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

JÔNATAS TRABUCO BELOTTI

IMPLEMENTAÇÃO REDE NEURAL *PERCEPTRON* MULTICAMADAS

RELATÓRIO

PONTA GROSSA
2017

JÔNATAS TRABUCO BELOTTI

IMPLEMENTAÇÃO REDE NEURAL *PERCEPTRON* MULTICAMADAS

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 **3**
1.1 ESTUDO DE CASO 4
2 DESENVOLVIMENTO ESTUDO DE CASO **5**
2.1 IMPLEMENTAÇÃO 5
2.1.1 Inicialização dos pesos sinápticos 5
2.1.2 Cálculo do erro quadrático médio 7
2.1.3 Função de ativação 10
2.1.4 Treinamento 11
2.1.5 Teste 15
2.2 EXECUÇÃO DO TREINAMENTO 18
2.3 EXECUÇÃO DO TESTE 19
3 CONCLUSÃO **22**
REFERÊNCIAS **23**
APÊNDICE A - IMPLEMENTAÇÃO DA CLASSE MLP **24**
ANEXO A - CONJUNTO DE TREINAMENTO **33**
ANEXO B - CONJUNTO DE TESTE **39**

1 INTRODUÇÃO

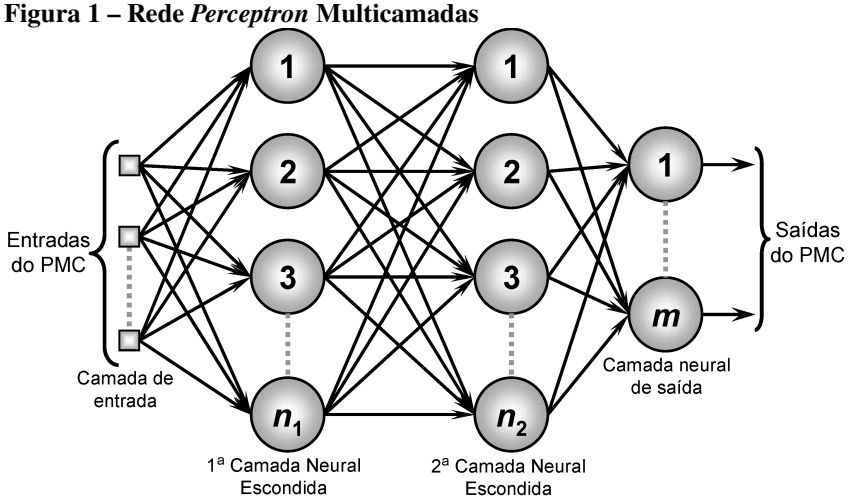
O *Perceptron* de Múltiplas Camadas (PMC, ou MLP do inglês *Multilayer Perceptron*) é constituído por um conjunto de neurônios artificiais dispostos em varias camadas, de modo que o sinal de entrada se propaga para frente através da rede, camada por camada. Dentre suas camadas existe a camada de entrada que recebe os sinais de entrada, a camada de saída que entrega o resultado obtido pela rede e no meio dessas podem existir quantas camadas forem necessárias. Essas camadas são chamadas intermediárias ou ocultas (HAYKIN, 2001). Note que uma MLP é constituida de pelo menos 2 camadas neurais, sendo uma camada de saída e pelo menos 1 camada escondida (SILVA; SPATTI; FLAUZINO, 2010).

As MLPs são consideradas uma das arquiteturas mais versáteis quanto a aplicabilidade, sendo utilizadas em diversas áreas do conhecimento. Dentre as utilizações da MLP estão: aproximação universal de funções, reconhecimento de padrões, identificação e controle de processos, previsão de series temporais e otimização de sistemas (SILVA; SPATTI; FLAUZINO, 2010).

Silva, Spatti e Flauzino (2010) atribuem o início da grande popularidade e das extensas aplicabilidades das MLPs ao final da década de 1980, em virtude da publicação do livro *Parallel distributed processing* de Rumelhart *et al.* (1987). Nesse livro os autores explicitaram o algoritmo de aprendizagem *backpropagation*, permitindo assim sua utilização no treinamento das MLPs (SILVA; SPATTI; FLAUZINO, 2010).

A Figura 1 mostra a arquitetura da rede *Perceptron* Multicamadas.

Figura 1 – Rede *Perceptron* Multicamadas



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

Na Figura 1 é possível ver a camada de entrada, composta pelos sinais de entrada $\{x_1, x_2, x_3, \ldots, x_n\}$, a primeira camada neural escondida da rede composta pelos neurônios $\{1, 2, 3, \ldots, n_1\}$, a segunda camada neural escondida da rede composta pelos neurônios $\{1, 2, 3, \ldots, n_2\}$ e por fim a camada neural de saída composta pelos neurônios $\{1, 2, 3, \ldots, m\}$. Note que a primeira camada da rede, a camada de entrada, não possui neurônios, apenas os sinais de entrada, os neurônios da MLP estão localizados nas camadas escondidas e na camada de saída.

Este relatório tem como objetivo descrever o desenvolvimento do projeto prático 5.8 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 do tipo *Perceptron* Multicamadas para ser usada como um aproximador universal de funções na confecção de um processador de imagens de ressonância magnética.

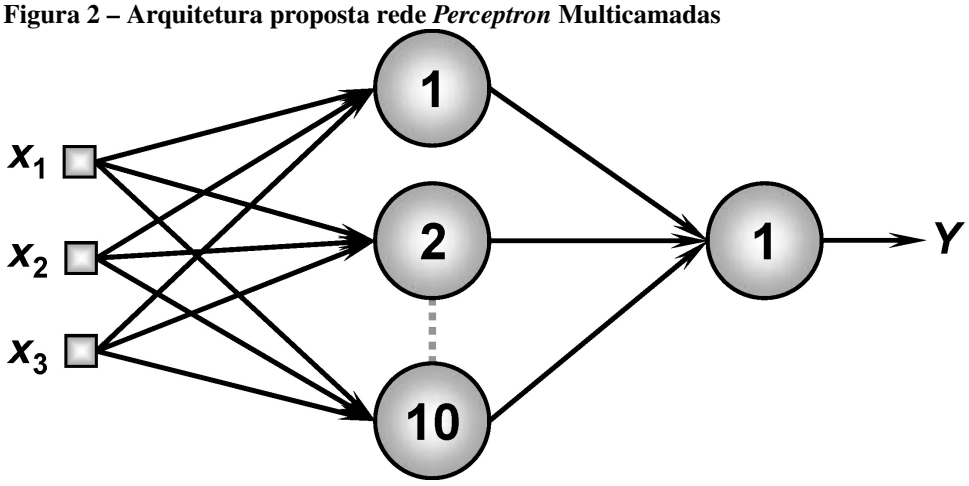
1.1 ESTUDO DE CASO

O projeto prático 5.8 do livro Redes neurais artificias para engenharia e ciências aplicadas de Silva, Spatti e Flauzino (2010), propõe o desenvolvimento de uma rede neural *Perceptron* Multicamadas para ser usada como um aproximador universal de funções na confecção de um processador de imagens de ressonância magnética. A partir da análise de 3 grandezas representadas por $\{x_1, x_2$ e $x_3\}$ o processador de imagens deve estimar a energia absorvida do sistema, denotada por y.

O livro disponibiliza os conjuntos de treinamento e de teste para a implementação da rede, aqui apresentados nos Anexos A e B respectivamente.

Também é descrita a arquitetura que a rede deve ter, a rede deve possuir 3 camadas, uma camada de entrada contendo 3 sinais de entrada $\{x_1, x_2$ e $x_3\}$, uma camada escondida contendo 10 neurônios e a camada de saída com apenas 1 neurônio. A Figura 2 mostra detalhadamente a arquitetura que a MLP deve ter para a realização do Projeto prático.

Figura 2 – Arquitetura proposta rede *Perceptron* Multicamadas



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

Para o treinamento da MLP é especificado a utilização do algoritmo *backpropagation*.

2 DESENVOLVIMENTO ESTUDO DE CASO

Esse capítulo mostra o desenvolvimento do estudo de caso proposto na Seção 1.1, apresentando a implementação da rede na Seção 2.1, a execução do treinamento na Seção 2.2 e a execução do teste da rede na Seção 2.3.

