UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DIRETORIA DE PESQUISA E PÓS GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

JÔNATAS TRABUCO BELOTTI

IMPLEMENTAÇÃO PROJETO PRÁTICO 6.5: REDE RBF PARA CLASSIFICAÇÃO DE PADRÕES

RELATÓRIO

PONTA GROSSA
2017

JÔNATAS TRABUCO BELOTTI

IMPLEMENTAÇÃO PROJETO PRÁTICO 6.5: REDE RBF PARA CLASSIFICAÇÃO DE PADRÕES

Relatório apresentado como requisito parcial à obtenção de nota na disciplina de Fundamentos de Redes Neurais Artificiais do Programa de Pós-Graduação em Engenharia Elétrica, da Universidade Tecnológica Federal do Paraná– Campus Ponta Grossa.

Professor: Prof. Dr. Sérgio Okida

PONTA GROSSA
2017

SUMÁRIO

1 INTRODUÇÃO

As redes neurais funções de base radial (FBR, ou RBF do inglês *radial basis function*) possuem uma ampla variedade de aplicações, sendo possivel a sua utilização em quase todos os tipos de problemas tratados pelas redes Perceptron Multicamadas (PMC, ou MLP do inglês *Multilayer Perceptron*), inclusive aproximação de funções e classificação de padrões (SILVA; SPATTI; FLAUZINO, 2010).

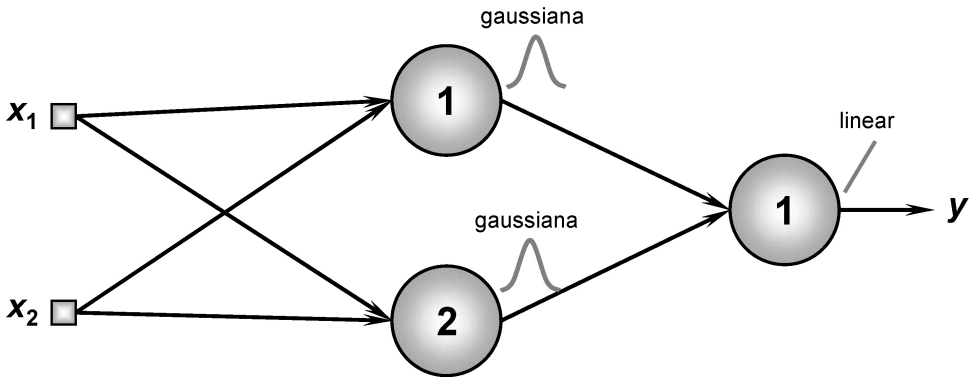
Ao contrário da MLP, a rede RBF possui apenas uma camada intermediária. Camada está onde as funções de ativação são do tipo gaussiana. Além disso, a RBF não possui limiar de ativação na camada intermediária, apenas na camada de saída (SILVA; SPATTI; FLAUZINO, 2010).

Esse relatório tem como objetivo descrever o desenvolvimento do Projeto Prático 6.5 do livro Redes neurais artificias para engenharia e ciências aplicadas de Silva, Spatti e Flauzino (2010). O projeto consiste na implementação, treinamento e teste de uma rede neural RBF para ser usada como classificadora de padrões na verificação da presença ou não de radiação em determinada substância nuclear.

1.1 ESTUDO DE CASO

O Projeto Prático 6.5 do livro Redes neurais artificias para engenharia e ciências aplicadas de Silva, Spatti e Flauzino (2010) mostra que a determinação da presença ou não de radiação em determinada substância nuclear pode ser realizada mediante a análise da concentração de duas variáveis $\{x_1$ e $x_2\}$. Através de 50 situações conhecidas decidiu-se então pela criação de uma rede neural do tipo RBF para determinar então se as substâncias são radioativas ou não. A Figura 1 apresenta a arquitetura da rede proposta pelo livro.

Figura 1 – Arquitetura proposta rede RBF



Fonte: (SILVA; SPATTI; FLAUZINO, 2010).

Pela Figura 1 verifica-se que a rede RBF a ser criada deve possuir 2 entradas, 2 neurônios na camada intermediaria e apenas 1 neurônio na camada de saída. Ainda analisando a Fi-

gura 1 nota-se que a função de ativação da camada de saída deve ser do tipo linear.

A padronização para classificar a saída da rede em 'presença de radiação' e 'ausência de radiação' é apresentada na Tabela 1.

