UNIVERSIDADE ESTADUAL PAULISTA

FACULDADE DE CIÊNCIAS E TECNOLOGIA

IMPLEMENTAÇÃO REDE NEURAL: BACK-PROPAGATION

RELATÓRIO

INTELIGÊNCIA ARTIFICIAL

PROF. DR. ALMIR OLIVETTE ARTERO

BRUNO SANTOS DE LIMA

LEANDRO UNGARI CAYRES

PRESIDENTE PRUDENTE

AGOSTO - 2017

BRUNO SANTOS DE LIMA

LEANDRO UNGARI CAYRES

IMPLEMENTAÇÃO REDE NEURAL: BACK-PROPAGATION

RELATÓRIO

Relatório do segundo trabalho prático da disciplina de Inteligência Artificial, lecionada pelo docente Dr. Almir Olivette Artero, no curso Bacharelado em Ciência da Computação – Departamento de Matemática e Computação da Faculdade de Ciências e Tecnologia (FCT Unesp – Presidente Prudente).

PRESIDENTE PRUDENTE

AGOSTO – 2017

**SUMÁRIO**

[1 INTRODUÇÃO 4](#_Toc489798767)

[2 Ferramenta: Rede neural 6](#_Toc489798768)

[2.1 Utilização da ferramenta. 7](#_Toc489798769)

[3 Resultados 13](#_Toc489798770)

[3.1 Primeira configuração 13](#_Toc489798771)

[3.2 Segunda configuração 13](#_Toc489798772)

[3.3 Terceira configuração 14](#_Toc489798773)

[3.4 Quarta configuração 14](#_Toc489798774)

[4 Codificações 14](#_Toc489798775)

[REFERÊNCIAS BIBLIOGRÁFICAS 27](#_Toc489798776)

**LISTA DE FIGURAS**

[Figura 1 – Instantâneo da interface da ferramenta 6](#_Toc489798781)

[Figura 2 - Carregar Treinamento 7](#_Toc489798782)

[Figura 3 - Arquivo de treinamento carregado 8](#_Toc489798783)

[Figura 4 - Informações da Estrutura da rede neural 8](#_Toc489798784)

[Figura 5 - Alterar número de neurônios da camada oculta 9](#_Toc489798785)

[Figura 6 - Alterar pesos 9](#_Toc489798786)

[Figura 7 - Exemplo alterar pesos 9](#_Toc489798787)

[Figura 8 - Treinamento da rede neural 10](#_Toc489798788)

[Figura 9 - Carregar Teste 11](#_Toc489798789)

[Figura 10 - Arquivo de teste carregado 11](#_Toc489798790)

[Figura 11 - Finalização do teste 12](#_Toc489798791)

[Figura 12 - Limpando arquivos de treinamento e teste para uma nova execução de uma nova rede neural 12](#_Toc489798792)

# **1 INTRODUÇÃO**

Neste trabalho da disciplina de Inteligência Artificial o objetivo era desenvolver uma ferramenta na qual o usuário da mesma consiga trabalhar e simular uma Rede Neural de três camadas.