2.1 IMPLEMENTAÇÃO

A rede neural *Perceptron* Multicamadas foi implementada na linguagem Java, visando facilitar o acesso ao código fonte, o mesmo, juntamente com um arquivo contendo o conjunto de dados de treinamentoe um arquivo com o conjunto de dados de teste foram disponibilizados em um repositório do *GitHub* que pode ser acessado pelo link <https://github.com/jonatastbelotti/redePerceptronMultiplasCamadas>.

A rede neural foi implementada através da classe MLP, que é apresentada no Apêndice A. As Seções 2.1.1, 2.1.2, 2.1.3, 2.1.4 e 2.1.5 trazem a explicação detalhada dos principais métodos da classe.

2.1.1 Inicialização dos pesos sinápticos

Como definido no livro, os pesos sinápticos de todos os neurônios de todas as camadas foram gerados de forma aleatória com valores entre 0 e 1. A inicialização dos pesos sinápticos foi implementada no método construtor da classe MLP. O Código 1 apresenta a implementação do método construtor da classe MLP.

Código 1 – Construtor da classe MLP

```
1  public class MLP {
2
3    public static final int NUM_SINAIS_ENTRADA = 3;
4    private final double TAXA_APRENDIZAGEM = 0.1;
5    private final double PRECISAO = 0.000001;
6    private final double BETA = 1.0;
7    private final int NUM_NEU_CAMADA_ESCONDIDA = 10;
8    private final int NUM_NEU_CAMADA_SAIDA = 1;
9
10   private int numEpocas;
11   private double[] entradas;
12   private double[][] pesosCamadaEscondida;
```

```
  private double[] pesosCamadaSaida;
14  private double[] potencialCamadaEscondida;
  private double[] saidaCamadaEscondida;
16  private double saida;

18  public MLP() {
    Random random;
20
    entradas = new double[NUM_SINAIS_ENTRADA + 1];
22    pesosCamadaEscondida = new double[NUM_NEU_CAMADA_ESCONDIDA][
       NUM_SINAIS_ENTRADA + 1];
    pesosCamadaSaida = new double[NUM_NEU_CAMADA_ESCONDIDA + 1];
24    saidaCamadaEscondida = new double[NUM_NEU_CAMADA_ESCONDIDA +
       1];
    potencialCamadaEscondida = new double[NUM_NEU_CAMADA_ESCONDIDA
       + 1];
26
    //Iniciando os pesos sinpticos
28    random = new Random();
    for (int i = 0; i < NUM_NEU_CAMADA_ESCONDIDA; i++) {
30      for (int j = 0; j < NUM_SINAIS_ENTRADA + 1; j++) {
        pesosCamadaEscondida[i][j] = random.nextDouble();
32      }

34      pesosCamadaSaida[i] = random.nextDouble();
    }
36    pesosCamadaSaida[NUM_NEU_CAMADA_ESCONDIDA] = random.nextDouble
       ();
  }
```

Fonte: Autoria própria.

Analisando o Código 1 verifica-se que na linha 28 o gerador de números pseudoaleatórios é iniciado[1], os laços das linhas 29 e 30 fazem com que todos os pesos sinápticos de todos os neurônios da camada escondida sejam iniciados pelo método *random.nextDouble()*, que gera números pseudoaleatórios entre 0 e 1. Por fim, na linha 34 os pesos sinápticos da camada de saída são gerados também pelo método *random.nextDouble()*.

[1] Em Java não é necessário definir a semente para o gerador de números pseudoaleatórios, pois caso uma semente não seja informada ele assume que você deseja utilizar a hora atual como semente.

2.1.2 Cálculo do erro quadrático médio

Para cada saída dada pela MLP é necessário dizer o quão próximo da saída desejada a rede está, com esse objetivo a Equação 2.1 apresenta a formula para calcular o erro quadrático da rede para uma entrada k.

$$E(k) = \frac{1}{2} \sum_{j=1}^{m} (d_j(k) - \mathbf{y_j})^2 \tag{2.1}$$

onde $E(k)$ é o erro quadrático para a amostra k, m é o número de neurônios da camada de saída, $d_j(k)$ é a saída desejada do neurônio j da camada de saída para a amostra k e $\mathbf{y_j}$ é a saída obtida pelo neurônio j da camada de saída para a amostra k. Percebe-se que o erro quadrático da MLP para uma amostra k é a soma dos erros quadráticos de todos os neurônios da camada de saída para a amostra k.

Por sua vez, ao final de cada época de treinamento deseja-se saber o erro quadrático da rede para com todo o conjunto de treinamento, tal objetivo é obtido através da Equação 2.2, que apresenta a formula para calcular o erro quadrático médio da MLP para um conjunto de treinamento com p amostras.

$$\mathrm{E_{qm}} = \frac{1}{p} \sum_{k=1}^{p} E(k) \tag{2.2}$$

onde $\mathrm{E_{qm}}$ é o erro quadrático médio da rede, p é o tamanho do conjunto de amostras e $E(k)$ é o erro quadrático da rede para a amostra k calculado pela Equação 2.1.

Perceba que como a arquitetura da rede Perceptron Multicamadas proposta tem apenas 1 neurônio na camada de saída a Equação 2.1 pode ser simplificada para a Equação 2.3.

$$E(k) = \frac{(d(k) - y(k))^2}{2} \tag{2.3}$$

onde $E(k)$ é o erro quadrático da rede para a amostra k, $d(k)$ é a saída esperada da rede para a amostra k e $y(k)$ é a saída da rede para a amostra k.

Na implementação em Java, esse cálculo é realizado pelo método erroQuadraticoMedio, método este que é apresentado no Código 2.

Código 2 – Método erroQuadraticoMedio da classe MLP

```
1  private double erroQuadraticoMedio(Arquivo arquivo) {
2    FileReader arq;
     BufferedReader lerArq;
4    String linha;
     String[] vetor;
6    int i;
```

```
    int numAmostras;
    double saidaEsperada;
    double valorParcial;
    double erro;

    erro = 0D;
    numAmostras = 0;

    try {
      arq = new FileReader(arquivo.getCaminhoCompleto());
      lerArq = new BufferedReader(arq);

      linha = lerArq.readLine();
      if (linha.contains("x1")) {
        linha = lerArq.readLine();
      }

      while (linha != null) {
        vetor = linha.split("\\s+");
        i = 0;

        if (vetor[0].equals("")) {
          i = 1;
        }

        numAmostras++;
        entradas[0] = -1.0;
        entradas[1] = Double.parseDouble(vetor[i++].replace(",", ".
           "));
        entradas[2] = Double.parseDouble(vetor[i++].replace(",", ".
           "));
        entradas[3] = Double.parseDouble(vetor[i++].replace(",", ".
           "));
        saidaEsperada = Double.parseDouble(vetor[i].replace(",", ".
           "));

        //Calculando saidas da camada escondida
        saidaCamadaEscondida[0] = potencialCamadaEscondida[0] = -1D
           ;
        for (i = 1; i < saidaCamadaEscondida.length; i++) {
          valorParcial = 0D;
```

```
44            for (int j = 0; j < NUM_SINAIS_ENTRADA + 1; j++) {
                valorParcial += entradas[j] * pesosCamadaEscondida[i -
                    1][j];
46            }

48            potencialCamadaEscondida[i] = valorParcial;
              saidaCamadaEscondida[i] = funcaoLogistica(valorParcial);
50          }

52          //Calculando saida da camada de saída
            valorParcial = -1D * pesosCamadaSaida[0];
54          for (i = 1; i < NUM_NEU_CAMADA_ESCONDIDA + 1; i++) {
              valorParcial += saidaCamadaEscondida[i - 1] *
                  pesosCamadaSaida[i];
56          }
            saida = funcaoLogistica(valorParcial);
58
            //Calculando erro
60          erro = erro + (Math.pow((saidaEsperada - this.saida), 2D) /
                2D);

62          linha = lerArq.readLine();
          }
64
          arq.close();
66        erro = erro / (double) numAmostras;

68      } catch (FileNotFoundException ex) {
        } catch (IOException ex) {
70      }

72      return erro;
    }
```

Fonte: Autoria própria.

Perceba que o laço da linha 24 percorre todas as entradas do conjunto de treinamento. Para cada entrada é calculada a saída da rede (linha 57) e o seu respectivo erro quadrático em comparação com a saída desejada (linha 60), note que na implementação foi utilizada a Equação 2.3 para o cálculo do erro quadrático. Por fim, depois de percorrer todos as amostras de treinamento, na linha 66 e erro quadrático médio é calculado de acordo com a Equação 2.2.