Tabela 1 – Padronização saída RBF

Status de radiação	**Saída (y)**
Presença	1
Ausência	-1

Fonte: (SILVA; SPATTI; FLAUZINO, 2010).

2 DESENVOLVIMENTO DO PROJETO

A rede RBF proposta na Seção 1.1 foi desenvolvida na linguagem Java, a classe RBF é a responsável por implementar o funcionamento da rede, o código fonte da classe RBF está disponível no Apêndice A. Com o objetivo de facilitar o acesso ao código fonte da rede desenvolvida, o mesmo juntamente com um arquivo contendo o conjunto de dados de treinamento, um arquivo contendo o conjunto de dados de teste e o programa já compilado foram disponibilizados em um repositório do *GitHub*[1] que pode ser acessado pelo link <https://github.com/jonatastbelotti/redeRBF>.

Na Seção 2.1 é apresentado a primeira etapa do treinamento da RBF, a segunda etapa do treinamento é apresentada na Seção 2.2 e por fim na Seção 2.3 é apresentado o resultado obtido pela rede a partir do treinamento descrito nas seções 2.1 e 2.2.

2.1 TREINAMENTO CAMADA INTERMEDIÁRIA

A primeira etapa do treinamento da rede RBF consiste no ajuste dos pesos sinápticos da camada intermediária, que nada mais são do que os centros das funções gaussianas, e no cálculo da variância da função gaussiana para cada neurônio da camada intermediária por meio do algoritmo *k-means*.

A primeira etapa do treinamento da rede RBF desenvolvida foi executada com o conjunto de dados de treinamento presente no Anexo A. A Tabela 2 apresenta as coordenadas do centro de cada agrupamento juntamente com a sua variância.

Tabela 2 – Resultado primeira etapa treinamento

Agrupamento		Centro	Variância
1	$\mathbf{x_1}$	$0,5247954545454546$	$0,10126169714876032$
	$\mathbf{x_2}$	$0,7501318181818184$	
2	$\mathbf{x_1}$	$0,4531333333333333$	$0,0973760888888889$
	$\mathbf{x_2}$	$0,21089999999999998$	

Fonte: Autoria própria.

2.2 TREINAMENTO CAMADA SAÍDA

A segunda etapa do treinamento da RBF consiste no ajuste dos pesos sinápticos da camada de saída. O treinamento da camada de saída foi realizado utilizando a regra Delta generalizada com taxa de aprendizado de $\eta = 0,01$ e precisão de $\varepsilon = 10^{-7}$. Os pesos sinápticos

[1] No repositório do *GitHub* o caminho para acessar a classe RBF.java é './RedeRBF/src/Modelo/RBF.java'.

da camada de saída foram iniciados de maneira aleatória com valores entre 0 e 1. O conjunto de dados de treinamento utilizado foi o mesmo da Seção 2.1, disponibilizado no Anexo A.

A Tabela 3 apresenta os pesos sinápticos da camada de saída ao final do treinamento.

Tabela 3 – Segunda etapa treinamento

Parâmetro	Valor
$\mathbf{W_{1,1}^{(2)}}$	$0,3559855331978549$
$\mathbf{W_{2,1}^{(2)}}$	$-0,9888284080358978$
θ_1	$1,757645896276698$

Fonte: Autoria própria.

A segunda etapa do treinamento da RBF foi executada em 338 épocas e obteve um erro quadrático médio de $\mathrm{E_{qm}} = 0,22014719186640322$.

2.3 TESTE

Após o treinamento a rede RBF desenvolvida foi testada visando verificar sua capacidade de generalização, para tanto foram utilizadas 10 amostras de teste, essas amostras de teste são apresentadas na Tabela 4 e estão disponíveis no Anexo B. A Tabela 4 apresenta o resultado dos testes executado na rede RBF, nela é possível verificar qual a saída desejada para cada amostra de teste, juntamente com a saída dada pela rede antes e depois do pós-processamento. Na Tabela 4 também é possível verificar qual a porcentagem de acerto que a rede obteve ao fim do teste.