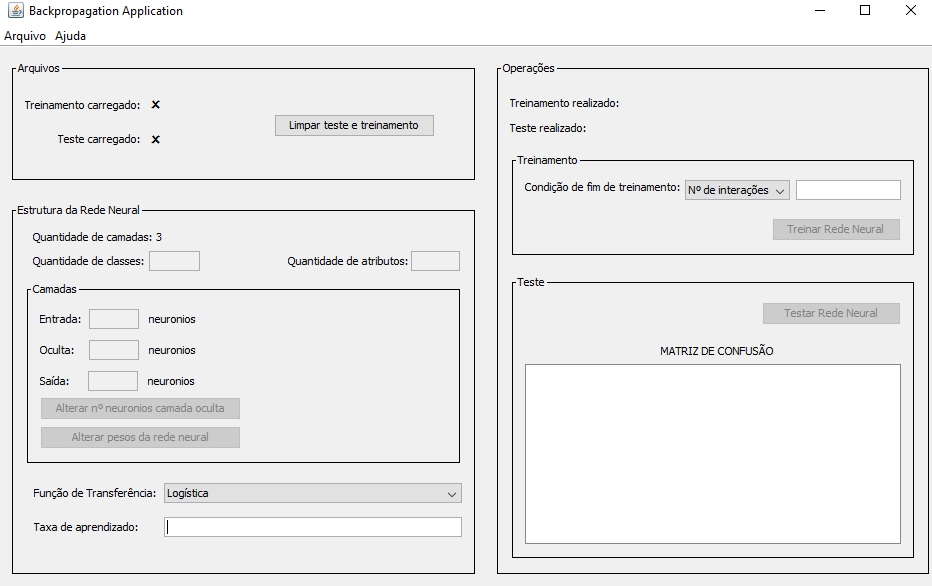
A ferramenta foi implementada utilizando a linguagem de programação Java. Como IDE foi utilizado o NetBeans, contando também com a utilização de um sistema de controle de versão Git para versionamento do projeto.

Este relatório está divido em: Na seção 2 é descrito detalhes da rede neural e de como utilizar a ferramenta, a seção 3 relata os resultados do treinamento e do teste realizado na rede neural implementada, além de apresentar algumas conclusões sobre os resultados em diferentes configurações, por fim a seção 4 é descrito todas as codificações (classes Java) implementadas.

# **2 Ferramenta: Rede neural**

As redes neurais foram desenvolvidas nos anos 40 por McCulloch e pelo matemático Pitts. Acreditava-se que as redes neurais seriam capazes de simular o comportamento do cérebro humano, sendo assim capaz de resolver qualquer tipo de problema. Apesar de conseguir solucionar uma alta gama de problemas sua principal utilização é na resolução de problemas de reconhecimento de padrões, seja em voz, imagens, faces, impressões digitais, entre outros. [1]

A rede neural implementada neste trabalho contém três camadas e utiliza o princípio de Back-Propagation, desenvolvido na década de 80. Trata-se de um algoritmo de treinamento mais utilizado em redes de múltiplas camadas, onde o erro obtido na camada de saída é propagado pela rede até a camada de entrada [1]. Na Figura 1 é ilustrado a interface da ferramenta desenvolvida.

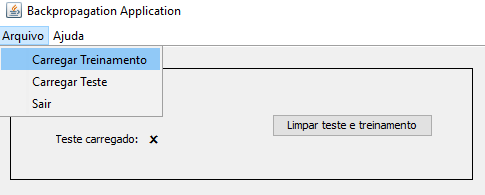


**Figura 1 – Instantâneo da interface da ferramenta**

## **2.1 Utilização da ferramenta.**

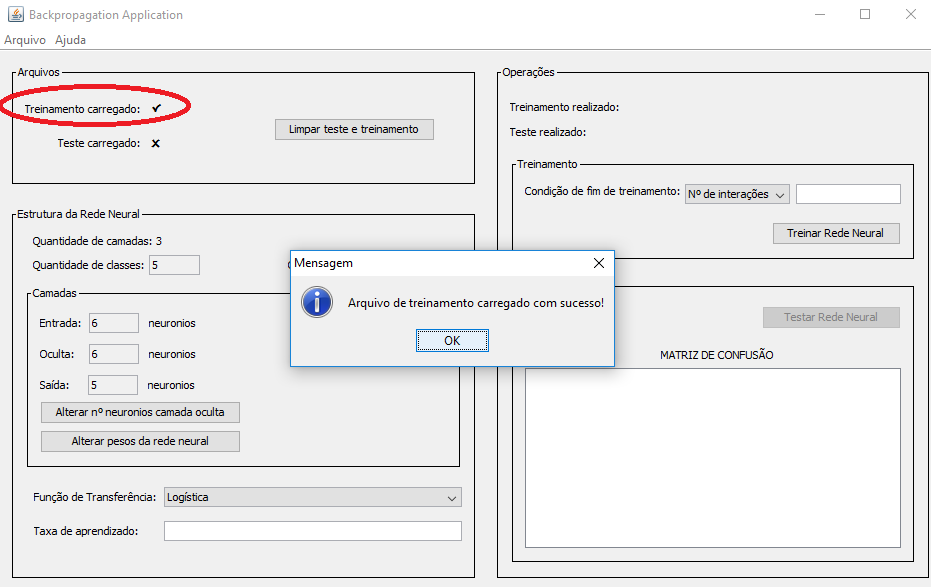
Para utilizar a ferramenta deve-se seguir alguns passos afim de garantir um bom funcionamento. Existem dois passos básicos: Treinamento e Teste. Para realizar esses dois passos deve ser carregado arquivos com a extensão .csv.

