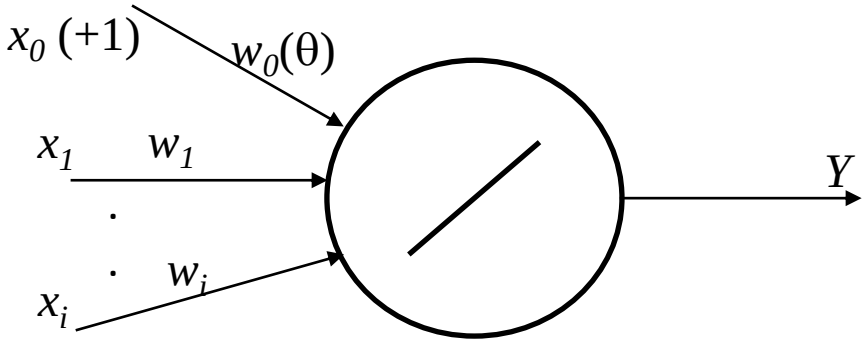
Neurônio do tipo *Adaline* e regra Delta

profº Mauricio Conceição Mario

Modelo de neurônio *Adaline* e regra Delta

- O modelo de neurônio *Adaline* se caracteriza pelo uso de uma função de ativação linear – $f(u) = u$, onde $u = \sum_{i=0}^{n} x_i w_i$

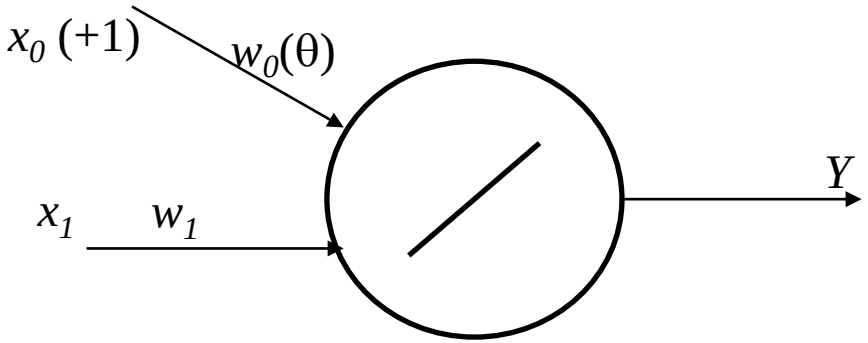


(Braga, 2007)

- Não há um limiar de ativação para o modelo *Adaline*. O parâmetro θ é chamado de **termo de polarização** e pode ser considerado no cálculo da saída *Y*.
- No neurônio do tipo *Adaline* o parâmetro θ corresponde a um grau de liberdade a mais para o neurônio, que resulta em um deslocamento da função de ativação em relação à origem do sistema de coordenadas.
- No neurônio do tipo *Perceptron* a interpretação do parâmetro θ é a de que este determina a separação linear do espaço de entrada.

Modelo de neurônio *Adaline* e regra Delta

- Saída Y: $Y = w_1 * x_1 + w_0$ (*1)



Portanto Y pode ser considerada uma função linear genérica de x_1, parametrizada por w_0 e w_1. A saída Y corresponde a uma combinação linear das entradas x_i, em que os pesos da combinação linear são obtidos através do treinamento, podendo estes assumir quaisquer valores reais.

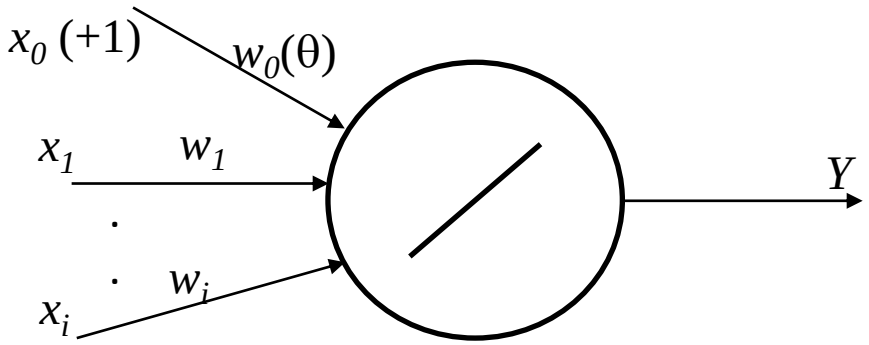
Para um *Adaline* de n entradas a saída Y é calculada através da expressão:

