdef RBF < handle
    properties

        ni % Número de entradas
        nh % Numero de neuronios ocultos
        no % Numero de saidas
        n_classes % Número de classes

        % Vetores de entrada e ativação
        xi % Vetor de entradas
        yh % Saída na camada oculta
        yo % Saída da rede
        centro % Vetor com índices de cada classe retornado pelo kmeans

        % Vetores de erro e gradientes locais
        delta
        erro

        % Matrizes de centros e pesos sinápticos
        wo
        sigma2
        wc
    end

    methods (Access = private)
        function iniciar_centros(obj,met_de_ini,padroes)
            if met_de_ini == 's'
                figure;scatter(padroes(:,1),padroes(:,2))
                centros = ginput(obj.nh);
                [obj.centro,centros] =...
                kmeans(padroes(:,1:obj.ni),obj.nh,'start',obj.wc');
                close;
            else
                if isempty(obj.wc)
                    [obj.centro,centros] =...
                    kmeans(padroes(:,1:obj.ni),obj.nh);
                else
                    [obj.centro,centros] =...
                    kmeans(padroes(:,1:obj.ni),obj.nh,'start',obj.wc');
                end
            end
            obj.wc = centros';
        end
    end

    methods
        function self = RBF(ni, nh, no, n_classes)
            % Inicialização dos parâmetros da rede
            self.ni = ni;
            self.nh = nh;
            self.no = no;
            self.n_classes = n_classes;

            % Inicialização dos principais sinais
            self.xi = zeros(self.ni,1);
            self.yh = ones(self.nh+1,1);
```

```
    self.yo = zeros(self.no,1);
    self.delta = zeros(self.no,1);

    % Vetor que conterá o erro da última camada
    self.erro = zeros(self.no,1);

    % Inicialização de pesos sinápticos e mudança anterior
    % nos pesos para o momento
    self.wc = [];
    self.wo = 2*rand(nh+1,no)-1;
end

function saida = atualizar(obj,entradas)
    % Verificando a quantidade de entradas
    try
        obj.xi = entradas(:);
    catch e
        throw(e);
    end

    % Passando pelas exponenciais
    for c = 1:1:obj.nh
        obj.yh(c,1) = exp(-(obj.xi - obj.wc(:,c))'*...
        (obj.xi - obj.wc(:,c)))/(2*obj.sigma2(c));
    end

    % Passando pelo combinador linear
    obj.yo = (obj.yh'*obj.wo)';

    saida = obj.yo;
end

function erro_quad = ajustes_dos_pesos(obj, desejado, eta)
    % Verificando a quantidade de saidas
    try
        obj.erro = desejado - obj.yo;
    catch e
        throw(e);
    end

    % Calculando o delta na camada de saída
    obj.delta = obj.erro;

    % Fazendo correções nas sinapses da camada de saída
    obj.wo = obj.wo + eta*(obj.delta*obj.yh')';

    E = 0.5*(obj.erro'*obj.erro);

    erro_quad = E;
end

function treinar(obj, padroes, epocas, eta, met_de_ini)
    % padroes   --> Padrões a serem utilizados no treino
    % epocas    --> Número de épocas
    % eta       --> Taxa de aprendizado
```

```
            % met_de_ini--> Inicialização dos centros:
            %               'a' (aleatórios), 's' (supervisionado) e
            %               'as' (autosupervisionado)
            [n_pad, n_io] = size(padroes);
            J = zeros(epocas,1);

            %if (met_de_ini ~= 'a')
            obj.iniciar_centros(met_de_ini,padroes)
            %end

            'Valor de wo antes do treino'
            obj.wo
            % Calculo da variância
            for c=1:1:obj.nh
                obj.sigma2(c) =...
                mean(sum((padroes(obj.centro==c,1:obj.ni)'-
obj.wc(c)).^2));
            end

            for i = 1:1:epocas
                padroes = padroes(randperm(n_pad),:); % Misturando
entradas
                erro_quad = 0;
                for p = 1:1:n_pad
                    entrada = padroes(p,1:(obj.ni));
                    desejado = padroes(p,obj.ni+1:n_io);
                    atualizar(obj,entrada);
                    erro_quad =...
                    erro_quad + obj.ajustes_dos_pesos(desejado, eta);
                end
                J(i) = erro_quad;
            end
            figure;
            plot(1:1:epocas,J);

            'Valor de wo após o treino'
            obj.wo
        end

        function saida_funcao = testar(obj,padroes)
            n_pad = size(padroes,1);
            saida = zeros(n_pad,1);
            for p = 1:1:n_pad
                entrada = padroes(p,:);
                % Armazena apenas as saídas em saida
                saida(p,1) = obj.atualizar(entrada);
            end
            saida_funcao = saida;
        end

    end

end
```

4. Utilize uma rede NARX, no caso uma rede neural perceptron de múltiplas camadas com realimentação global, para fazer a predição de um passo, até predição de três passos da série temporal $x(n) = 1 + cos(n + cos2(n))$. Avalie o desempenho mostrando para cada caso os erros de predição.

   RESOLUÇÃO:

   Foi utilizado a *toolbox* do MATLAB, como segue nas figuras 1 e 2. O código segue abaixo:

```
% Rede NARX

% Geracao dos dados
x = {};
y = {};

n = 1:0.1:10;

xv = 1 + cos(n + cos(n).^2); % Entrada como vetor

for i=1:100
    n = i/10;
    x{1,i} = 1 + cos(n + cos(n).^2); % Entrada como celula
    y{1,i} = 1 + cos(n + cos(n).^2);
end