Primeiramente deve-se treinar a rede para isso necessita carregar o arquivo de treinamento, clique em **Arquivo -> Carregar Treinamento**, escolha o arquivo e pronto o arquivo de treinamento já estará carregado. Na Figura 2 é ilustrado esse processo.



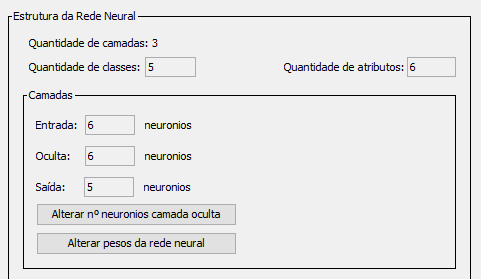
**Figura 2 - Carregar Treinamento**

Após selecionar o arquivo de treinamento, uma mensagem informada que o arquivo foi carregado com sucesso aparecerá, além disso uma letra V no canto superior esquerdo da ferramenta indicará que o arquivo foi carregado. Observe na Figura 3.



**Figura 3 - Arquivo de treinamento carregado**

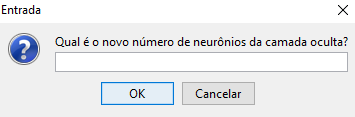
Após carregar o arquivo de treinamento é possível ver a estrutura da rede, as informações de quantidade de camadas, classes, atributos, e neurônios de cada camada são mostrada na interface no campo Estrutura da Rede Neural. Na Figura 4 é ilustrada essas informações.



**Figura 4 - Informações da Estrutura da rede neural**

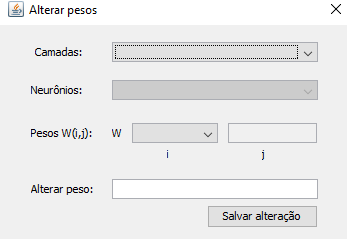
Neste momento antes de treinar a rede é possível fazer duas modificações: alterar o número de neurônios da camada oculta e também alterar os pesos da rede neural, para qualquer uma dessas modificações basta clicar em um dos botões mostrados na Figura 4.

Clicando no botão **Alterar o nº de neurônios camada oculta**, basta inserir a nova quantidade na janela que aparecerá, na Figura 5 é mostrada essa janela.

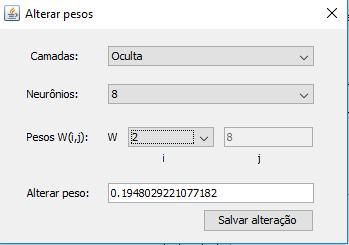


**Figura 5 - Alterar número de neurônios da camada oculta**

Clicando no botão **Alterar pesos da rede neural** uma janela será aberta, Figura 6, para escolher qual camada, neurônio e peso que deseja alterar. A Figura 7 mostra um exemplo.