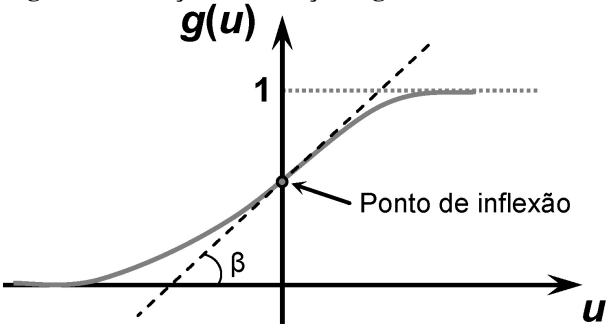
2.1.3 Função de ativação

O livro define que a função de ativação usada em todos os neurônios de todos as camadas da MLP deve ser a Função logística, função esta que assume sempre valores entre 0 e 1 dada pela Equação 2.4.

$$g(u) = \frac{1}{1 + e^{-\beta u}} \tag{2.4}$$

onde $g(u)$ é a função de ativação, e é a constante de *Euler*, β é uma constante real associada ao valor da inclinação da função logística frente ao seu ponto de inflexão e u é o potencial de ativação do neurônio. O efeito da constante β fica evidente ao analisarmos o gráfico da Figura 3, que apresenta o comportamento da Função logística.

Figura 3 – Função de ativação logística



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

Note que na Figura 3 β é o angulo de inclinação da função logística frente ao seu ponto de inflexão.

O Código 3 apresenta a implementação em Java da função logística descrita pela Equação 2.4 através do método funcaoLogistica.

Código 3 – Método funcaoLogistica da classe MLP

```
1  private double funcaoLogistica(double valor) {
2    return 1D / (1D + Math.pow(Math.E, -1D * BETA * valor));
   }
```

Fonte: Autoria própria.

A função logística é uma função totalmente diferenciável, ou seja, sua derivada de primeira ordem existe e é conhecida em todos os pontos de seu domínio de definição (SILVA; SPATTI; FLAUZINO, 2010). Comprovando esse fato a Equação 2.5 apresenta a derivada de primeira ordem da função logística.

$$g'(u) = \frac{\beta e^{-\beta u}}{(e^{-\beta u} + 1)^2} \tag{2.5}$$

A derivada da função logística também foi implementada em Java, através do método funcaoLogisticaDerivada apresentado no Código 4.

Código 4 – Método funcaoLogisticaDerivada da classe MLP

```
1  private double funcaoLogisticaDerivada(double valor) {
2    return (BETA * Math.pow(Math.E, -1D * BETA * valor)) / Math.pow
       ((Math.pow(Math.E, -1D * BETA * valor) + 1D), 2D);
   }
```

Fonte: Autoria própria.

Como pode ser visto na linha 6 do Código 1, na implementação da MLP foi utilizado o valor 1 para a constante β.

2.1.4 Treinamento

O treinamento da MLP é realizado pelo algoritmo de aprendizagem *backpropagation*. Para o projeto prático foram definidas uma taxa de aprendizagem $\eta = 0,1$ e precisão $\varepsilon = 10^{-6}$, como podem ser vistas nas linhas 4 e 5 do Código 1. O Código 5 apresenta o método treinar da classe MLP que implementa o *backpropagation* na rede.

Código 5 – Método treinar da classe MLP

```
1  public boolean treinar(Arquivo arquivoTreinamento) {
2    FileReader arq;
     BufferedReader lerArq;
4    String linha;
     String[] vetor;
6    int i;
     double saidaEsperada;
8    double erroAtual;
     double erroAnterior;
10   double valorParcial;
     double gradienteCamadaSaida;
12   double gradienteCamadaEscondida;

14   numEpocas = 0;
     erroAtual = erroQuadraticoMedio(arquivoTreinamento);
16
     Comunicador.iniciarLog("Início treinamento da MLP");
18   Comunicador.addLog(String.format("Erro inicial: %.10f",
       erroAtual).replace(".", ","));
     Comunicador.addLog("Época Eqm");
```

```
20
    try {
22      do {
          this.numEpocas++;
24        erroAnterior = erroAtual;
          arq = new FileReader(arquivoTreinamento.getCaminhoCompleto
             ());
26        lerArq = new BufferedReader(arq);

28        linha = lerArq.readLine();
          if (linha.contains("x1")) {
30          linha = lerArq.readLine();
          }
32
          while (linha != null) {
34          vetor = linha.split("\\s+");
            i = 0;
36
            if (vetor[0].equals("")) {
38            i = 1;
            }
40
            entradas[0] = -1.0;
42          entradas[1] = Double.parseDouble(vetor[i++].replace(",",
               "."));
            entradas[2] = Double.parseDouble(vetor[i++].replace(",",
               "."));
44          entradas[3] = Double.parseDouble(vetor[i++].replace(",",
               "."));
            saidaEsperada = Double.parseDouble(vetor[i].replace(",",
               "."));
46
            //Calculando saidas da camada escondida
48          saidaCamadaEscondida[0] = potencialCamadaEscondida[0] =
               -1D;
            for (i = 1; i < saidaCamadaEscondida.length; i++) {
50            valorParcial = 0D;

52            for (int j = 0; j < NUM_SINAIS_ENTRADA + 1; j++) {
                valorParcial += entradas[j] * pesosCamadaEscondida[i
                   - 1][j];
54            }
```

```
56            potencialCamadaEscondida[i] = valorParcial;
              saidaCamadaEscondida[i] = funcaoLogistica(valorParcial)
                 ;
58          }

60          //Calculando saida da camada de saída
            valorParcial = -1D * pesosCamadaSaida[0];
62          for (i = 1; i < NUM_NEU_CAMADA_ESCONDIDA + 1; i++) {
              valorParcial += saidaCamadaEscondida[i - 1] *
                 pesosCamadaSaida[i];
64          }
            saida = funcaoLogistica(valorParcial);
66
            //Ajustando pesos sinapticos da camada de saida
68          gradienteCamadaSaida = (saidaEsperada - saida) *
               funcaoLogisticaDerivada(valorParcial);

70          for (i = 0; i < pesosCamadaSaida.length; i++) {
              pesosCamadaSaida[i] = pesosCamadaSaida[i] + (
                 TAXA_APRENDIZAGEM * gradienteCamadaSaida *
                 saidaCamadaEscondida[i]);
72          }

74          //Ajustando pesos sinapticos da camada escondida
            for (i = 0; i < NUM_NEU_CAMADA_ESCONDIDA; i++) {
76            gradienteCamadaEscondida = gradienteCamadaSaida *
                 pesosCamadaSaida[i + 1] * funcaoLogisticaDerivada(
                 potencialCamadaEscondida[i + 1]);

78            for (int j = 0; j < NUM_SINAIS_ENTRADA + 1; j++) {
                pesosCamadaEscondida[i][j] = pesosCamadaEscondida[i][
                   j] + (TAXA_APRENDIZAGEM * gradienteCamadaEscondida
                    * entradas[j]);
80            }
            }
82
              linha = lerArq.readLine();
84          }

86          arq.close();
            erroAtual = erroQuadraticoMedio(arquivoTreinamento);
```

```
88          Comunicador.addLog(String.format("%d    %.10f", numEpocas,
                erroAtual).replace(".", ","));
        } while (Math.abs(erroAtual - erroAnterior) > PRECISAO &&
            numEpocas < 10000);
90
        Comunicador.addLog("Fim do treinamento.");
92      } catch (FileNotFoundException ex) {
        return false;
94      } catch (IOException ex) {
        return false;
96      }

98      return true;
    }
```

Fonte: Autoria própria.

O método apresentado no Código 5 recebe como parâmetro um arquivo contendo todas as amostras do conjunto de treinamento. Como o algoritmo *backpropagation* realiza o treinamento supervisionado cada amostra do conjunto de treinamento deve estar acompanhada da sua respectiva resposta.

A condição da linha 89 faz com o laço da linha 22 seja executado enquanto $|E_{qm}^{Atual} - E_{qm}^{Anterior}|$ seja maior que a precisão e o número de épocas do treinamento seja menor que 10.000. Cada iteração desse laço corresponde a uma época no treinamento da MLP.

Por sua vez, o laço da linha 33 passa por todas as amostras do conjunto de treinamento. Entre as linhas 41 e 44 são obtidos os valores do vetor de entradas para a amostra de treinamento k, $\mathbf{x}(k)$ (no Código 5 chamado de entradas), sendo que o primeiro valor de entrada corresponde ao limiar de ativação, com valor -1. Na linha 45 é obtido o valor da saída esperada para a amostra.