% Treinamento da rede
net = narxnet(10);
[xo,xi,~,to] = preparets(net,x,{},y);
net = train(net,xo,to,xi);
s = net(xo,xi);

sv = [];
for i=1:90
    sv(i) = s{i}; % saida como vetor
end

plot(xv,'b')
hold on
plot(sv,'r')
```

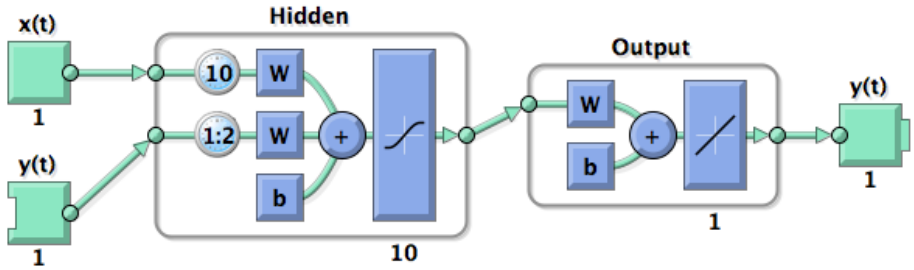

Figura 1: Arquitetura da rede NARX utiizada.

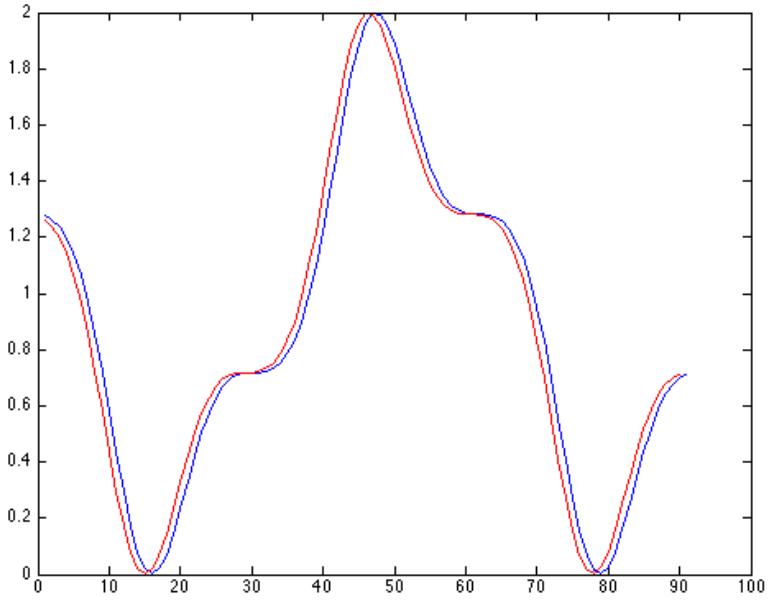

Figura 2: Resultado da rede NARX.

5. Implemente uma rede de Hopfield, para reconhecer as letras AFC. (Para cada letra forme uma matriz binária de pixel). Verifique o desempenho com as letras sendo apresentadas de forma ruidosa.

   RESOLUÇÃO:

   A implementação da rede Hopfield é feita como segue:

$$u_j(k) = \sum_{i=1}^{n} W_{ji} \cdot v_i(k-1) + i_j^b, \text{ onde } j = 1, \cdots, n$$

$$v_j(k) = g(u_j(k))$$

   onde k é um passo de iteração.

   Para garantir a estabilidade da rede, segue-se o algoritmo:

1) Especificar a matriz de pesos W e o vetor de limiares i^b;
2) Apresentar vetor inicial de entradas ($x^{(0)}$);
3) $v^{atual} \leftarrow x^{(0)}$;
4) Repetir as instruções:
   $v^{anterior} \leftarrow v^{atua}$;
   $u \leftarrow W \cdot v^{anterior} + i^b$;
   $v^{atual} \leftarrow g(u)$;
   Até que: $v^{atual} \cong v^{anterior}$
5) $v^{final} \leftarrow v^{atual}$ (v^{final} representa um ponto de equilíbrio)

6. Dado o modelo não linear de espaço de estado abaixo, obtenha o modelo de espaço de estados linearizado para ser utilizado no algoritmo EKF.

$$x(n+1) = f(n, x(n)) + v_1(n)$$

$$y(n) = c(n, x(n)) + v_2(n)$$

$$f(n, x(n)) = \begin{bmatrix} x_1(n) + x_2^2(n) \\ nx_1(n) - x_1(n)x_2(n) \end{bmatrix}$$

$$c(n, x(n)) = x_1(n)x_2^2(n) + v_2(n)$$

RESOLUÇÃO:

Foi implementado o EKF como segue abaixo, porém para validar o código foi utilizado uma outra função dada por:

$$f(x(n)) = \begin{bmatrix} x_2(n) \\ x_3(n) \\ 0.05 \cdot x_1(n) \cdot (x_2(n) + x_3(n)) \end{bmatrix}$$

Os resultados estão na figura 3.

```
%% Teste do EKF

n = 3;                  % Numero de estados
dp = 0.1;               % Desvio padrao do processo
r = 0.1;                % Desvio padrao da medida
Q = dp^2*eye(n);        % Covariancia do processo
R = r^2;                % Covariancia da medida
f = @(x)[x(2);x(3);0.05*x(1)*(x(2)+x(3))];  % Equacoes de estados nao lineares
h = @(x)x(1);                               % Equacoes de medidas
s = [0;0;1];                                % Estado inicial
x = s + dp*randn(3,1);                      % Estado inicial com ruido
P = eye(n);                                 % Estado inicial de covariancia
N = 20;                                     % Numero total de passos
xV = zeros(n,N);                            % Estado estimado
sV = zeros(n,N);                            % Estado atual
zV = zeros(1,N);
for k=1:N
  z = h(s) + r*randn;                       % Medidas
  sV(:,k)= s;                               % Estado atual armazenado
  zV(k)  = z;                               % Medida armazenada
  [x, P] = ekf(f,x,P,h,z,Q,R);              % EKF
  xV(:,k) = x;                              % Estimacao armazenada
  s = f(s) + dp*randn(3,1);                 % Processo atualizado
end
for k=1:3                                   % Plotando resultados
  subplot(3,1,k)
  plot(1:N, sV(k,:), '-', 1:N, xV(k,:), '--')
end
```

```
function [x,P] = ekf(fstate,x,P,hmeas,z,Q,R)

    [x1,A]=jaccsd(fstate,x);    %nonlinear update and linearization at current state
    P=A*P*A'+Q;                 %partial update
    [z1,H]=jaccsd(hmeas,x1);    %nonlinear measurement and linearization
    P12=P*H';                   %cross covariance
    % K=P12*inv(H*P12+R);       %Kalman filter gain
    % x=x1+K*(z-z1);            %state estimate
    % P=P-K*P12';               %state covariance matrix
    R=chol(H*P12+R);            %Cholesky factorization
    U=P12/R;                    %K=U/R'; Faster because of back substitution
    x=x1+U*(R'\(z-z1));         %Back substitution to get state update
    P=P-U*U';                   %Covariance update, U*U'=P12/R/R'*P12'=K*P12.