$$Y = w_0 + w_1 * x_1 + w_2 * x_2 + ... + w_n * x_n$$

onde os parâmetros w_i são resultado do treinamento.

(Braga, 2007)

Modelo de neurônio *Adaline* e regra Delta



- A **regra Delta** consiste na atualização do vetor de pesos *w* em direção contrária à variação do gradiente de erro quadrático (saída desejada – saída obtida):

Sendo o erro quadrático: $E = \frac{1}{2}(Yd - Y)^2$

$$\Delta w_i \propto - \partial E / \partial w_i$$

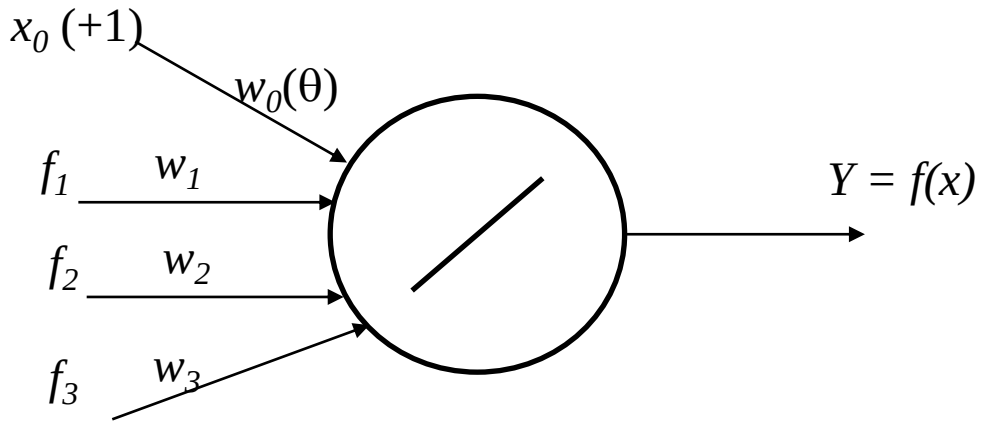
e $$w_i(n+1) = w_i(n) + \eta\, e\, x_i(n)$$

onde η = constante de proporcionalidade que define a velocidade com que o vetor de pesos é atualizado (ou taxa de aprendizado da rede).

(Braga, 2007)

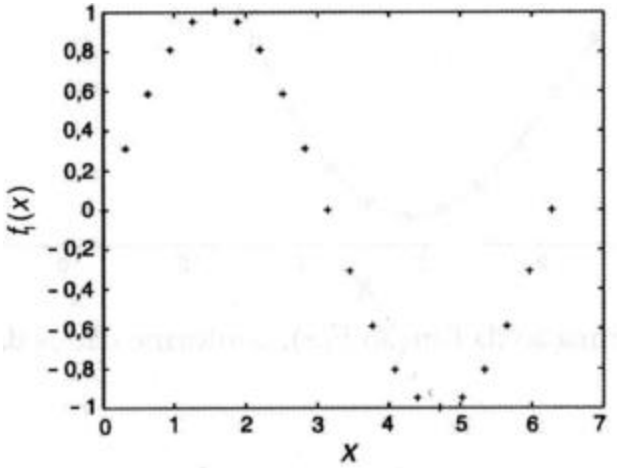
Modelo de neurônio *Adaline* e regra Delta

- O modelo de neurônio *Perceptron* se caracteriza como separador linear e o modelo de neurônio *Adaline* é utilizado para fazer combinação linear de funções ou aproximação funcional. Exemplo: combinação linear