De posse do vetor de entradas $\mathbf{x}(k)$, entre as linhas 48 e 58 são calculadas as saídas de todos os neurônios da camada escondida. Note que o método funcaoLogistica trata-se da função de ativação, anteriormente explicada na Seção 2.1.3. Após obter as saídas da camada escondida entre as linhas 61 e 65 é calculada a saída da camada de saída e com isso a saída da rede.

Com o fim da fase *forward*, inicia-se a fase *backward* para ajustar os pesos sinápticos da rede.

Na linha 68 é calculado o gradiente da camada de saída. Note que o método funcaoLogisticaDerivada implementa a derivada de primeira ordem da função de ativação, anteriormente explicada na Seção 2.1.3. De posse do valor do gradiente, entre as linhas 70 e 72 os pesos sinápticos do neurônio da camada de saída são atualizados.

Os laços das linhas 75 e 78 calculam o gradiente de cada neurônio da camada oculta e ajustam os pesos sinápticos de cada neurônio da camada oculta respectivamente. Finalizando

assim a fase *backward*.

2.1.5 Teste

Após o treinamento da MLP é necessário testa-la a fim de validar o desempenho dos pesos sinápticos obtidos. O livro pede que no treinamento seja obtido o erro relativo (%) entre os valores desejados e os obtidos pela MLP. Tal valor é calculado através da Equação 2.6.

$$\text{Erro}(k) = \frac{100}{d(k)}.|d(k) - y(k)| \tag{2.6}$$

onde Erro(k) é o erro relativo da rede em relação a amostra k, $d(k)$ é a saída esperada para a amostra k e $y(k)$ é a saída da rede para a amostra k.

Também é pedido o Erro relativo médio (%) em relação a todas as amostras do conjunto de teste. Por sua vez o erro relativo médio é calculado pela Equação 2.7.

$$\text{Erro}_{\text{medio}} = \frac{\sum_{k=1}^{p} \text{Erro}(k)}{p} \tag{2.7}$$

onde $\text{Erro}_{\text{medio}}$ é o erro relativo médio (%) da rede em relação a todas as amostras do conjunto de teste, p é o tamanho do conjunto de teste e Erro(k) é o erro relativo da rede em relação a amostra k do conjunto de teste calculado pela Equação 2.6.

Além do erro relativo médio, o livro pede a variância do erro relativo, calculada pela Equação 2.8.

$$s(\text{Erro})^2 = \left(\sum_{i=1}^{p} \text{Erro}(i)^2 \right) - p.{\text{Erro}_{\text{medio}}}^2 \tag{2.8}$$

onde $s(\text{Erro})^2$ é variância do erro relativo para com todas as amostras do conjunto de teste, p é o tamanho do conjunto de teste, Erro(i) é o erro relativo da rede para a amostra i do conjunto de teste calculado pela Equação 2.6 e $\text{Erro}_{\text{medio}}$ é o erro relativo médio da rede para com todo o conjunto de teste calculado pela Equação 2.7.

O Código 6 apresenta a implementação do método testar, que recebe como parâmetro o arquivo contendo o conjunto de dados de teste e realiza o teste da rede para todo o conjunto de teste.

Código 6 – Método testar da classe MLP

```
1  public void testar(Arquivo arquivoTeste) {
2    FileReader arq;
     BufferedReader lerArq;
4    String linha;
     String[] vetor;
```

```
6       int i;
        int numAmostras;
8       double saidaEsperada;
        double valorParcial;
10      double erroRelativo;
        double variancia;
12
        Comunicador.iniciarLog("Início teste da MLP");
14      Comunicador.addLog("Resposta - Saída rede          Erro");
        erroRelativo = 0D;
16      variancia = 0D;
        numAmostras = 0;
18
        try {
20          arq = new FileReader(arquivoTeste.getCaminhoCompleto());
            lerArq = new BufferedReader(arq);
22
            linha = lerArq.readLine();
24          if (linha.contains("x1")) {
              linha = lerArq.readLine();
26          }

28          while (linha != null) {
              vetor = linha.split("\\s+");
30            i = 0;

32            if (vetor[0].equals("")) {
                i = 1;
34            }

36            numAmostras++;
              entradas[0] = -1.0;
38            entradas[1] = Double.parseDouble(vetor[i++].replace(",",
                 "."));
              entradas[2] = Double.parseDouble(vetor[i++].replace(",",
                 "."));
40            entradas[3] = Double.parseDouble(vetor[i++].replace(",",
                 "."));
              saidaEsperada = Double.parseDouble(vetor[i].replace(",",
                 "."));
42
              //Calculando saidas da camada escondida
```

```
44          saidaCamadaEscondida[0] = potencialCamadaEscondida[0] =
               -1D;
            for (i = 1; i < saidaCamadaEscondida.length; i++) {
46            valorParcial = 0D;

48            for (int j = 0; j < NUM_SINAIS_ENTRADA + 1; j++) {
                valorParcial += entradas[j] * pesosCamadaEscondida[i
                   - 1][j];
50            }

52            potencialCamadaEscondida[i] = valorParcial;
              saidaCamadaEscondida[i] = funcaoLogistica(valorParcial)
                 ;
54          }

56          //Calculando saida da camada de saída
            valorParcial = -1D * pesosCamadaSaida[0];
58          for (i = 1; i < NUM_NEU_CAMADA_ESCONDIDA + 1; i++) {
              valorParcial += saidaCamadaEscondida[i - 1] *
                 pesosCamadaSaida[i];
60          }
            saida = funcaoLogistica(valorParcial);
62
            //Porcentagem de erro em cada amostra
64          valorParcial = (100D / saidaEsperada) * (Math.abs(
               saidaEsperada - saida));
            erroRelativo += valorParcial;
66          variancia += Math.pow(valorParcial, 2D);
            Comunicador.addLog(String.format("%.4f          %.10f
               %.10f%%", saidaEsperada, saida, valorParcial));
68
            linha = lerArq.readLine();
70        }

72        //Calculando erro relativo médio
          erroRelativo = erroRelativo / (double) numAmostras;
74
          //Calculando variância
76        variancia = variancia - ((double)numAmostras * Math.pow(
             erroRelativo, 2D));
          variancia = variancia / ((double) (numAmostras - 1));
78        variancia = variancia * 100D;
```

```
80          Comunicador.addLog("Fim do teste");
            Comunicador.addLog(String.format("Erro relativo médio: %.10
               f%%", erroRelativo));
82          Comunicador.addLog(String.format("Variância: %.10f%%",
               variancia));

84          arq.close();
        } catch (FileNotFoundException ex) {
86      } catch (IOException ex) {
        }
88    }
```

Fonte: Autoria própria.

O laço da linha 28 passa por todas as amostras do conjunto de teste. Na linha 36 é calculado o tamanho do conjunto de teste. Enquanto entre as linhas 37 e 41 são obtidos o vetor de entradas da rede e a respectiva saída esperada.

Entre as linhas 44 e 54 são calculadas as saídas da camada escondida, em seguida, entre as linhas 57 e 61 é calculada a saída da camada de saída, que também é a saída da rede. De posse da saída da rede e da saída esperada, entre as linhas 64 e 78 são calculados média do erro relativo e a variância do erro relativo de acordo com as equações 2.7 e 2.8.

2.2 EXECUÇÃO DO TREINAMENTO

A MLP desenvolvida foi treinada 5 vezes para ter seu funcionamento validado. Nos 5 treinamentos foram utilizados taxa de aprendizado e precisão conforme descrito na Seção 2.1.4. Foi utilizado o conjunto de treinamento apresentado no Anexo A em todos os treinamentos, para facilitar a utilização dos dados de treinamento os mesmos já se encontram normalizados. A Tabela 1 apresenta os resultados obtidos em cada treinamento, nela é possível ver o valor do Erro quadrático médio obtido em cada treinamento juntamente com a quantidade de épocas necessárias para realizar cada treinamento.

Tabela 1 – Resultados do treinamento

Treinamento	Erro quadrático médio	Número total de épocas
1º(T1)	0,000772	139
2º(T2)	0,000774	115
3º(T3)	0,000779	96
4º(T4)	0,000766	118
5º(T5)	0,000791	98

Fonte: Autoria própria.