end

function [z,A]=jaccsd(fun,x)

    % JACCSD Jacobian through complex step differentiation
    % [z J] = jaccsd(f,x)
    % z = f(x)
    % J = f'(x)
    %
    z=fun(x);
    n=numel(x);
    m=numel(z);
    A=zeros(m,n);
    h=n*eps;
    for k=1:n
        x1=x;
        x1(k)=x1(k)+h*i;
        A(:,k)=imag(fun(x1))/h;
    end

end
```

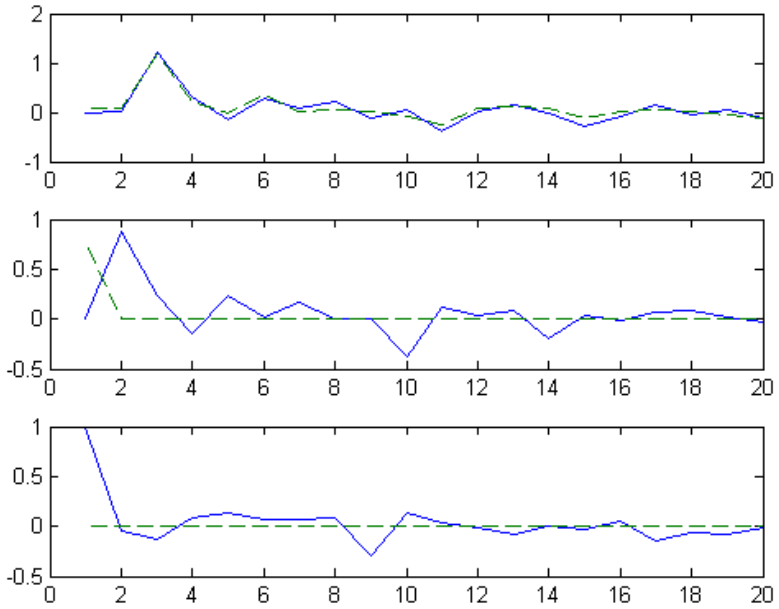

Figura 3: Resultado da EKF.

7. Um problema interessante para testar a capacidade de uma rede neural atuar como classificado de padrões é o problema das duas espirais intercaladas. Gere os exemplos de treinamento usando as seguintes equações:
para espiral 1 $x = \frac{\theta}{4}cos(\theta)$, $y = \frac{\theta}{4}sen(\theta)$, $\theta \geq 0$
para espiral 2 $x = (\frac{\theta}{4} + 0.8)cos(\theta)$, $y = (\frac{\theta}{4} + 0.8)sen(\theta)$, $\theta \geq 0$
fazendo θ assumir 51 igualmente espaçados valores entre 0 e 20 radianos. Utilize uma rede competitiva e em seguida uma rede SOM para atuar como classificador auto-supervisionado, isto é, a espiral 1 sendo uma classe e espiral 2 sendo outra classe. Para comparar as regiões de decisões formadas pela rede , gere uma grade uniforme com 100 x 100 exemplos de teste em um quadrado [-5,5]. Esboce os pontos classificados pela rede.

   RESOLUÇÃO:

   Foram implementados o algoritmo competitivo e o algoritmo SOM, como segue abaixo junto com a especificação:

   A execução do algoritmo competitivo para classificar os dados da expiral (classe 1) e da expiral 2 (classe 2) foi definida com as seguintes especificações:

   N de neurônios: 100
   N de épocas: 5000
   Taxa de aprendizagem: 0,1
   N de entradas: 2

   Os pesos (representados pelos círculos vermelhos) foram inicializados aleatoriamente em um espaço tridimensional normalizado, como mostrado na figura 4.

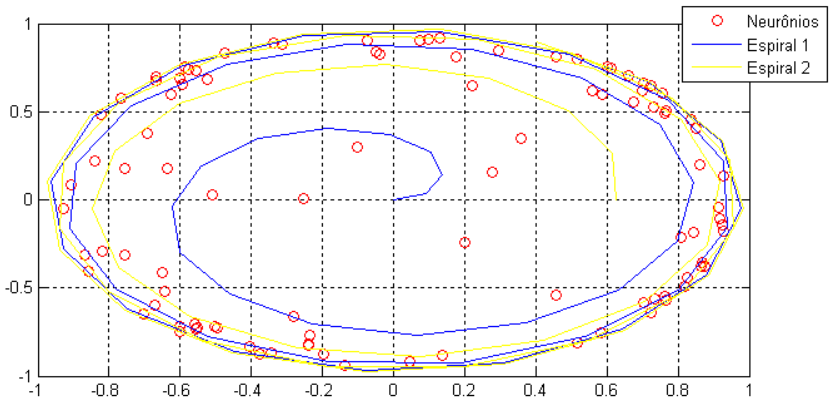

Figura 4: Resultado da rede SOM para a espiral.

Nessa figura é possível perceber que poucos pesos encontram-se alinhados as espirais. Após o treinamento do algoritmo competitivo, os pesos foram deslocados, de tal forma a alinharem-se às espirais com ocorrência de poucos pesos desalinhados, como pode ser visto na figura 5.

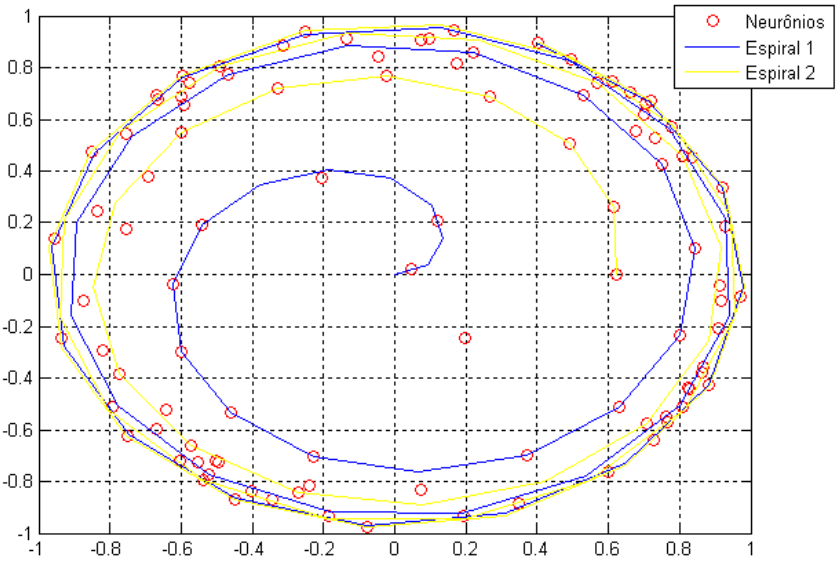

Figura 5: Resultado da rede SOM para a espiral alinhado.