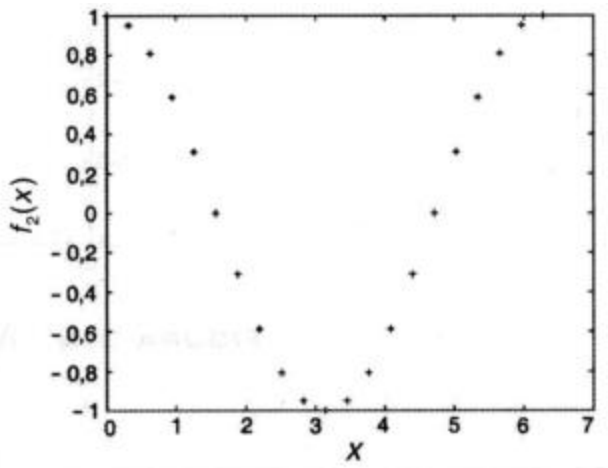



```
f1[x]= Math.sin(x*Math.PI/180);
f2[x]= Math.cos(x*Math.PI/180);
f3[x]= x*Math.PI/180;
f[x] = -Math.PI + 0.565*f1[x] + 2.657*f2[x] + 0.674*f3[x];
```

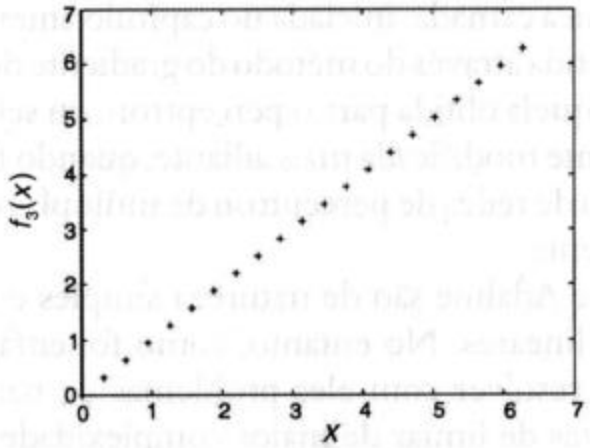
(Braga, 2007)



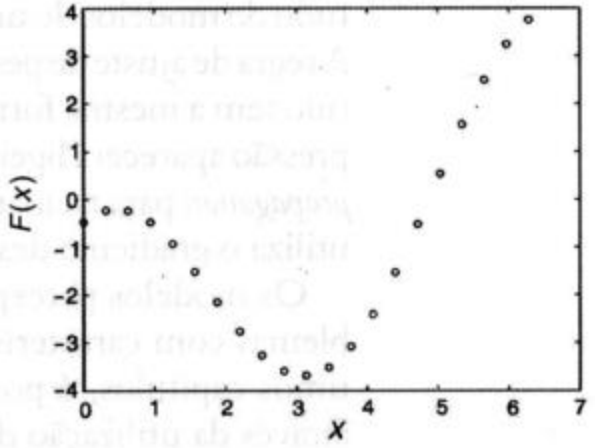
(a) Dados amostrados da função $f_1(x) = \text{sen}(x)$