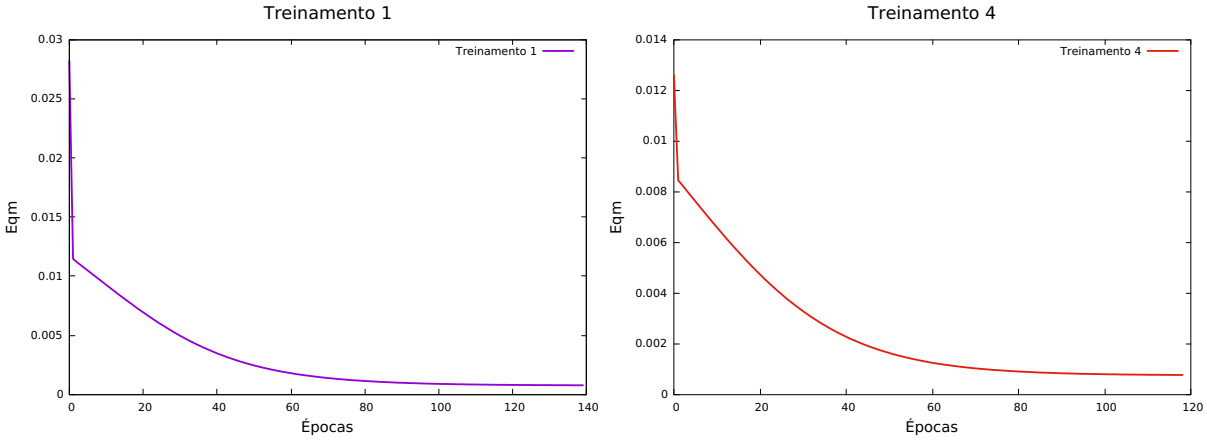
Para os 5 treinamentos os valores do erro quadrático médio ficaram próximos, tendo

uma média de $0,000774$, do mesmo modo o número total de épocas para o treinamento da rede também teve seus valores próximos, com média de 115 épocas.

Pelos dados da Tabela 1 verifica-se que tanto o erro quadrático médio como o número total de épocas variam de treinamento para treinamento. Isso se deve ao fato de que no início de cada treinamento os pesos sinápticos de todos os neurônios de todas as camadas são iniciados de forma aleatória, dessa forma o erro quadrático médio inicial da rede é sempre diferente em cada treinamento. Partindo de uma configuração diferente de pesos sinápticos a cada treinamento chega-se também a um erro quadrático médio e um número de épocas diferentes em cada treinamento.

Os gráficos da Figura 4 mostram a relação entre o erro quadrático médio e o número para os 2 treinamentos com o maior número de épocas, os treinamentos $T1$ e $T4$ com 139 e 118 épocas respectivamente.

Figura 4 – Erro quadrático médio em função do número de épocas



Fonte: Autoria própria.

Analisando os gráficos da Figura 4 nota-se que apesar do número de épocas dos 2 treinamentos serem diferentes, o comportamento das curvar é similar. As duas curvar apresentam uma queda, sendo que a velocidade do decaimento vai diminuindo ao passar das épocas, ao passo que quanto mais próximo do final do treinamento a velocidade de decaimento é quase inexistente.

2.3 EXECUÇÃO DO TESTE

Para cada um dos 5 treinamentos realizados, descritos na Seção 2.2, a MLP foi testada com o proposito de verificar sua capacidade de abstração. Tal teste foi realizado com a utilização de um conjunto de teste composto por 20 amostras, onde nenhuma das amostras presentes no conjunto de teste fizeram parte do conjunto de treinamento. Essas amostras de teste estão disponíveis na Tabela 2 e no Anexo B. Verifica-se na Tabela 2 que para cada amostra de teste existe uma saída esperada d associada.

Tabela 2 – Conjunto de amostras de teste

Amostra	$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$	$\mathbf{d}$
1	$0,0611$	$0,2860$	$0,7464$	$0,4831$
2	$0,5102$	$0,7464$	$0,0860$	$0,5965$
3	$0,0004$	$0,6916$	$0,5006$	$0,5318$
4	$0,9430$	$0,4476$	$0,2648$	$0,6843$
5	$0,1399$	$0,1610$	$0,2477$	$0,2872$
6	$0,6423$	$0,3229$	$0,8567$	$0,7663$
7	$0,6492$	$0,0007$	$0,6422$	$0,5666$
8	$0,1818$	$0,5078$	$0,9046$	$0,6601$
9	$0,7382$	$0,2647$	$0,1916$	$0,5427$
10	$0,3879$	$0,1307$	$0,8656$	$0,5836$
11	$0,1903$	$0,6523$	$0,7820$	$0,6950$
12	$0,8401$	$0,4490$	$0,2719$	$0,6790$
13	$0,0029$	$0,3264$	$0,2476$	$0,2956$
14	$0,7088$	$0,9342$	$0,2763$	$0,7742$
15	$0,1283$	$0,1882$	$0,7253$	$0,4662$
16	$0,8882$	$0,3077$	$0,8931$	$0,8093$
17	$0,2225$	$0,9182$	$0,7820$	$0,7581$
18	$0,1957$	$0,8423$	$0,3085$	$0,5826$
19	$0,9991$	$0,5914$	$0,3933$	$0,7938$
20	$0,2299$	$0,1524$	$0,7353$	$0,5012$

Fonte: Autoria própria.

Os resultados dos testes da MLP para cada um dos 5 treinamentos realizados juntamente com o erro relativo médio (%) de cada teste estão anotados na Tabela 3.

A análise dos porcentuais de erro relativo médio presentes na Tabela 3 possibilita dizer qual dos 5 treinamentos realizados da a MLP uma maior capacidade de generalização, basta escolher aquele com a menor porcentagem de erro relativo médio. Dessa forma, pelos dados da Tabela 3 a configuração da MLP obtida pelo treinamento T2 é a mais adequada para o sistema de ressonância magnética, pois a porcentagem de erro relativo médio do treinamento T2 é a menor dentro todos os treinamentos, com o valor de $1,22237\%$.

Tabela 3 – Resultado do teste

Amostra	d	y(T1)	y(T2)	y(T3)	y(T4)	y(T5)
1	$0,4831$	$0,490260$	$0,485166$	$0,487102$	$0,489197$	$0,481442$
2	$0,5965$	$0,588996$	$0,594787$	$0,593086$	$0,592241$	$0,594545$
3	$0,5318$	$0,529787$	$0,530082$	$0,528444$	$0,528944$	$0,525027$
4	$0,6843$	$0,701649$	$0,704101$	$0,705695$	$0,705233$	$0,708791$
5	$0,2872$	$0,292554$	$0,287820$	$0,285930$	$0,287969$	$0,285560$
6	$0,7663$	$0,754082$	$0,750790$	$0,751311$	$0,751529$	$0,751511$
7	$0,5666$	$0,569855$	$0,566283$	$0,569903$	$0,571175$	$0,568455$
8	$0,6601$	$0,681084$	$0,678641$	$0,678784$	$0,678543$	$0,676060$
9	$0,5427$	$0,530737$	$0,533233$	$0,533958$	$0,534265$	$0,535226$
10	$0,5836$	$0,605103$	$0,600425$	$0,603284$	$0,604649$	$0,600345$
11	$0,6950$	$0,692187$	$0,690735$	$0,690338$	$0,689436$	$0,687913$
12	$0,6790$	$0,670505$	$0,672669$	$0,674047$	$0,673580$	$0,676322$
13	$0,2956$	$0,301995$	$0,297581$	$0,294457$	$0,295789$	$0,293434$
14	$0,7742$	$0,776158$	$0,781257$	$0,780157$	$0,779966$	$0,782669$
15	$0,4662$	$0,470099$	$0,464016$	$0,466595$	$0,468982$	$0,460989$
16	$0,8093$	$0,819418$	$0,816288$	$0,816090$	$0,816700$	$0,817525$
17	$0,7581$	$0,781223$	$0,779609$	$0,777788$	$0,774931$	$0,775637$
18	$0,5826$	$0,591228$	$0,594273$	$0,592219$	$0,591322$	$0,591090$
19	$0,7938$	$0,792754$	$0,795181$	$0,795749$	$0,796320$	$0,799093$
20	$0,5012$	$0,499988$	$0,494637$	$0,497550$	$0,499782$	$0,492714$
Erro relativo médio (%)		$1,49092\%$	$1,22237\%$	$1,23276\%$	$1,23832\%$	$1,32608\%$
Variância (%)		$111,0003\%$	$102,894\%$	$103,328\%$	$100,693\%$	$82,198\%$

Fonte: Autoria própria.