```
classdef ALG_COMPET < handle
    properties
        xi % matriz de amostras de treinamento
        xin % vetor de entrada do k-ésimo padrão
        %xnk % vetro normalizado de entrada do k-ésimo padrão
        w % matriz do pesos
        wn % vetro normalizado de pesos
        eta % taxa de aprendizagem
        lx % nº de linhas das matizes de pesos e de treinamento (nº de entrdadas)
        cx % nº de colunas da matriz de amostras de treinamneto (nº de amostras)
        cw % nº de colunas da matriz de pesos (nº de neurônios)
    end
    methods
        function obj = ALG_COMPET(n_ent,n_amost, n_class)
            % inicialização da matriz de pesos e de amostaras
            obj.xi = -1*ones(n_ent+1, n_amost);% Sem bias
            %obj.w = 2*rand(n_ent+1,n_class)-1;
%             nw = ceil(n_amost);
%             sxi = 0;
%              for j = 1:n_class
%                  for i = 1:n_ent
%                       for k = 1:nw
%                            %sxi = sxi + obj.xi(i,k);
%                       end
%                      %obj.w(i,j) = (sxi/nw)*j;
%                  end
%              end
            obj.lx = n_ent+1;% nº de entradas sem bias
            obj.cx = n_amost;% nº de amostras
            obj.cw = n_class;% nº de neurônios
        end
        function nor(obj,entradat)
            obj.xi(2:obj.lx,:) = entradat;
            obj.w = 6*rand(obj.lx,obj.cw)-3;% Inicialização aleatória da matriz de↙
pesos
            obj.w(1,:) = -1;
            % Normalização dos vetores de pesos e de treinamento
             for xc = 1:obj.cx
                 obj.xin(:,xc) = obj.xi(:,xc)/norm(obj.xi(:,xc));
             end
             for wc = 1:obj.cw
                 obj.wn(:,wc) = obj.w(:,wc)/norm(obj.w(:,wc));
             end
            plot(obj.wn(2,:),obj.wn(3,:),'or')
            hold on
            plot(obj.xin(2,1:51),obj.xin(3,1:51),'b')
            plot(obj.xin(2,52:102),obj.xin(3,52:102),'y')
            %sphere
            grid
        end
        function [mdist, W] = trei(obj,txap,nep)
            obj.eta = txap; %Taxa de aprendizagem
            a=0;
            dist = [];
            mdist = [];
```

```
            ind = 0;
            epoca = 1;
            while a==0
                a=0;
                for k = 1:obj.cx
                    for j = 1:obj.cw
                        somatxw = 0;
                        for i = 1:obj.lx
                            somatxw = somatxw + (obj.xin(i,k)-obj.wn(i,j))^2;
                        end
                        dist(j) = sqrt(somatxw);
                    end
                    [mdist, ind] = min(dist);
                    obj.wn(:,ind) = obj.wn(:,ind)+ obj.eta*(obj.xin(:,k)-obj.wn(:,↙
ind));
                end
                if epoca == nep
                    a=1;
                end
                epoca=epoca+1;
            end
            plot(obj.wn(2,:),obj.wn(3,:),'or')
            hold on
            plot(obj.xin(2,1:51),obj.xin(3,1:51),'b')
            plot(obj.xin(2,52:102),obj.xin(3,52:102),'y')
            grid
            W = obj.wn;
        end
%         function ind = valid(obj, entradav, pesost)
%             obj.xi = entradav;
%             % Normalização dos vetores de pesos e de treinamento
%             for xc = 1:obj.cx
%                 obj.xin(:,xc) = obj.xi(:,xc)/norm(obj.xi(:,xc));
%             end
%             obj.wn = pesost;
%             plot(obj.wn(1,:),obj.wn(2,:),'*r')
%             hold on
%             plot(obj.xin(1,1:51),obj.xin(2,1:51),'*b')
%             plot(obj.xin(1,52:102),obj.xin(2,52:102),'*y')
%             grid
%             figure
%             for k = 1:obj.cx
%                     for j = 1:obj.cw
%                         somatxw = 0;
%                         for i = 1:obj.lx
%                             somatxw = somatxw + (obj.xin(i,k)-obj.wn(i,j))^2;
%                         end
%                         dist(j) = sqrt(somatxw);
%                     end
%                     [mdist ind(k)] = min(dist);
%                     if ind == 1
%                          plot(obj.xin(1,k),obj.xin(2,k),'*k')
%                     else
%                         plot(obj.xin(1,k),obj.xin(2,k),'*r')
%                     end
```

```
%                       hold on
%                   end
%                   plot3(obj.wn(1,:),obj.wn(2,:),'*y')
%                   grid
%               end
    end

end
```

```
classdef ALG_SOM < handle
    properties
        xi % matriz de amostras de treinamento
        xin % vetor de entrada do k-ésimo padrão
        w % matriz do pesos
        wn % vetro normalizado de pesos
        eta % taxa de aprendizagem
        gau % Função de vizinhança lateral dos neurônios no espaço de saída
        lx % nº de linhas das matizes de pesos e de treinamento (nº de entrdadas)
        cx % nº de colunas da matriz de amostras de treinamneto (nº de amostras)
        cw % nº de colunas da matriz de pesos (nº de neurônios)
    end
    methods
        function obj = ALG_SOM(n_ent,n_amost, n_neu)
            % Criação da matriz de pesos e de amostaras
            obj.xi = -1*ones(n_ent+1, n_amost);% O +1 é para o bias -1
            obj.w = ones(n_ent+1,n_neu);% matriz de pesos
%             nw = ceil(0.2*n_amost);
%             sxi = 0;
%             for j = 1:n_neu
%                 for i = 1:n_ent
%                     for k = 1:nw
%                         sxi = sxi + obj.xi(i,k);
%                     end
%                     obj.w(i,j) = (sxi/nw)*5*j;
%                 end
%             end
            obj.lx = n_ent+1;% nº de entradas
            obj.cx = n_amost;% nº de amostras
            obj.cw = n_neu;% nº de neurônios
        end