(b) Dados amostrados da função $f_2(x) = \cos(x)$



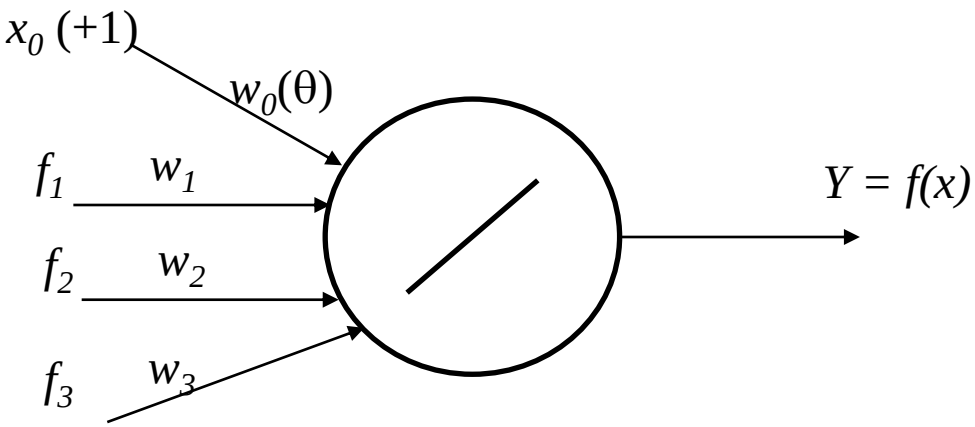
(c) Dados amostrados da função $f_3(x) = x$



(d) Dados amostrados da função combinada $F(x)$

(Braga, 2007)

Modelo de neurônio *Adaline* e regra Delta




```
f1[x]= Math.sin(x*Math.PI/180);
f2[x]= Math.cos(x*Math.PI/180);
f3[x]= x*Math.PI/180;
f[x] = -Math.PI + 0.565*f1[x] + 2.657*f2[x] + 0.674*f3[x];
```

Os coeficientes -PI, 0.565, 2.657 e 0.674 podem ser considerados os pesos que satisfazem a combinação linear proposta para gerar f(x).

Classe combinação_linear

```
public class combinação_linear {
        double [] f1 = new double[300];
    double [] f2 = new double[300];
    double [] f3 = new double[300];
    double [] f = new double[300];
    double [] t = new double[300];

    double w [][]= {{-2.4013}, {0.393}, {1.902}, {0.429}};

    double taxa_aprendizado = 0.000027;

  public void iteração_II() {

      for(int x = 0; x < 300; x++){
          f1[x]= Math.sin(x*Math.PI/180);
          f2[x]= Math.cos(x*Math.PI/180);
          f3[x]= x*Math.PI/180;
          f[x] = -Math.PI + 0.565*f1[x] + 2.657*f2[x] + 0.674*f3[x];
      }
        int n = 0;
        while (n < 32){ //nº de treinamentos
 w = entradaf1(w);
     System.out.println("pesos w = \t" + w[0][0]);
     System.out.println("pesos w = \t" + w[1][0]);
     System.out.println("pesos w = \t" + w[2][0]);
     System.out.println("pesos w = \t" + w[3][0]);
     n = n + 1;
     System.out.println("número de treinamentos " + n + "\n");
    }
 }
```

Classe combinação_linear

```
public double [][] entradaf1(double [][]w){
System.out.println("entrada f1 ");

    for (int i = 0; i <300; i++){
       t[i] = w[0][0] + w[1][0]*f1[i] + w[2][0]*f2[i] + w[3][0]*f3[i];
   w[0][0] = w[0][0] + taxa_aprendizado*Math.pow((f[i]-t[i]),2.0)*0.5;
   w[1][0]= w[1][0] + taxa_aprendizado*Math.pow((f[i]-t[i]),2.0)*0.5*f1[i];
   w[2][0]= w[2][0] + taxa_aprendizado*Math.pow((f[i]-t[i]),2.0)*0.5*f2[i];
   w[3][0]= w[3][0] + taxa_aprendizado*Math.pow((f[i]-t[i]),2.0)*0.5*f3[i];
}

return w;
}
```

```
public double[] testa_af(double[][] ww){
        System.out.println("TESTE DA REDE NEURAL \n "  );
        double []ff = new double[300];
        double []gg = new double[600];

    System.out.println("pesos resultante do treinamento  "  );
    System.out.println("pesos w = \t" + ww[0][0]);
    System.out.println("pesos w = \t" + ww[1][0]);
    System.out.println("pesos w = \t" + ww[2][0]);
    System.out.println("pesos w = \t" + ww[3][0]);
```