3 CONCLUSÃO

Foi desenvolvida uma rede neural *Perceptron* multicamadas composta por 3 sinais de entrada, 1 camada escondida com 10 neurônios e 1 camada de saída com 1 neurônio para ser usada como aproximadora universal de funções em um processador de imagens de ressonância magnética. Dos 5 treinamentos realizados na rede a configuração de pesos sinápticos com o maior poder de generalização foi a resultante do treinamento T2, pois é a que apresentou a menor porcentagem de erro relativo médio em comparação com todos os treinamentos realizados, com $1,22237\%$ de erro relativo médio.

Conclui-se que a rede neural *Perceptron* multicamadas desenvolvida como aproximadora universal de funções no processador de imagens de ressonância magnética atenderá a todas as necessidades do processador de imagens, pois a mesma apresenta um excelente poder de generalização com porcentagem de erro de $1,22237\%$.

REFERÊNCIAS

HAYKIN, Simon. **Redes Neurais: principios e prática**. 2. ed. Porto Alegre: Bookman, 2001. ISBN 978-85-7307-718-6.

RUMELHART, David E *et al.* **Parallel distributed processing**. MIT press Cambridge, MA, USA:, 1987. v. 1. Disponível em: <http://www.cs.toronto.edu/~fritz/absps/pdp2.pdf>. Acesso em: 01 out. 2017.

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 MLP

```
1  package Model;
2
   import Controle.Comunicador;
4  import Recursos.Arquivo;
   import java.io.BufferedReader;
6  import java.io.FileNotFoundException;
   import java.io.FileReader;
8  import java.io.IOException;
   import java.util.Random;
10
   /**
12  *
    * @author Jônatas Trabuco Belotti [jonatas.t.belotti@hotmail.com]
14  */
   public class MLP {
16
     public static final int NUM_SINAIS_ENTRADA = 3;
18   private final double TAXA_APRENDIZAGEM = 0.1;
     private final double PRECISAO = 0.000001;
20   private final double BETA = 1.0;
     private final int NUM_NEU_CAMADA_ESCONDIDA = 10;
22   private final int NUM_NEU_CAMADA_SAIDA = 1;

24   private int numEpocas;
     private double[] entradas;
26   private double[][] pesosCamadaEscondida;
     private double[] pesosCamadaSaida;
28   private double[] potencialCamadaEscondida;
     private double[] saidaCamadaEscondida;
30   private double saida;

32   public MLP() {
       Random random;
34
       entradas = new double[NUM_SINAIS_ENTRADA + 1];
36     pesosCamadaEscondida = new double[NUM_NEU_CAMADA_ESCONDIDA][
          NUM_SINAIS_ENTRADA + 1];
       pesosCamadaSaida = new double[NUM_NEU_CAMADA_ESCONDIDA + 1];
38     saidaCamadaEscondida = new double[NUM_NEU_CAMADA_ESCONDIDA +
```

```
        1];
    potencialCamadaEscondida = new double[NUM_NEU_CAMADA_ESCONDIDA
        + 1];

    //Iniciando os pesos sinpticos
    random = new Random();
    for (int i = 0; i < NUM_NEU_CAMADA_ESCONDIDA; i++) {
      for (int j = 0; j < NUM_SINAIS_ENTRADA + 1; j++) {
        pesosCamadaEscondida[i][j] = random.nextDouble();
      }

      pesosCamadaSaida[i] = random.nextDouble();
    }
    pesosCamadaSaida[NUM_NEU_CAMADA_ESCONDIDA] = random.nextDouble
        ();
  }

  public boolean treinar(Arquivo arquivoTreinamento) {
    FileReader arq;
    BufferedReader lerArq;
    String linha;
    String[] vetor;
    int i;
    double saidaEsperada;
    double erroAtual;
    double erroAnterior;
    double valorParcial;
    double gradienteCamadaSaida;
    double gradienteCamadaEscondida;

    numEpocas = 0;
    erroAtual = erroQuadraticoMedio(arquivoTreinamento);

    Comunicador.iniciarLog("Início treinamento da MLP");
    Comunicador.addLog(String.format("Erro inicial: %.10f",
        erroAtual).replace(".", ","));
    Comunicador.addLog("Época Eqm");

    try {
      do {
        this.numEpocas++;
        erroAnterior = erroAtual;
```

```
        arq = new FileReader(arquivoTreinamento.getCaminhoCompleto
            ());
78      lerArq = new BufferedReader(arq);

80      linha = lerArq.readLine();
        if (linha.contains("x1")) {
82        linha = lerArq.readLine();
        }
84
        while (linha != null) {
86        vetor = linha.split("\\s+");
          i = 0;
88
          if (vetor[0].equals("")) {
90          i = 1;
          }
92
          entradas[0] = -1.0;
94        entradas[1] = Double.parseDouble(vetor[i++].replace(",",
              "."));
          entradas[2] = Double.parseDouble(vetor[i++].replace(",",
              "."));
96        entradas[3] = Double.parseDouble(vetor[i++].replace(",",
              "."));
          saidaEsperada = Double.parseDouble(vetor[i].replace(",",
              "."));
98
          //Calculando saidas da camada escondida
100       saidaCamadaEscondida[0] = potencialCamadaEscondida[0] =
              -1D;
          for (i = 1; i < saidaCamadaEscondida.length; i++) {
102         valorParcial = 0D;

104         for (int j = 0; j < NUM_SINAIS_ENTRADA + 1; j++) {
              valorParcial += entradas[j] * pesosCamadaEscondida[i
                  - 1][j];
106         }

108         potencialCamadaEscondida[i] = valorParcial;
            saidaCamadaEscondida[i] = funcaoLogistica(valorParcial)
                ;
110       }
```

```
112             //Calculando saida da camada de saída
                valorParcial = -1D * pesosCamadaSaida[0];
114             for (i = 1; i < NUM_NEU_CAMADA_ESCONDIDA + 1; i++) {
                  valorParcial += saidaCamadaEscondida[i - 1] *
                     pesosCamadaSaida[i];
116             }
                saida = funcaoLogistica(valorParcial);
118
                //Ajustando pesos sinapticos da camada de saida
120             gradienteCamadaSaida = (saidaEsperada - saida) *
                   funcaoLogisticaDerivada(valorParcial);

122             for (i = 0; i < pesosCamadaSaida.length; i++) {
                  pesosCamadaSaida[i] = pesosCamadaSaida[i] + (
                     TAXA_APRENDIZAGEM * gradienteCamadaSaida *
                     saidaCamadaEscondida[i]);
124             }

126             //Ajustando pesos sinapticos da camada escondida
                for (i = 0; i < NUM_NEU_CAMADA_ESCONDIDA; i++) {
128               gradienteCamadaEscondida = gradienteCamadaSaida *
                     pesosCamadaSaida[i + 1] * funcaoLogisticaDerivada(
                     potencialCamadaEscondida[i + 1]);

130               for (int j = 0; j < NUM_SINAIS_ENTRADA + 1; j++) {
                    pesosCamadaEscondida[i][j] = pesosCamadaEscondida[i][
                       j] + (TAXA_APRENDIZAGEM * gradienteCamadaEscondida
                        * entradas[j]);
132               }
                }
134
                  linha = lerArq.readLine();
136             }

138           arq.close();
              erroAtual = erroQuadraticoMedio(arquivoTreinamento);
140           Comunicador.addLog(String.format("%d   %.10f", numEpocas,
                 erroAtual).replace(".", ","));
            } while (Math.abs(erroAtual - erroAnterior) > PRECISAO &&
               numEpocas < 10000);
142
```

```
        Comunicador.addLog("Fim do treinamento.");
144   } catch (FileNotFoundException ex) {
        return false;
146   } catch (IOException ex) {
        return false;
148   }

150   return true;
    }
152
    public void testar(Arquivo arquivoTeste) {
154   FileReader arq;
      BufferedReader lerArq;
156   String linha;
      String[] vetor;
158   int i;
      int numAmostras;
160   double saidaEsperada;
      double valorParcial;
162   double erroRelativo;
      double variancia;
164
      Comunicador.iniciarLog("Início teste da MLP");
166   Comunicador.addLog("Resposta - Saída rede          Erro");
      erroRelativo = 0D;
168   variancia = 0D;
      numAmostras = 0;
170
      try {
172       arq = new FileReader(arquivoTeste.getCaminhoCompleto());
          lerArq = new BufferedReader(arq);
174
          linha = lerArq.readLine();
176       if (linha.contains("x1")) {
            linha = lerArq.readLine();
178       }

180       while (linha != null) {
            vetor = linha.split("\\s+");
182         i = 0;

184         if (vetor[0].equals("")) {
```

```
            i = 1;
186         }

188         numAmostras++;
            entradas[0] = -1.0;
190         entradas[1] = Double.parseDouble(vetor[i++].replace(",",
               "."));
            entradas[2] = Double.parseDouble(vetor[i++].replace(",",
               "."));
192         entradas[3] = Double.parseDouble(vetor[i++].replace(",",
               "."));
            saidaEsperada = Double.parseDouble(vetor[i].replace(",",
               "."));
194
            //Calculando saidas da camada escondida
196         saidaCamadaEscondida[0] = potencialCamadaEscondida[0] =
               -1D;
            for (i = 1; i < saidaCamadaEscondida.length; i++) {
198           valorParcial = 0D;

200           for (int j = 0; j < NUM_SINAIS_ENTRADA + 1; j++) {
                valorParcial += entradas[j] * pesosCamadaEscondida[i
                   - 1][j];
202           }

204           potencialCamadaEscondida[i] = valorParcial;
              saidaCamadaEscondida[i] = funcaoLogistica(valorParcial)
                 ;
206         }

208         //Calculando saida da camada de saída
            valorParcial = -1D * pesosCamadaSaida[0];
210         for (i = 1; i < NUM_NEU_CAMADA_ESCONDIDA + 1; i++) {
              valorParcial += saidaCamadaEscondida[i - 1] *
                 pesosCamadaSaida[i];
212         }
            saida = funcaoLogistica(valorParcial);
214
            //Porcentagem de erro em cada amostra
216         valorParcial = (100D / saidaEsperada) * (Math.abs(
               saidaEsperada - saida));
            erroRelativo += valorParcial;
```

```
218             variancia += Math.pow(valorParcial, 2D);
                Comunicador.addLog(String.format("%.4f          %.10f
                    %.10f%%", saidaEsperada, saida, valorParcial));
220
                linha = lerArq.readLine();
222           }

224           //Calculando erro relativo médio
              erroRelativo = erroRelativo / (double) numAmostras;
226
              //Calculando variância
228           variancia = variancia - ((double)numAmostras * Math.pow(
                  erroRelativo, 2D));
              variancia = variancia / ((double) (numAmostras - 1));
230           variancia = variancia * 100D;

232           Comunicador.addLog("Fim do teste");
              Comunicador.addLog(String.format("Erro relativo médio: %.10
                  f%%", erroRelativo));
234           Comunicador.addLog(String.format("Variância: %.10f%%",
                  variancia));

236           arq.close();
        } catch (FileNotFoundException ex) {
238     } catch (IOException ex) {
        }
240   }

242   private double erroQuadraticoMedio(Arquivo arquivo) {
        FileReader arq;
244     BufferedReader lerArq;
        String linha;
246     String[] vetor;
        int i;
248     int numAmostras;
        double saidaEsperada;
250     double valorParcial;
        double erro;
252
        erro = 0D;
254     numAmostras = 0;
```

```
256     try {
          arq = new FileReader(arquivo.getCaminhoCompleto());
258       lerArq = new BufferedReader(arq);

260       linha = lerArq.readLine();
          if (linha.contains("x1")) {
262         linha = lerArq.readLine();
          }
264
          while (linha != null) {
266         vetor = linha.split("\\s+");
            i = 0;
268
            if (vetor[0].equals("")) {
270           i = 1;
            }
272
            numAmostras++;
274         entradas[0] = -1.0;
            entradas[1] = Double.parseDouble(vetor[i++].replace(",", ".
               "));
276         entradas[2] = Double.parseDouble(vetor[i++].replace(",", ".
               "));
            entradas[3] = Double.parseDouble(vetor[i++].replace(",", ".
               "));
278         saidaEsperada = Double.parseDouble(vetor[i].replace(",", ".
               "));

280         //Calculando saidas da camada escondida
            saidaCamadaEscondida[0] = potencialCamadaEscondida[0] = -1D
               ;
282         for (i = 1; i < saidaCamadaEscondida.length; i++) {
              valorParcial = 0D;
284
              for (int j = 0; j < NUM_SINAIS_ENTRADA + 1; j++) {
286             valorParcial += entradas[j] * pesosCamadaEscondida[i -
                   1][j];
              }
288
              potencialCamadaEscondida[i] = valorParcial;
290           saidaCamadaEscondida[i] = funcaoLogistica(valorParcial);
            }
```

```
292
            //Calculando saida da camada de saída
294         valorParcial = -1D * pesosCamadaSaida[0];
            for (i = 1; i < NUM_NEU_CAMADA_ESCONDIDA + 1; i++) {
296           valorParcial += saidaCamadaEscondida[i - 1] *
                 pesosCamadaSaida[i];
            }
298         saida = funcaoLogistica(valorParcial);

300         //Calculando erro
            erro = erro + (Math.pow((saidaEsperada - this.saida), 2D) /
                2D);
302
            linha = lerArq.readLine();
304       }

306       arq.close();
          erro = erro / (double) numAmostras;
308
        } catch (FileNotFoundException ex) {
310     } catch (IOException ex) {
        }
312
        return erro;
314   }

316   private double funcaoLogistica(double valor) {
        return 1D / (1D + Math.pow(Math.E, -1D * BETA * valor));
318   }

320   private double funcaoLogisticaDerivada(double valor) {
        return (BETA * Math.pow(Math.E, -1D * BETA * valor)) / Math.pow
            ((Math.pow(Math.E, -1D * BETA * valor) + 1D), 2D);
322   }

324 }
```