Tabela 4 – Resultado teste

Amostra	$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{d}$	$\mathbf{y}$	$\mathbf{y^{pos}}$
1	0.8705	0.9329	-1	-0.7712871187721616	-1
2	0.0388	0.2703	1	0.2600680194427229	1
3	0.8236	0.4458	-1	-0.10436691798752618	-1
4	0.7075	0.1502	1	0.7393696276524443	1
5	0.9587	0.8663	-1	-0.6689870105764796	-1
6	0.6115	0.9365	-1	-1.0551230688247575	-1
7	0.3534	0.3646	1	0.7127872599954558	1
8	0.3268	0.2766	1	0.9585853259342835	1
9	0.6129	0.4518	-1	0.1752447254409879	1
10	0.9948	0.4962	-1	-0.34107848305539173	-1
Taxa de acerto (%):				90%	

Fonte: Autoria própria.

Analisando os dados da Tabela 4 verifica-se que a taxa de acerto da rede RBF foi de 90%, no teste a rede errou apenas a amostra de teste 9.

Algumas possibilidades podem ser exploradas visando aumentar a taxa de acerto da rede. A quantidade de neurônios presentes na camada intermediaria pode ser alterada visando

determinar qual a melhor quantidade de neurônios na camada intermediária para esse problema em específico, isso pode ser realizado através do algoritmo de validação cruzada. O valor da precisão pode ser diminuído visando obter um erro quadrático médio menor ao final do treinamento, o que resulta em um melhor desempenho no teste da rede. Além disso os conjuntos de treinamento e de teste podem ser revistos, conjuntos de treinamento e de teste maiores tendem a compreender de melhor forma todos os elementos do problema, resultando em um melhor treinamento e consequentemente uma taxa de acerto maior na execução do teste.

3 CONCLUSÃO

Foi desenvolvida uma rede RBF para ser utilizada como classificadora de padrões na determinação de radiação em substâncias. A rede foi desenvolvida com 2 entradas, 2 neurônios na camada intermediária e 1 neurônio na camada de saída. Após o treinamento a rede obteve uma taxa de acerto de 90% na execução do teste.

Concluí-se que a rede RBF desenvolvida será capaz de suprir todas a necessidades do problema proposto visto que a mesma obteve uma taxa de acerto de 90%.

REFERÊNCIAS

SILVA, Ivan Nunes da; SPATTI, Danilo Hernane; FLAUZINO, Rogério Andrade. **Redes Neurais Artificiais Para Engenharia e Cincias Aplicadas - Curso Pratico**. 1. ed. São Paulo: ARTLIBER, 2010. ISBN 978-85-88098-53-4.