Classe combinação_linear

```
        for (int i = 0; i < 300; i++)
          ff[i] = ww[0][0] + ww[1][0]*f1[i] + ww[2][0]*f2[i] + ww[3][0]*f3[i];
        int k = 0;
        for (int i = 0; i < 300; i++)
         gg[i] = f[i];
        for (int j = 300; j < 600; j++){
         gg[j] = ff[k];
         k++;
        }

        return gg;
    }
}
```

Classe teste

```
import javax.swing.JOptionPane;
import javax.swing.*;
import java.awt.*;

public class teste extends javax.swing.JFrame{

    static double [] pesos = new double[600];
    static double [] backp = new double[600];

    public static void main(String args[]){

    grafico g = new grafico();
    grafico gf;

    combinação_linear af= new   combinação_linear();

   double [][] x5 = {{3.09},{0.6},{2.233},{0.604}};
   double [][] x6 = {{3.1246},{0.5527},{2.6568},{0.6692}};
   af.iteração_II();
   //pesos = af.testa_af(x5);
   pesos = af.testa_af(af.w);
   //set_pesos(pesos);

JFrame aplicacao = new JFrame();
aplicacao.getContentPane().setBackground (new Color (255,255,255));
aplicacao.setTitle("Redes Neurais Artificiais");
aplicacao.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
aplicacao.add(g);
aplicacao.setSize(500, 480);
aplicacao.setVisible(true);

    }
   public double [] get_pesos (){
 return pesos;
   }
}
```

Classe grafico

```
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JPanel;

public class grafico extends JPanel{

    static String mx, mn;
   static double maximo , minimo , passo, a, b, c, d, e, f, l, h, i, j, m;

   double evidencia;
   double []t = new double [300];
   double []w = new double [300];
   private double []ff = new double [600];
   private double []fg = new double [600];

              teste   aa = new   teste ( );


   grafico(){}
```

```
 grafico (double [] shapes){
     ff = shapes;


 }


  public void paintComponent(Graphics g) {

  super.paintComponent(g);

minimo = 0; maximo = 1;
passo = (maximo - minimo)/10;

a = minimo;
b = minimo + passo;
c = minimo + 2*passo;
d = minimo + 3*passo;
e = minimo + 4*passo;
f = minimo + 5*passo;
l = minimo + 6*passo;
h = minimo + 7*passo;
i = minimo + 8*passo;
j = minimo + 9*passo;
m = minimo + 10*passo;

  g.setColor(Color.lightGray);
```

Classe grafico

```
 //linhas horizontais
int p = 70;//p = 80;
 do{
for (int i = 0; i < 320; i = i + 20)
g.drawLine(0, p, i, p);
p = p + 22;//20;
}
 while(p <= 340);

//linhas verticais
int q = 0;
 do{
for (int i = 0; i < 200; i = i + 20)
g.drawLine(q, 70, q, 332);
q = q + 20;
}
 while(q < 320);
```

```
  g.setColor(Color.red);
  g.drawString("*sinal 1 " , 60, 50);

  g.setColor(Color.blue);
  g.drawString("*sinal 2 " , 130, 50);
```