ANEXO A - CONJUNTO DE TREINAMENTO

x1	x2	x3	d
0.8799	0.7998	0.3972	0.8399
0.5700	0.5111	0.2418	0.6258
0.6796	0.4117	0.3370	0.6622
0.3567	0.2967	0.6037	0.5969
0.3866	0.8390	0.0232	0.5316
0.0271	0.7788	0.7445	0.6335
0.8174	0.8422	0.3229	0.8068
0.6027	0.1468	0.3759	0.5342
0.1203	0.3260	0.5419	0.4768
0.1325	0.2082	0.4934	0.4105
0.6950	1.0000	0.4321	0.8404
0.0036	0.1940	0.3274	0.2697
0.2650	0.0161	0.5947	0.4125
0.5849	0.6019	0.4376	0.7464
0.0108	0.3538	0.1810	0.2800
0.9008	0.7264	0.9184	0.9602
0.0023	0.9659	0.3182	0.4986
0.1366	0.6357	0.6967	0.6459
0.8621	0.7353	0.2742	0.7718
0.0682	0.9624	0.4211	0.5764
0.6112	0.6014	0.5254	0.7868
0.0030	0.7585	0.8928	0.6388
0.7644	0.5964	0.0407	0.6055
0.6441	0.2097	0.5847	0.6545
0.0803	0.3799	0.6020	0.4991
0.1908	0.8046	0.5402	0.6665
0.6937	0.3967	0.6055	0.7595
0.2591	0.0582	0.3978	0.3604
0.4241	0.1850	0.9066	0.6298
0.3332	0.9303	0.2475	0.6287
0.3625	0.1592	0.9981	0.5948
0.9259	0.0960	0.1645	0.4716
0.8606	0.6779	0.0033	0.6242
0.0838	0.5472	0.3758	0.4835
0.0303	0.9191	0.7233	0.6491
0.9293	0.8319	0.9664	0.9840