APÊNDICE A - IMPLEMENTAÇÃO DA CLASSE RBF EM JAVA

```
1  package Modelo;
2
   import Controle.Comunicador;
4  import Recursos.Arquivo;
   import Recursos.Numero;
6  import java.util.ArrayList;
   import java.util.List;
8  import java.util.Random;

10 /**
    *
12  * @author Jonatas Trabuco Belotti [jonatas.t.belotti@hotmail.com]
    */
14 public class RBF {

16   public static final int NUM_ENTRADAS = 2;
     private final int NUM_NEU_CAMADA_ESCONDIDA = 2;
18   public static final int NUM_NEU_CAMADA_SAIDA = 1;
     private final double LIMIAR_ATIVACAO = -1D;
20   private final double TAXA_APRENDIZADO = 0.01;
     private final double PRECISAO = Math.pow(10D, -7D);
22
     private double[] entradas;
24   private double[] saidasEsperadas;
     private double[] variancia;
26   private double[][] pesosCamadaEscondida;
     private double[][] pesosCamadaSaida;
28   private double[] saidaCamadaEscondida;
     private double[] saidaCamadaSaida;
30   private int numEpocas;

32   public RBF() {
       this.entradas = new double[NUM_ENTRADAS];
34     this.variancia = new double[NUM_NEU_CAMADA_ESCONDIDA];
       this.pesosCamadaEscondida = new double[NUM_NEU_CAMADA_ESCONDIDA
          ][NUM_ENTRADAS];
36     this.pesosCamadaSaida = new double[NUM_NEU_CAMADA_SAIDA][
          NUM_NEU_CAMADA_ESCONDIDA + 1];//+1 por causa do peso do
          limiar de ativacao
```

```
    this.saidaCamadaEscondida = new double[NUM_NEU_CAMADA_ESCONDIDA
        + 1];//+1 por causa do limiar de ativacao
38  this.saidaCamadaSaida = new double[NUM_NEU_CAMADA_SAIDA];
    this.saidasEsperadas = new double[NUM_NEU_CAMADA_SAIDA];
40  }

42  public boolean treinar(Arquivo arquivoTreinamento) {
    Comunicador.iniciarLog("Iniciando treinamento da rede RBF");
44
    arquivoTreinamento.abrirArquivo();
46
    //Realiza primeiro o treinamento da camada escondida
48  if (treinarCamadaEscondida(arquivoTreinamento) == false) {
      return false;
50  }

52  //Por ultimo realiza o treinamento da camada de saida
    if (treinarCamadaSaida(arquivoTreinamento) == false) {
54    return false;
    }
56
    Comunicador.addLog("Rede RBF treinada com sucesso");
58  imprimirPesosCamadaEscondida();
    imprimirPesosCamadaSaida();
60
    //Apenas se os 2 treinamentos forem bem sucedidos retorna
       verdadeiro
62  return true;
  }
64
  public void testar(Arquivo arquivoTeste) {
66  String log1;
    String log2;
68  String log3;
    boolean errou;
70  int numErros;

72  Comunicador.iniciarLog("Iniciando teste da rede RBF");

74  arquivoTeste.abrirArquivo();
    numErros = 0;
76
```

```
        //Para cada amostra do conjunto de teste
78      for (String linha : arquivoTeste.getLinhasArquivo()) {
          recuperarEntradas(linha);
80
          //Calculando saida da rede
82        calcularSaida();

84        log1 = "";
          log2 = "";
86        log3 = "";
          errou = false;
88        for (int neuronio = 0; neuronio < NUM_NEU_CAMADA_SAIDA;
             neuronio++) {
            log1 += String.format("%f ", saidasEsperadas[neuronio]);
90          log2 += String.format("%s ", Double.toString(
               saidaCamadaSaida[neuronio]));
            log3 += String.format("%f ", posProcessamento(
               saidaCamadaSaida[neuronio]));
92
            //Verifica se acertou ou errou a amostra
94          if (posProcessamento(saidaCamadaSaida[neuronio]) !=
               saidasEsperadas[neuronio]) {
              errou = true;
96          }
          }
98
          Comunicador.addLog("" + log1 + log2 + log3 + "");
100
          if (errou) {
102         numErros++;
          }
104     }

106     Comunicador.addLog("Fim do teste da RBF");
        Comunicador.addLog(String.format("Porcentagem de acerto: %.2f%%
           ", (100D / (double) arquivoTeste.getLinhasArquivo().size())
           * ((double) arquivoTeste.getLinhasArquivo().size() -
           numErros)));
108   }

110   private boolean treinarCamadaEscondida(Arquivo arquivoTreinamento
         ) {
```

```
        List[] gruposOmega;
112     double distanciaEuclidiana;
        double menorDistanciaEuclidiana;
114     double valorParcial;
        int neuronioMenorDistanciaEuclidiana;
116     int indiceAmostra;
        boolean mudouGrupoOmega;
118
        Comunicador.addLog("Treinamento camada escondida");
120
        //Iniciando o vetor que ira guardar os grupos omega
122     gruposOmega = new List[NUM_NEU_CAMADA_ESCONDIDA];
        for (int i = 0; i < NUM_NEU_CAMADA_ESCONDIDA; i++) {
124       gruposOmega[i] = new ArrayList();
        }
126
        try {
128       //Iniciando pesos da camada escondida com os primeiros
             valores do conjunto de entradas
          for (int i = 0; i < NUM_NEU_CAMADA_ESCONDIDA; i++) {
130         recuperarEntradas(arquivoTreinamento.getLinhaIndice(i));

132         for (int j = 0; j < NUM_ENTRADAS; j++) {
              pesosCamadaEscondida[i][j] = entradas[j];
134         }
          }
136
          do {
138         mudouGrupoOmega = false;
            indiceAmostra = 0;
140
            //Para cada amostra de treinamento
142         for (String linha : arquivoTreinamento.getLinhasArquivo())
               {
              indiceAmostra++;
144           recuperarEntradas(linha);
              menorDistanciaEuclidiana = Double.MAX_VALUE;