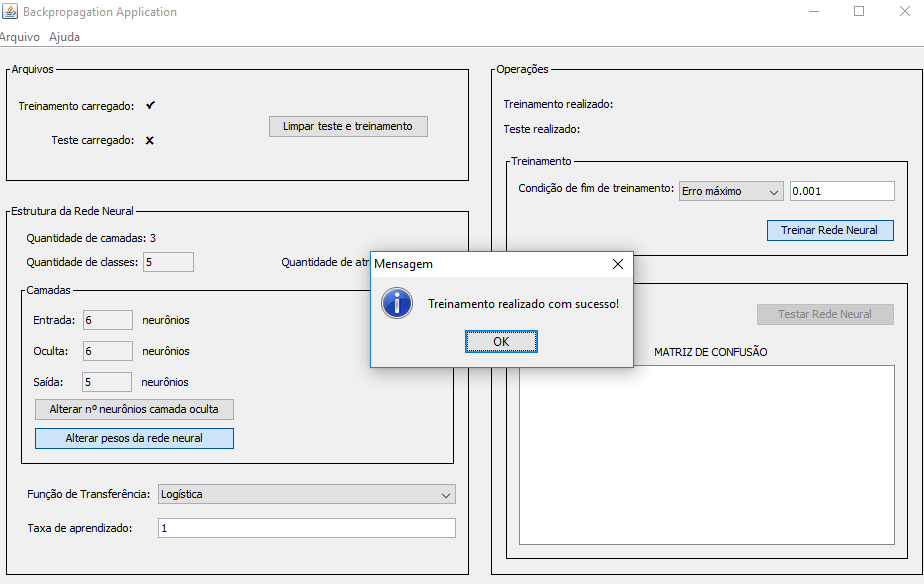


**Figura 6 - Alterar pesos**



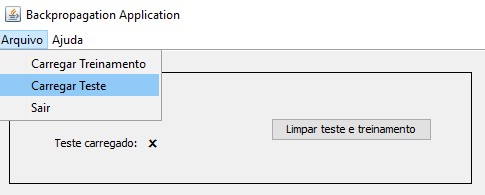
**Figura 7 - Exemplo alterar pesos**

Após essas configurações opcionais deve-se realizar outras duas configurações obrigatórias antes de realizar o treinamento da rede, primeiro deve-se escolher a função de transferência bem como a taxa de aprendizado, depois escolher a condição de parada do treinamento, seja por erro máximo ou por número de interações e especificar seus valores. Com isso basta clicar no botão **Treinar Rede Neural** e o treinamento será realizado, ao fim do treinamento uma mensagem será exibida na tela informando o fim do treinamento e pedindo para carregar o arquivo de teste e realizar o teste na rede. Na Figura 8 é ilustrado esse processo de treinamento.



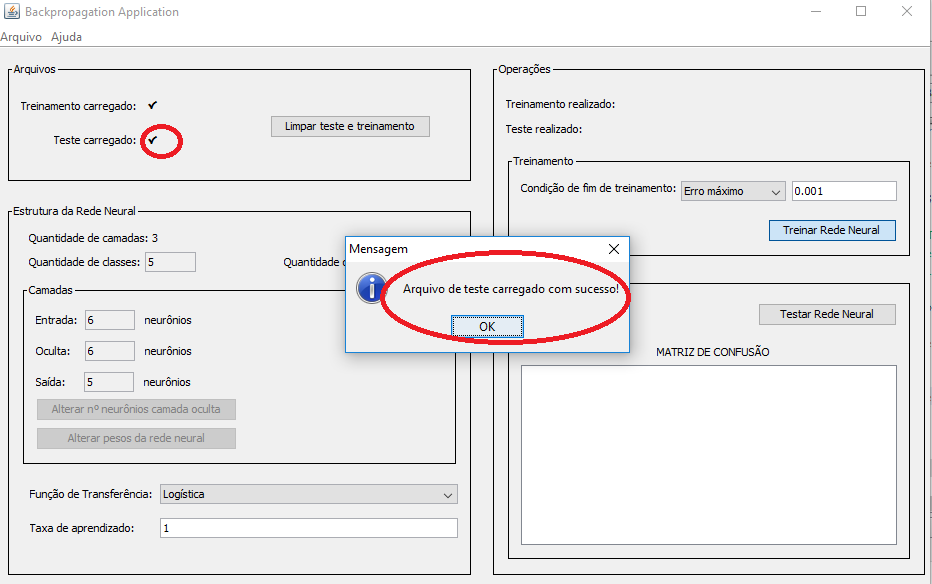
**Figura 8 - Treinamento da rede neural**

Agora com a rede treinada já é possível testar a rede para obter a matriz de confusão. Primeiro deve-se carregar o arquivo de teste, Menu **Arquivo -> Carregar Teste**. Observe a Figura 9.