        function nors(obj,entradat,iniw)
            obj.xi(2:obj.lx,:) = entradat;
            if iniw == 'aleat'
            obj.w = 6*rand(obj.lx,obj.cw)-3;% Inicialização aleatória da matriz de↙
pesos
            obj.w(1,:) = -1;
            end
%             if iniw == 'aleat'
%             obj.w = 6*rand(obj.lx,obj.cw)-3;% Inicialização grade quadrada da matriz↙
de pesos
%             obj.w(1,:) = -1;
%             end
%             if iniw == 'aleat'
%             obj.w = 6*rand(obj.lx,obj.cw)-3;% Inicialização grade exagonal da matriz↙
de pesos
%             obj.w(1,:) = -1;
%             end
            % Inicialiazação e Normalização dos vetores de pesos e de treinamento
             for xc = 1:obj.cx
                 obj.xin(:,xc) = obj.xi(:,xc)/norm(obj.xi(:,xc));
             end
             for wc = 1:obj.cw
                 obj.wn(:,wc) = obj.w(:,wc)/norm(obj.w(:,wc));
```

```
            end
            plot(obj.wn(2,:),obj.wn(3,:),'or')
            hold on
            plot(obj.xin(2,1:51),obj.xin(3,1:51),'-g')
            plot(obj.xin(2,52:102),obj.xin(3,52:102),'-b')
            grid
        end
        function treis(obj,txap,nep,r_gau)
            obj.eta = txap; %Taxa de aprendizagem fixa
%             obj.eta = txap; %Taxa de aprendizagem variável
            a=0;
            ind = 0;
            epoca = 1;
            while a==0
                a=0;
                % Cálculo da distância entre a amostra atual e todos os vetores w
                for k = 1:obj.cx
                    for j = 1:obj.cw
                        somatxw = 0;
                        for i = 1:obj.lx
                            somatxw = somatxw + (obj.xin(i,k)-obj.wn(i,j))^2;
                        end
                        dist(j) = sqrt(somatxw);
                    end
                    [mdist, ind] = min(dist); %determinação do neurônio vencedor
                    % Cálculo da distância da vizinhança e atualização
                    for j = 1:obj.cw
                        somatww = 0;
                        for i = 1:obj.lx
                            somatww = somatww + (obj.wn(i,ind)-obj.wn(i,j))^2;
                        end
                        distr = sqrt(somatww);
                        %dp = r_gau*exp-()% Desvio padrão variavel
                        obj.gau = exp(-(distr^2/(2*(r_gau)^2)));
                        obj.wn(:,j) = obj.wn(:,j)+ obj.eta*obj.gau*(obj.xin(:,k)-obj.wn↙
(:,j));
                    end

                end
                if epoca == nep
                    a=1;
                end
                epoca=epoca+1;
            end
            plot(obj.wn(2,:),obj.wn(3,:),'or')
            hold on
            plot(obj.xin(2,1:51),obj.xin(3,1:51),'-g')
            plot(obj.xin(2,52:102),obj.xin(3,52:102),'-b')
            grid
            W = obj.wn;
        end

%         function ind = valid(obj, entradav, pesost)
%             obj.xi(2:obj.lx,:) = entradav;
%             % Normalização dos vetores de pesos e de treinamento
```

```
%             for xc = 1:obj.cx
%                 obj.xin(:,xc) = obj.xi(:,xc)/norm(obj.xi(:,xc));
%             end
%             obj.wn = pesost;
%             plot3(obj.wn(1,:),obj.wn(2,:),obj.wn(3,:),'*r')
%             hold on
%             plot3(obj.xin(1,:),obj.xin(2,:),obj.xin(3,:),'*')
%             grid
%             figure
%             for k = 1:obj.cx
%                     for j = 1:obj.cw
%                         somatxw = 0;
%                         for i = 1:obj.lx
%                             somatxw = somatxw + (obj.xin(i,k)-obj.wn(i,j))^2;
%                         end
%                         dist(j) = sqrt(somatxw);
%                     end
%                     [mdist ind(k)] = min(dist);
%                     if ind == 1
%                          plot3(obj.xin(1,k),obj.xin(2,k),obj.xin(3,k),'*k')
%                     else
%                         plot3(obj.xin(1,k),obj.xin(2,k),obj.xin(3,k),'*r')
%                     end
%                     hold on
%             end
%             plot3(obj.wn(1,:),obj.wn(2,:),obj.wn(3,:),'*y')
%             grid
%         end
    end