146           neuronioMenorDistanciaEuclidiana = 0;

148           //Calcula a distancia euclidoana da entrada para cada
                 neuronio
              for (int neuronio = 0; neuronio <
```

```
            NUM_NEU_CAMADA_ESCONDIDA; neuronio++) {
150           distanciaEuclidiana = 0D;

152           for (int x = 0; x < NUM_ENTRADAS; x++) {
                distanciaEuclidiana += Math.pow(entradas[x] -
                    pesosCamadaEscondida[neuronio][x], 2D);
154           }

156           distanciaEuclidiana = Math.sqrt(distanciaEuclidiana);

158           //Verifica qual a menor distancia euclidiana
              if (distanciaEuclidiana < menorDistanciaEuclidiana) {
160             menorDistanciaEuclidiana = distanciaEuclidiana;
                neuronioMenorDistanciaEuclidiana = neuronio;
162           }
            }
164
            //Adiciona amostra no grupo omega do neuronio com menor
                distancia euclidiana
166         if (adicionarNoGrupoOmega(indiceAmostra,
                neuronioMenorDistanciaEuclidiana, gruposOmega)) {
              mudouGrupoOmega = true;
168         }
          }
170
          //Ajustando os pesos da camada escondida
172       for (int neuronio = 0; neuronio < NUM_NEU_CAMADA_ESCONDIDA;
              neuronio++) {
            for (int peso = 0; peso < NUM_ENTRADAS; peso++) {
174           valorParcial = 0D;

176           for (int indice = 0; indice < gruposOmega[neuronio].
                  size(); indice++) {
                recuperarEntradas(arquivoTreinamento.getLinhaNumero((
                    int) gruposOmega[neuronio].get(indice)));
178
                valorParcial += entradas[peso];
180           }

182           if (!gruposOmega[neuronio].isEmpty()) {//Evita divisao
                  por zero
                pesosCamadaEscondida[neuronio][peso] = (1D / (double)
```

```
                    gruposOmega[neuronio].size()) * valorParcial;
184             }
              }
186           }
        } while (mudouGrupoOmega);
188
        //Calculando variancia de cada funcao de ativacao
190     for (int neuronio = 0; neuronio < NUM_NEU_CAMADA_ESCONDIDA;
           neuronio++) {
          valorParcial = 0D;
192
          for (int amostra = 0; amostra < gruposOmega[neuronio].size
             (); amostra++) {
194         recuperarEntradas(arquivoTreinamento.getLinhaNumero((int)
              gruposOmega[neuronio].get(amostra)));

196           for (int i = 0; i < NUM_ENTRADAS; i++) {
              valorParcial += Math.pow(entradas[i] -
                 pesosCamadaEscondida[neuronio][i], 2D);
198           }
          }
200
          variancia[neuronio] = (1D / (double) gruposOmega[neuronio].
             size()) * valorParcial;
202     }

204     Comunicador.addLog("Fim do treinamento da camada escondida");
      } catch (Exception e) {
206     return false;
      }
208
      return true;
210   }

212   private boolean treinarCamadaSaida(Arquivo arquivoTreinamento) {
        Random random;
214     List<double[]> listaZ;
        int numAmostra;
216     double erroAtual;
        double erroAnterior;
218     double gradiente;
```

```
220     numEpocas = 0;
        numAmostra = 0;
222     listaZ = new ArrayList<double[]>();

224     //Iniciando pesos da camada de saida com valores aleatorios
           enttre 0 e 1
        random = new Random();
226
        for (int i = 0; i < NUM_NEU_CAMADA_SAIDA; i++) {
228       for (int j = 0; j < NUM_NEU_CAMADA_ESCONDIDA + 1; j++) {
            pesosCamadaSaida[i][j] = random.nextDouble();
230       }
        }
232
        //Calculando os valores dos vetores Z
234     for (String linha : arquivoTreinamento.getLinhasArquivo()) {
          numAmostra++;
236
          //Calculando saida da camada escondida
238       recuperarEntradas(linha);
          calcularSaida();
240       listaZ.add(new double[NUM_NEU_CAMADA_ESCONDIDA]);

242       for (int i = 0; i < NUM_NEU_CAMADA_ESCONDIDA; i++) {
            listaZ.get(numAmostra - 1)[i] = saidaCamadaEscondida[i];
244       }
        }
246
        //Para cada amostra do conjunto de treinamento calcula o
           gradiente e atualiza os pesos da camada de saida
248     erroAtual = erroQuadraticoMedio(arquivoTreinamento);
        do {
250       erroAnterior = erroAtual;

252       for (String linha : arquivoTreinamento.getLinhasArquivo()) {
            recuperarEntradas(linha);
254
            calcularSaida();
256
            //Atualizando pesos sinapticos da camada de saida em funcao
                do valor do gradiente
258         for (int neuronio = 0; neuronio < NUM_NEU_CAMADA_SAIDA;
```

```
            neuronio++) {
            gradiente = saidasEsperadas[neuronio] - saidaCamadaSaida[
               neuronio];

            for (int peso = 0; peso < NUM_NEU_CAMADA_ESCONDIDA + 1;
               peso++) {
              pesosCamadaSaida[neuronio][peso] += TAXA_APRENDIZADO *
                 gradiente * saidaCamadaEscondida[peso];
            }
          }
        }

        erroAtual = erroQuadraticoMedio(arquivoTreinamento);
        numEpocas++;
        Comunicador.addLog(String.format("%d %s", numEpocas, Double.
           toString(erroAtual)));
      } while (Math.abs(erroAtual - erroAnterior) > PRECISAO);

      return true;
    }

    private void calcularSaida() {