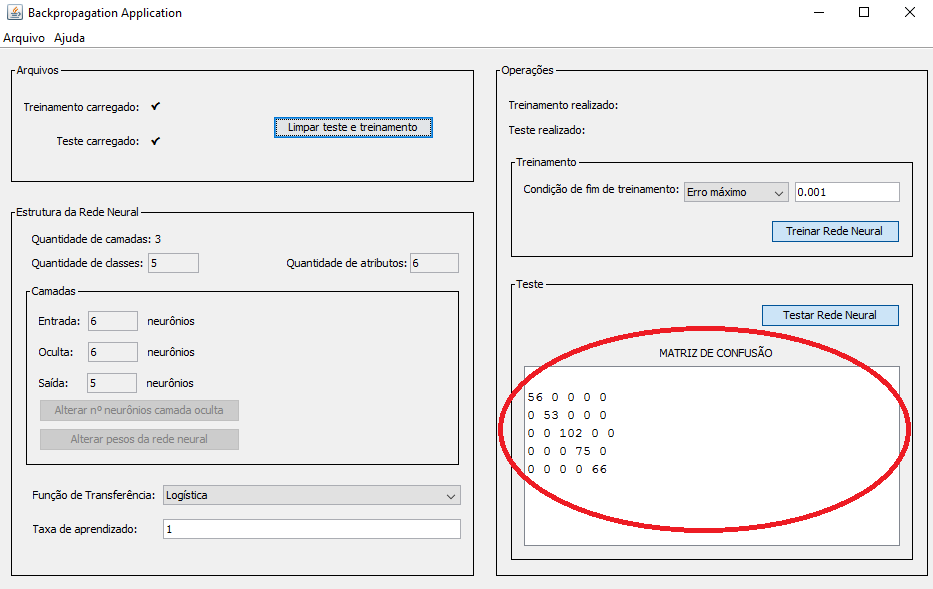
**Figura 9 - Carregar Teste**

Após selecionar o arquivo de teste, uma mensagem informada que o arquivo foi carregado com sucesso aparecerá, além disso uma letra V no canto superior esquerdo da ferramenta indicará que o arquivo de teste também foi carregado. Observe na Figura 10.



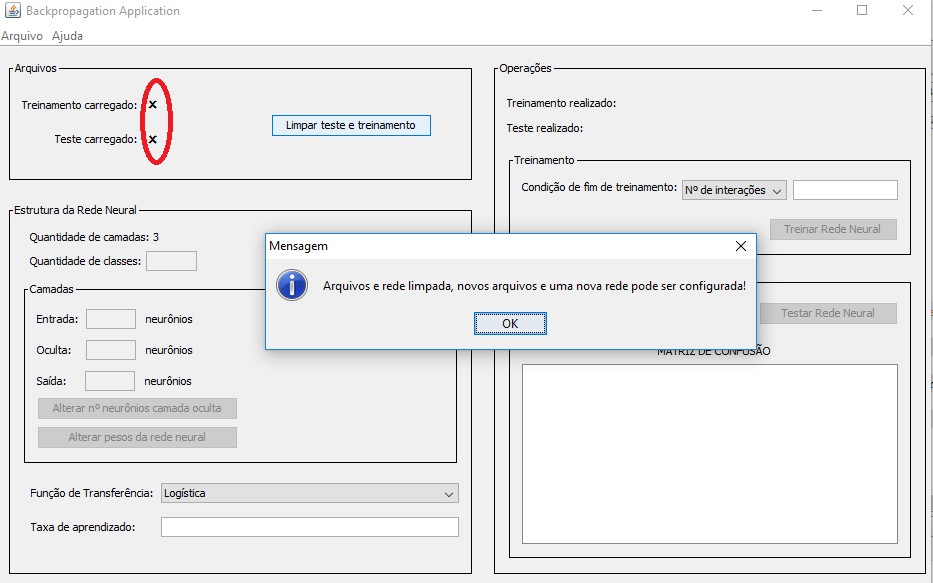
**Figura 10 - Arquivo de teste carregado**