end
```

8. Considere a distribuição dos padrões que tem como base em um círculo com raio igual a 0.25 centrado origem. Os pontos +1 e -1 de cada eixo são centros de quatro semicírculos que se interceptam no interior a as regiões que excluem o círculo de raio igual a 0.25 do quadrado originando quatro classes. Gere aleatoriamente os dados que venham formar estas distribuições de dados. Utilize a rede SOM de modo a quantizar através da distribuição de neurônios a distribuição dos dados.

   RESOLUÇÃO:

   Foi feito a geração dos dados através da equação do círculo:

$$x = \sqrt{r^2 - (y - y_0)^2} + x_0$$

$$y = \sqrt{r^2 - (x - x_0)^2} + y_0$$

   A classificação dos dados foi feito através de um vetor $[QXQYHV]$, em que para cada ponto gerado QX representa um dos dois quadrantes no eixo X, podendo assim assumir o valor 0 ou 1, o equivalente ocorre para QY no eixo Y, H representa se o ponto gerado encontra-se dentro de um semi-círculo horizontal e V para um semi-círculo vertical. Com este vetor é possível classificar o ponto gerado, visto que se o ponto estiver com valores de H e V iguais a 1 significa que o ponto encontra-se dentro de dois semi-círculos simultaneamente, ou seja, na região de interesse. Os valores de QX e QY descrevem em quais das quatro regiões os pontos se encontram.

   A rede foi treinada utilizando a *toolbox* do MATLAB, com o método *newsom*, com uma arquitetura 10x10 uniformimente distribuida. Com 500 pontos gerados e 200 iterações, tem-se:

   As especificações definidas para a execução do treinamento com o algoritmo implementado são as seguintes:

   N de neurônios: 100
   N de épocas: 500
   Taxa de aprendizagem constante: 0,1
   N de entradas: 2
   Desvio padrão constante: 0,05

   Foi implementado o mesmo algoritmo SOM já implementado para classificar os dados destacados em amarelo pertencentes as quatro classes da figura 9. Os pesos (crculos vermelhos) e os dados de treinamento (asteriscos azuis) das classes mencionadas acima foram gerados aleatoriamente e normalizados, figura 10.

   Após o treinamento, é possível notar na figura 10 que os pesos da maioria dos neurônios se deslocaram para a região dos dados de treinamento, ocorrendo poucos erros.

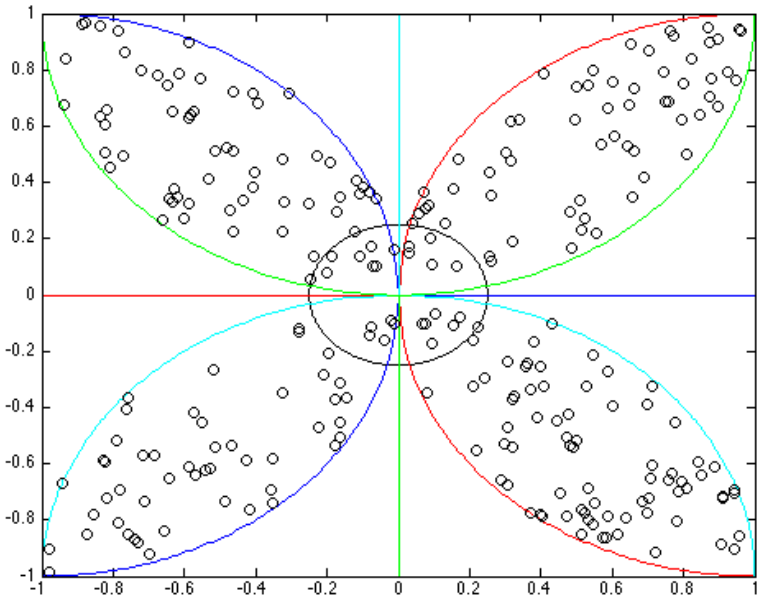

Figura 6: Desenho das regioes de interesse.

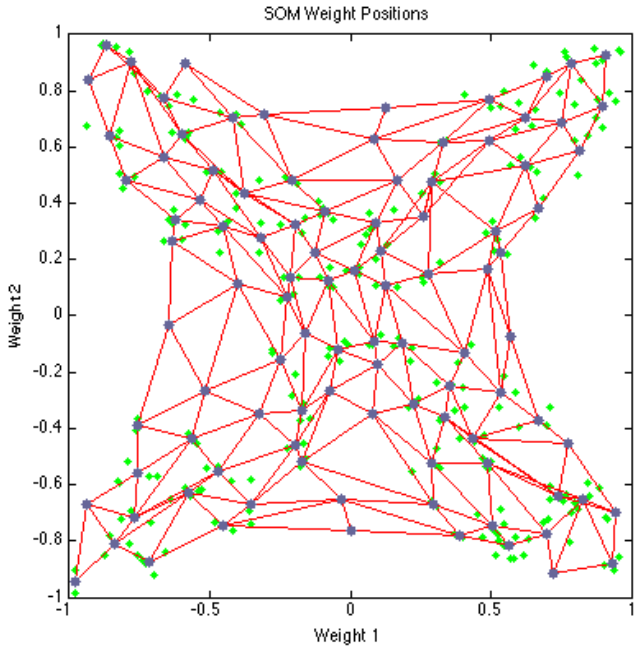

Figura 7: Resultado da rede.

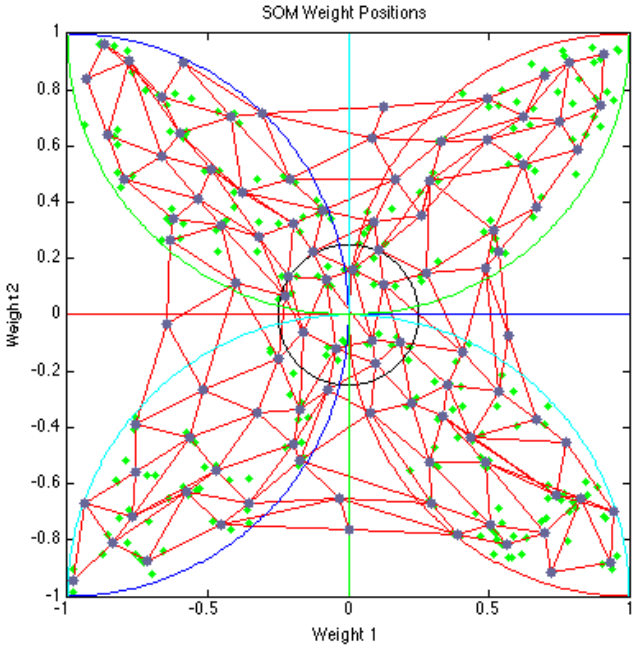

Figura 8: Resultado da rede com as regioes.

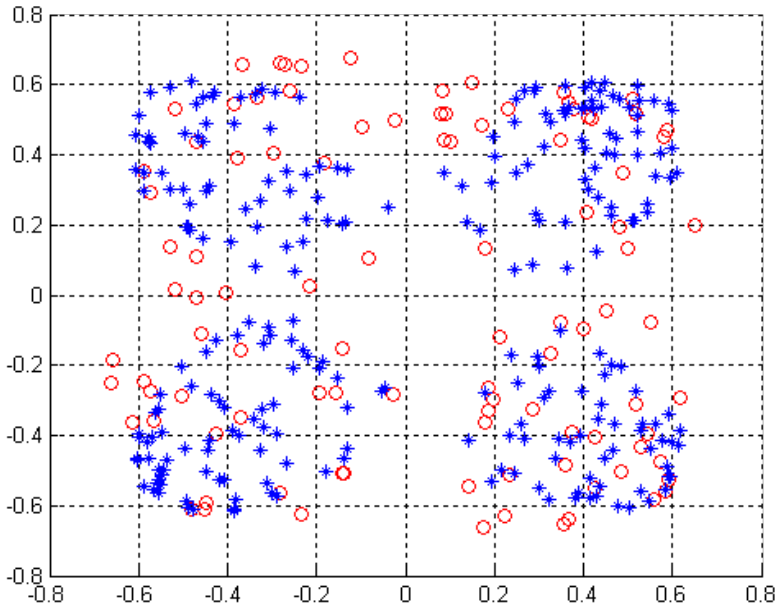

Figura 9: Inicializações dos dados de treinamento e dos pesos.

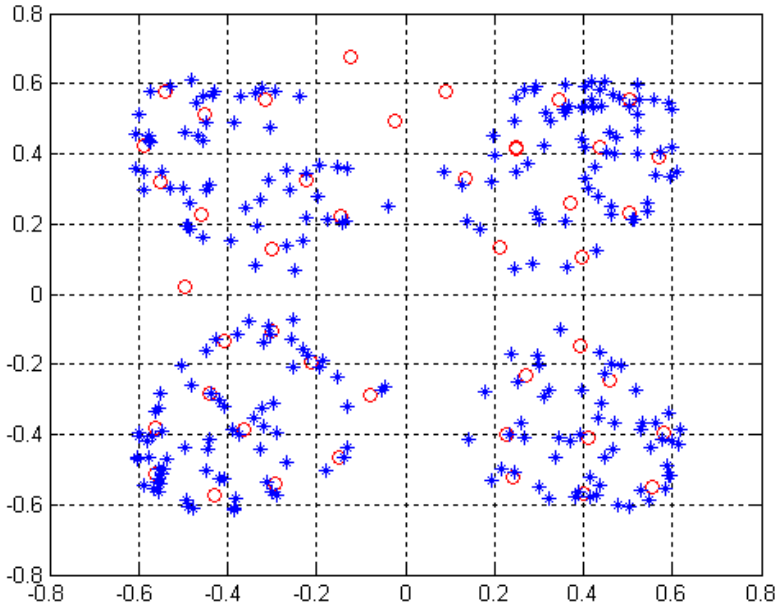

Figura 10: Resultado da rede SOM treinada que foi implementada.

9. Pesquise e apresente o formalismo do algoritmo K-means por lote.

10. Pesquise e apresente o formalismo do algoritmo SOM por lote.

RESOLUÇÃO:

Os algoritmos de particionamento visam subdividir os dados de entrada em números fixos de grupos pré-definido e otimizar funções critérios de adequação. Os mais populares são: K-means e Fuzzy C-means.