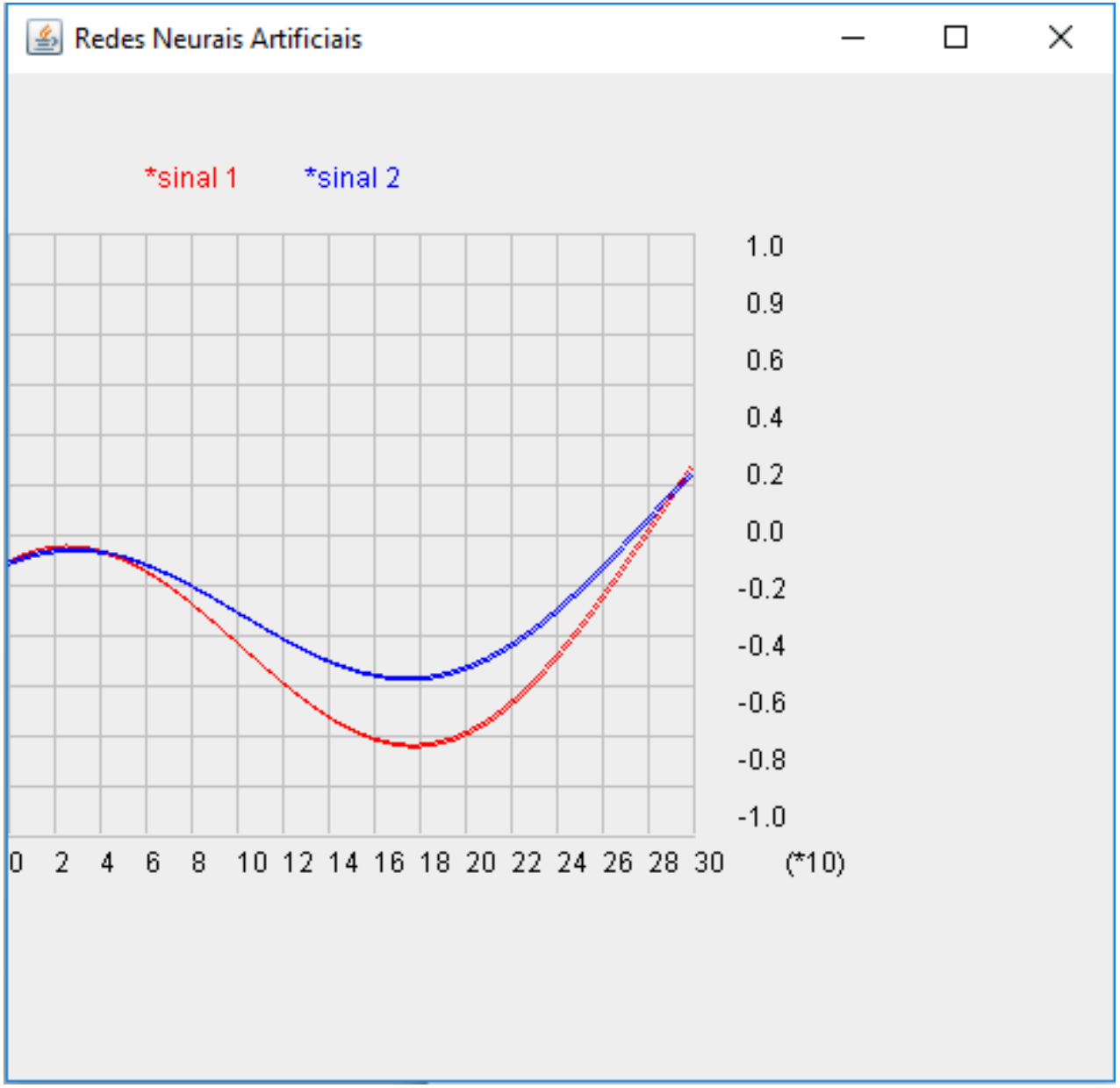
```
    g.setColor(Color.BLACK);
     int k = 0, escala_horizontal = 0;
     //escala horizontal
    for (int i = 0; i <= 300; i = i + 20 ){
    g.drawString(""+k , escala_horizontal, 350);
    escala_horizontal = escala_horizontal + 20;
    k = k + 2;}
    g.drawString("(*10)" , 340, 350);

//escala vertical
    g.drawString("-1.0   " , 320, 330); //+a
    g.drawString("-0.8   " , 320, 305); //+b
    g.drawString("-0.6   " , 320, 280); //+c
    g.drawString("-0.4   " , 320, 255); //+d
    g.drawString("-0.2   " , 320, 230); //+e
    g.drawString(" 0.0   " , 320, 205); //+f
    g.drawString(" 0.2   " , 320, 180); //+l
    g.drawString(" 0.4   " , 320, 155); //+h
    g.drawString(" 0.6   " , 320, 130); //+i
    g.drawString(" 0.9   " , 320, 105); //+j
    g.drawString(" 1.0   " , 320, 80);  //+m

    ff = aa.get_pesos();
```