      double valorParcial;

      //Calculando saida da camada escondida
      saidaCamadaEscondida[0] = LIMIAR_ATIVACAO;
      for (int neuronio = 0; neuronio < NUM_NEU_CAMADA_ESCONDIDA;
         neuronio++) {
        valorParcial = 0D;

        for (int entrada = 0; entrada < NUM_ENTRADAS; entrada++) {
          valorParcial += Math.pow(entradas[entrada] -
             pesosCamadaEscondida[neuronio][entrada], 2D);
        }

        saidaCamadaEscondida[neuronio + 1] = Math.pow(Math.E, (-1D) *
           (valorParcial / (2D * variancia[neuronio])));
      }

      //Calculando saida da camada de saida
      for (int neuronio = 0; neuronio < NUM_NEU_CAMADA_SAIDA;
         neuronio++) {
```

```
292         valorParcial = 0D;

294         for (int i = 0; i < NUM_NEU_CAMADA_ESCONDIDA + 1; i++) {
              valorParcial += pesosCamadaSaida[neuronio][i] *
                 saidaCamadaEscondida[i];
296         }

298         saidaCamadaSaida[neuronio] = valorParcial;
        }
300
    }
302
    private void recuperarEntradas(String linha) {
304     String[] vetor;
        int i;
306
        vetor = linha.split("\\s+");
308     i = 0;

310     if (vetor[0].equals("")) {
          i++;
312     }

314     //preenche o vetor de entradas a partir da linha lida do
           arquivo
        for (int j = 0; j < NUM_ENTRADAS; j++) {
316       entradas[j] = Numero.parseDouble(vetor[i++]);
        }
318
        //preenche o vetor das saidas esperadas a partir da linha lida
           do arquivo
320     if (vetor.length > i) {
          for (int j = 0; i + j < vetor.length; j++) {
322         saidasEsperadas[j] = Numero.parseDouble(vetor[i++]);
          }
324     }
    }
326
    private boolean adicionarNoGrupoOmega(int amostra, int neuronio,
       List[] gruposOmega) {
328     //Se a amostra ja esta no grupo nao ha  mudanca
        for (int indice = 0; indice < gruposOmega[neuronio].size();
```

```
          indice++) {
330         if ((int) gruposOmega[neuronio].get(indice) == amostra) {
              return false;
332         }
        }
334
        //Se a amostra esta em outro grupo deve ser removida
336     for (int i = 0; i < NUM_NEU_CAMADA_ESCONDIDA; i++) {
          //Nao olha no grupo onde deve ser adicionado
338       if (i == neuronio) {
            continue;
340       }

342       //remove a amostra do grupo antigo
          for (int indice = 0; indice < gruposOmega[i].size(); indice
             ++) {
344         if ((int) gruposOmega[i].get(indice) == amostra) {
              gruposOmega[i].remove(indice);
346         }
          }
348     }

350     //Adiciona amostra no grupo certo
        gruposOmega[neuronio].add(amostra);
352
        return true;
354   }

356   private double erroQuadraticoMedio(Arquivo arquivoTreinamento) {
        double erro;
358     double erroQuadMedio;
        int numAmostras;
360
        numAmostras = 0;
362     erro = 0D;
        erroQuadMedio = 0D;
364
        //Para cada amostra do conjunto de treinamento
366     for (String linha : arquivoTreinamento.getLinhasArquivo()) {
          numAmostras++;
368       recuperarEntradas(linha);
          calcularSaida();
```

```
370         erro = 0D;

372         //0 Erro e a soma dos erros de cada neuronio da camada de
               saida
            for (int neuronio = 0; neuronio < NUM_NEU_CAMADA_SAIDA;
               neuronio++) {
374           erro += Math.pow(saidasEsperadas[neuronio] -
                 saidaCamadaSaida[neuronio], 2D);
            }
376
            erro = erro / 2D;
378         erroQuadMedio += erro;
          }
380
          //Calculando Eqm
382       erroQuadMedio = erroQuadMedio / numAmostras;

384       return erroQuadMedio;
        }
386
        //Imprime os pesos sinapticos da camada escondida
388     private void imprimirPesosCamadaEscondida() {
          String texto;
390
          Comunicador.addLog("Pesos camada escondida:");
392
          for (int neuronio = 0; neuronio < NUM_NEU_CAMADA_ESCONDIDA;
             neuronio++) {
394         texto = "";

396         for (int i = 0; i < NUM_ENTRADAS; i++) {
              if (i > 0) {
398             texto += ", ";
              }
400
              texto += String.format("%s", Double.toString(
                 pesosCamadaEscondida[neuronio][i]));
402         }

404         Comunicador.addLog(String.format("  N%d: centro = (%s);
               variancia = %s", (neuronio + 1), texto, Double.toString(
               variancia[neuronio])));
```

```
      }
406   }

408   //Imprime os pesos sinapticos da camada de saida
      private void imprimirPesosCamadaSaida() {
410     String texto;

412     Comunicador.addLog("Pesos camada de saida");

414     for (int neuronio = 0; neuronio < NUM_NEU_CAMADA_SAIDA;
           neuronio++) {
        texto = "";
416
        for (int i = 0; i < NUM_NEU_CAMADA_ESCONDIDA + 1; i++) {
418       texto += String.format("%s ", Double.toString(
            pesosCamadaSaida[neuronio][i]));
        }
420
        Comunicador.addLog(String.format("N%d: %s", neuronio + 1,
           texto));
422     }
      }
424
      //Realiza o pos processamento
426   private double posProcessamento(double val) {
        if (val >= 0D) {
428       return 1D;
        }
430
        return -1D;
432   }

434 }
```