O Mapa Auto-Organizável de Kohonen (SOM) [Kohonen 1995] é uma rede neural não-supervisionada de aprendizado competitivo que possui propriedades de agrupamento e de visualização. Diferente do K-means, a rede SOM usa um conjunto de interações de vizinhança para aproximar a interação lateral neural e descobrir a estrutura topológica escondida nos dados, e além do vetor que obteve o melhor valor de similaridade (o vencedor), seus vizinhos também são atualizados, resultando em regiões nas quais neurônios em uma mesma vizinhança sejam bastante similares. Tal método também pode ser considerado como um algoritmo que mapeia dados de alta dimensionalidade espacial, R^p, em um espaço de dimensionalidade reduzida, geralmente 1D, 2D ou 3D, chamado mapa. Essa projeção permite a partição das entradas em grupos similares enquanto preserva sua topologia.

Considerando o algoritmo de mapa auto-organizável por lote introduzido por [F. Badran and Thiria 2005]. Seja $E = 1, ..., n$ o conjunto de entradas, onde cada entrada $x_i = (x_{i1}, ..., x_{ip})(i = 1, ..., n)$ pertence a R^p. Cada neurônio do mapa é representado por um protótipo $w_c = (w_{c1}, ..., w_{cp})(c = 1, ..., m)$ que também pertence a R_p.

O algoritmo SOM de treinamento por lote [F. Badran and Thiria 2005] é um algoritmo iterativo de duas etapas (afetação e representação, discutidos em seguida) onde todo o conjunto de dados (E) é apresentado ao mapa antes de algum ajuste ser feito. O algoritmo SOM por lote minimiza a seguinte função objetiva:

$$J = \sum_{i=1}^{n} \sum_{r=1}^{m} K^T(\delta(f^T(x_i), r)) d^2(x_i, w_r) \tag{1}$$

onde f é a função de alocação e $f(x_i)$ representa o neurônio do mapa correspondente ao indivíduo x_i, e $\delta(f(x_i)$, r é a distância entre o neurônio r do mapa e o neurônio correspondente a x_i. Assim, K^T, que é parametrizada por T (onde T representa a temperatura), é a função kernel de vizinhana que define a região de influência de cada neurônio r. Essa função objetivo é minimizada apenas para um valor de T fixo.

Essa função de custo é uma extensão da função de custo do K-means, onde a distância euclidiana é substituída por:

$$d^T(x_i, w_f(x_i)) = \sum_{r=1}^{m} K^T(\delta(f^T(x_i), r)) d^2(x_i, w_r) \tag{2}$$

onde

$$d^2(x_i, w_r) = \sum_{j=1}^{p} (x_{ij} - w_{rj})^2 \tag{3}$$

é a distância euclidiana. Essa distância generalizada é uma soma ponderada das distâncias euclidianas entre x_i e todos os vetores de referência da vizinhança do neurônio $f(x_i)$. Todos os neurônios do mapa são levados em consideração.

Quanto T é fixo, a minimização de J é realizada em duas etapas iterativas: uma afetação e uma representação. Durante a etapa de afetação, os vetores de referência (protótipos) permanecem fixos. A função de custo J é minimizada em relação à função de alocação e cada indivíduo x_i é assinalado ao seu neurônio mais próximo:

$$c = f^T(x_i) = \arg min_{1 \leq r \leq m} d^T(x_i, w_r) \tag{4}$$

Durante a etapa de representação, a função de alocação é fixada. A função de custo J é minimizada em relação aos protótipos. O protótipo w_c é atualizado para cada neurônio de acordo com:

$$w_c = \frac{\sum_{i=1}^{n} K^T(\delta(fT(x_i, c)) x_i}{\sum_{i=1}^{n} K^T(\delta(f^T(x_i, c))} \tag{5}$$

O algoritmo de mapa auto-organizável por lote (BSOM) [F. Badran and Thiria 2005] pode ser resumido como:

1) Inicialização
   Fixe o número m de neurônios (grupos);
   Fixe δ; Fixe a função de kernel K^T;
   Fixe o número de iterações Niter;
   Fixe T_{min}, T_{max}; Faça $T \leftarrow T_{max}$; Faça $t \leftarrow 0$;