Após isto basta clicar no botão Testar Rede Neural, com isso uma mensagem de teste concluído aparecerá e a matriz de confusão será mostrada, observe a Figura 11 onde esse processo final é ilustrado.



**Figura 11 - Finalização do teste**

Caso queira realizar esse processo novamente basta clicar no botão Limpar teste e treinamento e executar todos os passos acima novamente com novos arquivos. Observe a Figura 12.



**Figura 12 - Limpando arquivos de treinamento e teste para uma nova execução de uma nova rede neural**

# **3 Resultados**

Foram realizados alguns treinamentos e testes utilizando os arquivos “treinamento.csv” e “teste.csv”. Tais treinamentos e testes foram aplicados com configurações diversas.

## **3.1 Primeira configuração**

**Configuração:**

**Quantidade de neurônios na camada oculta:** 6

**Função de Transferência:** Logística

**Taxa de aprendizado:** 1.0

**Condição de fim de treinamento:** Erro máximo: 0.001

**Matriz de confusão:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 56 | 0 | 0 | 0 | 0 |
| 0 | 53 | 0 | 0 | 0 |
| 0 | 0 | 102 | 0 | 0 |
| 0 | 0 | 0 | 75 | 0 |
| 0 | 0 | 0 | 0 | 66 |

## **3.2 Segunda configuração**

**Configuração:**

**Quantidade de neurônios na camada oculta:** 6

**Função de Transferência:** Logística

**Taxa de aprendizado:** 1.0

**Condição de fim de treinamento:** Nº de interações: 10000

**Matriz de confusão:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 56 | 0 | 0 | 0 | 0 |
| 0 | 53 | 0 | 0 | 0 |
| 0 | 0 | 102 | 0 | 0 |
| 0 | 0 | 0 | 75 | 0 |
| 0 | 0 | 0 | 0 | 66 |

## **3.3 Terceira configuração**

**Configuração:**

**Quantidade de neurônios na camada oculta**: 6

**Função de Transferência**: Tangente Hiperbólica

**Taxa de aprendizado:** 0.1

**Condição de fim de treinamento:** Erro máximo: 0.001

**Matriz de confusão:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 56 | 0 | 0 | 0 | 0 |
| 1 | 52 | 0 | 0 | 0 |
| 0 | 0 | 102 | 0 | 0 |
| 0 | 0 | 0 | 75 | 0 |
| 0 | 0 | 0 | 0 | 66 |

## **3.4 Quarta configuração**

**Configuração:**

**Quantidade de neurônios na camada oculta**: 45

**Função de Transferência:** Tangente Hiperbólica

**Taxa de aprendizado**: 0.1

**Condição de fim de treinamento:** Erro máximo: 0.001

**Matriz de confusão:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 56 | 0 | 0 | 0 | 0 |
| 0 | 53 | 0 | 0 | 0 |
| 0 | 0 | 102 | 0 | 0 |
| 0 | 0 | 0 | 75 | 0 |
| 0 | 0 | 0 | 0 | 66 |

# **4 Codificações**

A seguir estão todas as codificações, ou seja, todas as classes Java criadas para construir a ferramenta.

**public** **class** CSVFile {

**public** **static** **int** *numEntrada*;

**public** **static** HashMap<Integer, **double**[]> *classes* = **new** HashMap();

**public** **static** **int** *numSaida*;

/\*\*

\* Realiza a leitura do conjunto de dados no arquivo especificado.