ANEXO A - CONJUNTO DE TREINAMENTO

x1	x2	d
0.2563	0.9503	-1.0000
0.2405	0.9018	-1.0000
0.1157	0.3676	1.0000
0.5147	0.0167	1.0000
0.4127	0.3275	1.0000
0.2809	0.5830	1.0000
0.8263	0.9301	-1.0000
0.9359	0.8724	-1.0000
0.1096	0.9165	-1.0000
0.5158	0.8545	-1.0000
0.1334	0.1362	1.0000
0.6371	0.1439	1.0000
0.7052	0.6277	-1.0000
0.8703	0.8666	-1.0000
0.2612	0.6109	1.0000
0.0244	0.5279	1.0000
0.9588	0.3672	-1.0000
0.9332	0.5499	-1.0000
0.9623	0.2961	-1.0000
0.7297	0.5776	-1.0000
0.4560	0.1871	1.0000
0.1715	0.7713	1.0000
0.5571	0.5485	-1.0000
0.3344	0.0259	1.0000
0.4803	0.7635	-1.0000
0.9721	0.4850	-1.0000
0.8318	0.7844	-1.0000
0.1373	0.0292	1.0000
0.3660	0.8581	-1.0000
0.3626	0.7302	-1.0000
0.6474	0.3324	1.0000
0.3461	0.2398	1.0000
0.1353	0.8120	1.0000
0.3463	0.1017	1.0000
0.9086	0.1947	-1.0000
0.5227	0.2321	1.0000

```
0.5153    0.2041    1.0000
0.1832    0.0661    1.0000
0.5015    0.9812   -1.0000
0.5024    0.5274   -1.0000
```

ANEXO B - CONJUNTO DE TESTE

x1	x2	d
0.8705	0.9329	-1
0.0388	0.2703	1
0.8236	0.4458	-1
0.7075	0.1502	1
0.9587	0.8663	-1
0.6115	0.9365	-1
0.3534	0.3646	1
0.3268	0.2766	1
0.6129	0.4518	-1
0.9948	0.4962	-1