   Selecione aleatoriamente m protótipos distintos $w_c(0)$,$E(c = 1; ..., m)$;
   Inicialize o mapa $L(m; W^0)$, onde $W^0 = (w_1^{(0)}, \cdots, w_m^{(0)})$;
   Afete cada objeto x_i ao neurônio mais próximo (grupo) do mesmo de acordo com a equao 4.

2) Etapa 1: Representação.
   Faça $T = T_{max}(\frac{T_{min}}{T_{max}})^{\frac{t}{N_{iter}-1}}$
   Mantenha a função de alocação fixa;
   Calcule os protótipos $w_c^{(t)}(c = 1, ..., m)$ de acordo com a equação 5;

3) Etapa 2: Afetação.
   Os protótipos $w_c^{(t)}(c = 1, ..., m)$ são fixados. Afete cada indivíduo $x_i(i = 1, ..., n)$ ao neurônio mais próximo do mesmo de acordo com a equação 4;

4) Critério de parada.
   se $T = T_{min}$, pare; caso contrário, faça $t = t + 1$ e vá para o passo 2 (Etapa 1).

Trabalhos

1. Pesquise e apresente um trabalho sobre a reconstrução tridimensional usando a rede SOM e a rede Neuro-GAS.

2. Pesquise e apresente um trabalho sobre Neurofuzzy.

Data de entrega: 23/05/2013

A entrega e apresentação dos trabalhos correspondem a um processo de avaliação. Portanto a presença é obrigatório.

Os trabalhos e a lista podem ser feito em grupo de até três componentes.

Na apresentação os componentes serão submetidos a questionamentos sobre a solução da lista e o desenvolvimento dos trabalhos.

Desenvolvimento da Pesquisa

Pesquise e apresente um trabalho sobre Neurofuzzy.

Os sistemas fuzzy são implementados a partir do conhecimento empírico fornecido por especialistas, já as Redes Neurais Artificiais necessitam do conhecimento implícito extraído de um conjunto de dados. Visando aproveitar essas duas características inerentes a cada uma dessas técnicas, os pesquisadores desenvolveram uma técnicas para gerar um modelo híbrido e assim, minimizar as deficiências. Os sistemas híbridos são a combinação de duas ou mais técnicas, em que, uma pode ser aplicada para melhorar as deficiências de outra. Assim, os Sistemas Neuro-Fuzzy (SNF) é um tipo de sistema híbrido incorporado pela combinação entre os Sistemas Fuzzy e as Redes Neurais Artificiais.

Os SNF é um Sistema de inferência Fuzzy, cujos parâmetros são ajustados através de algoritmos de aprendizagem de redes neurais, como ilustrado na figura 11.

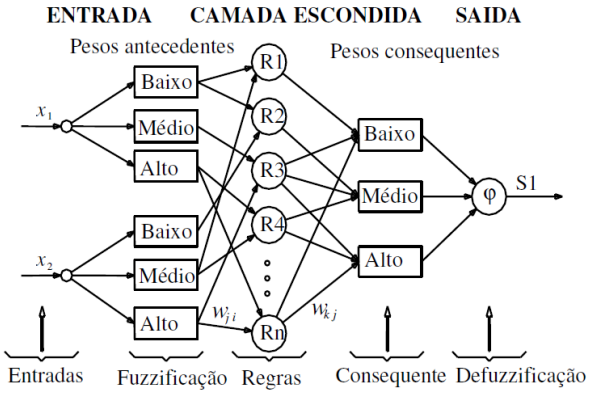

Figura 11: Sistema Fuzzy.

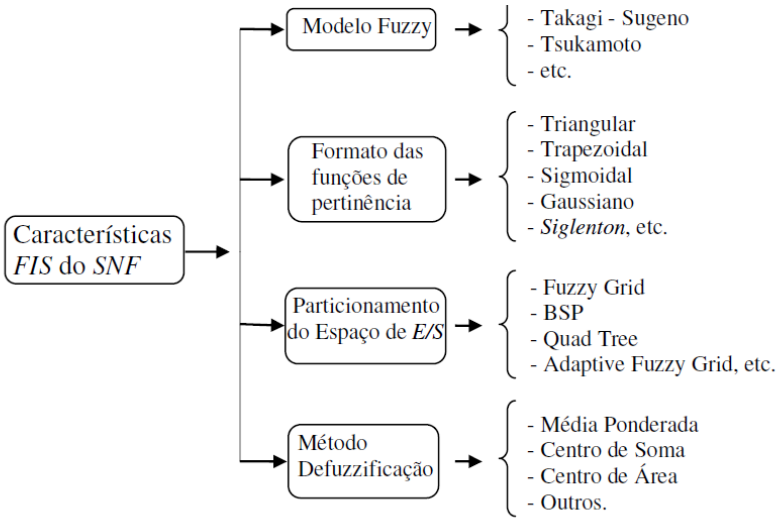

Figura 12: Caracteristicas Fuzzy.

As características Fuzzy do SNF são agrupadas em 4 Sub-classes demonstradas no esquema abaixo.

As características neurais do SNF estão divididas em três sub-classes apresentadas abaixo.

Alguns dos principais modelos de SNF são:
Adaptative Network based Fuzzy Inference Systems (ANFIS)
Neuro Fuzzy Classification (NEFCLASS)
Fuzzy Self-organized Map (FSOM)

As vantagens da utilização de SNF são:
Grande potencial para aplicações que combinam conhecimento qualitativo com robustez
Possuem interfaces de alto nível, de rápida computação e programação amigável. Apresentando conveniência para a extração de conhecimentos através do aprendizado.
Ausência da necessidade do conhecimento prévio do processo e melhora no processamento de ruídos dos dados.
Capacidade de auto-aprendizado, auto-organização e auto-direcionamento.

Como desvantagens o SNF tem:
Número de entradas reduzidos
Limitação da construção de sua própria estrutura, limitados pelo elevado número de regras.