0.7268	0.1440	0.9753	0.7096
0.2888	0.6593	0.4078	0.6328
0.5515	0.1364	0.2894	0.4745
0.7683	0.0067	0.5546	0.5708
0.6462	0.6761	0.8340	0.8933
0.3694	0.2212	0.1233	0.3658
0.2706	0.3222	0.9996	0.6310
0.6282	0.1404	0.8474	0.6733
0.5861	0.6693	0.3818	0.7433
0.6057	0.9901	0.5141	0.8466
0.5915	0.5588	0.3055	0.6787
0.8359	0.4145	0.5016	0.7597
0.5497	0.6319	0.8382	0.8521
0.7072	0.1721	0.3812	0.5772
0.1185	0.5084	0.8376	0.6211
0.6365	0.5562	0.4965	0.7693
0.4145	0.5797	0.8599	0.7878
0.2575	0.5358	0.4028	0.5777
0.2026	0.3300	0.3054	0.4261
0.3385	0.0476	0.5941	0.4625
0.4094	0.1726	0.7803	0.6015
0.1261	0.6181	0.4927	0.5739
0.1224	0.4662	0.2146	0.4007
0.6793	0.6774	1.0000	0.9141
0.8176	0.0358	0.2506	0.4707
0.6937	0.6685	0.5075	0.8220
0.2404	0.5411	0.8754	0.6980
0.6553	0.2609	0.1188	0.4851
0.8886	0.0288	0.2604	0.4802
0.3974	0.5275	0.6457	0.7215
0.2108	0.4910	0.5432	0.5913
0.8675	0.5571	0.1849	0.6805
0.5693	0.0242	0.9293	0.6033
0.8439	0.4631	0.6345	0.8226
0.3644	0.2948	0.3937	0.5240
0.2014	0.6326	0.9782	0.7143
0.4039	0.0645	0.4629	0.4547
0.7137	0.0670	0.2359	0.4602
0.4277	0.9555	0	0.5477
0.0259	0.7634	0.2889	0.4738

0.1871	0.7682	0.9697	0.7397
0.3216	0.5420	0.0677	0.4526
0.2524	0.7688	0.9523	0.7711
0.3621	0.5295	0.2521	0.5571
0.2942	0.1625	0.2745	0.3759
0.8180	0.0023	0.1439	0.4018
0.8429	0.1704	0.5251	0.6563
0.9612	0.6898	0.6630	0.9128
0.1009	0.4190	0.0826	0.3055
0.7071	0.7704	0.8328	0.9298
0.3371	0.7819	0.0959	0.5377
0.1555	0.5599	0.9221	0.6663
0.7318	0.1877	0.3311	0.5689
0.1665	0.7449	0.0997	0.4508
0.8762	0.2498	0.9167	0.7829
0.9885	0.6229	0.2085	0.7200
0.0461	0.7745	0.5632	0.5949
0.3209	0.6229	0.5233	0.6810
0.9189	0.5930	0.7288	0.8989
0.0382	0.5515	0.8818	0.5999
0.3726	0.9988	0.3814	0.7086
0.4211	0.2668	0.3307	0.5080
0.2378	0.0817	0.3574	0.3452
0.9893	0.7637	0.2526	0.7755
0.8203	0.0682	0.4260	0.5643
0.6226	0.2146	0.1021	0.4452
0.4589	0.3147	0.2236	0.4962
0.3471	0.8889	0.1564	0.5875
0.5762	0.8292	0.4116	0.7853
0.9053	0.6245	0.5264	0.8506
0.2860	0.0793	0.0549	0.2224
0.9567	0.3034	0.4425	0.6993
0.5170	0.9266	0.1565	0.6594
0.8149	0.0396	0.6227	0.6165
0.3710	0.3554	0.5633	0.6171
0.8702	0.3185	0.2762	0.6287
0.1016	0.6382	0.3173	0.4957
0.3890	0.2369	0.0083	0.3235
0.2702	0.8617	0.1218	0.5319
0.7473	0.6507	0.5582	0.8464

0.9108	0.2139	0.4641	0.6625
0.4343	0.6028	0.1344	0.5546
0.6847	0.4062	0.9318	0.8204
0.8657	0.9448	0.9900	0.9904
0.4011	0.4138	0.8715	0.7222
0.5949	0.2600	0.0810	0.4480
0.1845	0.7906	0.9725	0.7425
0.3438	0.6725	0.9821	0.7926
0.8398	0.1360	0.9119	0.7222
0.2245	0.0971	0.6136	0.4402
0.3742	0.9668	0.8194	0.8371
0.9572	0.9836	0.3793	0.8556
0.7496	0.0410	0.1360	0.4059
0.9123	0.3510	0.0682	0.5455
0.6954	0.5500	0.6801	0.8388
0.5252	0.6529	0.5729	0.7893
0.3156	0.3851	0.5983	0.6161
0.1460	0.1637	0.0249	0.1813
0.7780	0.4491	0.4614	0.7498
0.5959	0.8647	0.8601	0.9176
0.2204	0.1785	0.4607	0.4276
0.7355	0.8264	0.7015	0.9214
0.9931	0.6727	0.3139	0.7829
0.9123	0	0.1106	0.3944
0.2858	0.9688	0.2262	0.5988
0.7931	0.8993	0.9028	0.9728
0.7841	0.0778	0.9012	0.6832
0.1380	0.5881	0.2367	0.4622
0.6345	0.5165	0.7139	0.8191
0.2453	0.5888	0.1559	0.4765
0.1174	0.5436	0.3657	0.4953
0.3667	0.3228	0.6952	0.6376
0.9532	0.6949	0.4451	0.8426
0.7954	0.8346	0.0449	0.6676
0.1427	0.0480	0.6267	0.3780
0.1516	0.9824	0.0827	0.4627
0.4868	0.6223	0.7462	0.8116
0.3408	0.5115	0.0783	0.4559
0.8146	0.6378	0.5837	0.8628
0.2820	0.5409	0.7256	0.6939

0.5716	0.2958	0.5477	0.6619
0.9323	0.0229	0.4797	0.5731
0.2907	0.7245	0.5165	0.6911
0.0068	0.0545	0.0861	0.0851
0.2636	0.9885	0.2175	0.5847
0.0350	0.3653	0.7801	0.5117
0.9670	0.3031	0.7127	0.7836
0	0.7763	0.8735	0.6388
0.4395	0.0501	0.9761	0.5712
0.9359	0.0366	0.9514	0.6826
0.0173	0.9548	0.4289	0.5527
0.6112	0.9070	0.6286	0.8803
0.2010	0.9573	0.6791	0.7283
0.8914	0.9144	0.2641	0.7966
0.0061	0.0802	0.8621	0.3711
0.2212	0.4664	0.3821	0.5260
0.2401	0.6964	0.0751	0.4637
0.7881	0.9833	0.3038	0.8049
0.2435	0.0794	0.5551	0.4223
0.2752	0.8414	0.2797	0.6079
0.7616	0.4698	0.5337	0.7809
0.3395	0.0022	0.0087	0.1836
0.7849	0.9981	0.4449	0.8641
0.8312	0.0961	0.2129	0.4857
0.9763	0.1102	0.6227	0.6667
0.8597	0.3284	0.6932	0.7829
0.9295	0.3275	0.7536	0.8016
0.2435	0.2163	0.7625	0.5449
0.9281	0.8356	0.5285	0.8991
0.8313	0.7566	0.6192	0.9047
0.1712	0.0545	0.5033	0.3561
0.0609	0.1702	0.4306	0.3310
0.5899	0.9408	0.0369	0.6245
0.7858	0.5115	0.0916	0.6066
1.0000	0.1653	0.7103	0.7172
0.2007	0.1163	0.3431	0.3385
0.2306	0.0330	0.0293	0.1590
0.8477	0.6378	0.4623	0.8254
0.9677	0.7895	0.9467	0.9782
0.0339	0.4669	0.1526	0.3250

```
0.0080    0.8988    0.4201    0.5404
0.9955    0.8897    0.6175    0.9360
0.7408    0.5351    0.2732    0.6949
0.6843    0.3737    0.1562    0.5625
```

ANEXO B - CONJUNTO DE TESTE

x1	x2	x3	d
0.0611	0.2860	0.7464	0.4831
0.5102	0.7464	0.0860	0.5965
0.0004	0.6916	0.5006	0.5318
0.9430	0.4476	0.2648	0.6843
0.1399	0.1610	0.2477	0.2872
0.6423	0.3229	0.8567	0.7663
0.6492	0.0007	0.6422	0.5666
0.1818	0.5078	0.9046	0.6601
0.7382	0.2647	0.1916	0.5427
0.3879	0.1307	0.8656	0.5836
0.1903	0.6523	0.7820	0.6950
0.8401	0.4490	0.2719	0.6790
0.0029	0.3264	0.2476	0.2956
0.7088	0.9342	0.2763	0.7742
0.1283	0.1882	0.7253	0.4662
0.8882	0.3077	0.8931	0.8093
0.2225	0.9182	0.7820	0.7581
0.1957	0.8423	0.3085	0.5826
0.9991	0.5914	0.3933	0.7938
0.2299	0.1524	0.7353	0.5012