Classe grafico

```
   System.arraycopy(ff, 0, fg, 0, 600);


for (int nn = 0; nn < 300; nn++){
    t[nn] = ff[nn];
}



     System.arraycopy(ff, 300, w, 0, 300);



 int xx = 0; double y = 0, n = 0;

for (int z = 0; z <= 299; z++){
y = +203 +(-250*t[z])*0.1;
n = +203 +(-250*w[z])*0.1;
g.setColor(Color.red);
g.drawLine(xx-1,(int) y-1, xx, (int)y);
g.setColor(Color.blue);
g.drawLine(xx-1,(int)n-1, xx, (int)n);
xx++;
}

}

}
```

f[x] = -Math.PI + 0.565*f1[x] + 2.657*f2[x] + 0.674*f3[x];

t[i] = w[0][0] + w[1][0]*f1[i] + w[2][0]*f2[i] + w[3][0]*f3[i];



entrada f1
pesos w = -2.3681403570448603
pesos w = 0.3938658056066081
pesos w = 1.8770701096089122
pesos w = 0.5326629790070988
número de treinamentos 29

entrada f1
pesos w = -2.366262732164276
pesos w = 0.39386173678286246
pesos w = 1.8756267041566208
pesos w = 0.5385500775775777
número de treinamentos 30

entrada f1
pesos w = -2.3642908293956983
pesos w = 0.3938437042660612
pesos w = 1.8741178321427694
pesos w = 0.5447504892917647
número de treinamentos 31

entrada f1
pesos w = -2.3622159451606533
pesos w = 0.3938092085328464
pesos w = 1.8725386186376358
pesos w = 0.5512951943410876
número de treinamentos 32

TESTE DA REDE NEURAL

pesos resultante do treinamento
pesos w = -2.3622159451606533
pesos w = 0.3938092085328464
pesos w = 1.8725386186376358
pesos w = 0.5512951943410876

Referências Bibliográficas

- Braga AP, Carvalho APLF, Ludermir TB. *Redes Neurais Artificiais: teoria e aplicações*. Livros Técnicos e Científicos, Rio de Janeiro – RJ; 2007.

- Haykin S. *Neural Networks – A Comprehensive Foundation.* Prentice-Hall; 1994.

- Haykin S. *Redes Neurais – Princípios e prática.* 2a ed.. Porto Alegre: Bookman; 2001.

- Hebb DO. *The Organization of Behavior.* John Wiley & Sons; 1949.

- Heckerman D. *Probabilistic Similarity Networks.* MIT Press, Cambridge, Massachussets; 1991.

- Hopfield JJ. *Neurons with graded response have collective computational properties like those of two-state neurons.* Proceedings of the National Academy of Sciences of the United States of America, 79, 2554-2558; 1982.

- Minsky, M.; Papert, P.. *Perceptrons: An introduction to computational geometry.* MIT Press, Massachussets, 1969.

Referências Bibliográficas

- Mario MC. *Proposta de Aplicação das Redes Neurais Artificiais Paraconsistentes como Classificador de Sinais Utilizando Aproximação Funcional*. Univ. Federal de Uberlândia, Dissertação de Mestrado, Uberlândia; 2003.

- McCarthy J. *Programs with commom sense.* In Proceedings of the Symposium on Mechanisation of Thought Processes, Vol. 1, pp. 77-84, London. Her Majesty's Stationery Office; 1958.

- McCulloch W, Pitts W. *A logical calculus of the ideas immanent in nervous activity.* Bulletin of Mathematical Biophysics 5, 115-133; 1943.

- Rosenblatt F. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms.* Spartan, Chicago; 1962.

- Rumelhart DE, McClelland JL. *Parallel Distributed Processing.* MIT Press, Cambridge, Massachussets; 1986.

- Turing A. *Computing machinery and intelligence.* Mind, 59, 433-460; 1950.