\* **@param** filename Nome do arquivo.

\* **@return** Retorna um ArrayList com o conjunto de dados.

\* **@throws** FileNotFoundException

\*/

**public** **static** ArrayList<Dados> read(String filename) **throws** FileNotFoundException {

**int** classe;

Scanner leitura = **new** Scanner(**new** File(filename));

ArrayList<Dados> dados = **new** ArrayList<>();

**while** (leitura.hasNextLine()) {

String linha = leitura.nextLine().replaceAll("\"", "");

String[] elementos = linha.split(",");

*numEntrada* = elementos.length - 1;

**try** {

Double.*parseDouble*(elementos[0]);

**double**[] d = **new** **double**[elementos.length];

**for** (**int** i = 0; i < elementos.length; i++) {

d[i] = Double.*parseDouble*(elementos[i]);

}

classe = (**int**) d[elementos.length-1];

*classes*.put(classe, **null**);

dados.add(**new** Dados(d));

} **catch** (NumberFormatException e) {

}

}

*numSaida* = *classes*.keySet().size();

**return**(dados);

}

**public** **static** String getClasses() {

String s = "[";

**for**(Integer e: *classes*.keySet()){

s += e + ",";

}

s += "]";

**return**(s);

}

/\*\*

\* Define as saídas esperadas com base no conjunto de classes dos dados de entrada e na função de transferência escolhida.

\* **@param** funcaoTransferencia Função matemática escolhida.

\*/

**public** **static** **void** ajustarSaida(**int** funcaoTransferencia){

**double** min, max;

**if**(funcaoTransferencia == FuncaoTransferencia.LOGISTICA){

min = 0; max = 1;

}

**else**{

min = -1; max = 1;

}

**int** posicao = 0;

**for**(Integer i: *classes*.keySet()){

**double**[] vetor = **new** **double**[*classes*.keySet().size()];

**for**(**int** j = 0; j < vetor.length; j++){

**if**(posicao == j){

vetor[j] = max;

}

**else**{

vetor[j] = min;

}

}

posicao++;

*classes*.put(i, vetor);

}

}

}

**public** **class** Dados {

/\*\*

\* Consiste em um linha de dados, contendo os respectivos atributos e a classe real.

\*/

**private** **double**[] dados;

**public** Dados(**double**[] dados){

**this**.dados = dados;

}

**public** Dados(**int** size) {

**this**.dados = **new** **double**[size];

}

**public** **double** get(**int** i){

**return** dados[i];

}

**public** **void** set(**int** i, **double** e){

dados[i] = e;

}

**public** **double**[] getDados() {

**return** dados;

}

@Override

**public** String toString() {

String s = "[";

**for**(**double** e: **this**.dados){

s += String.*format*("%.2f", e) + ",";

}

**return** s + "\b]";

}

@Override

**public** Dados clone() {

Dados d = **new** Dados(**this**.dados.length);

**for**(**int** i = 0; i < **this**.dados.length; i++){

d.dados[i] = **this**.dados[i];

}

**return** d;

}

}

**public** **class** BackPropagation {

**private** **final** **int** numEntrada;

**private** **int** numOculta;

**private** **final** **int** numSaida;

**private** **double** taxaAprendizagem;

**private** **int** funcao;

**private** **final** Camada[] camadas;

**private** MatrizConfusao matrizConfusao;

**public** **static** **int** *contadorNeuronio*;

**private** **double** erroMax;

**private** **int** numIteracoesMax;

**private** **boolean** opcaoErro;

**public** **static** Random *random*;

**public** **static** **int** *PESO\_ALEATORIO* = 1;

**public** **static** **int** *PESO\_GAUSSIANO* = 2;

**public** **static** **int** *tipoPeso*;

/\*\*

\* A classe {@code BackPropagation} refere-se a uma rede neural cujo

\* algoritmo de treinamento consiste no algoritmo Backpropagation.

\*

\* **@param** numEntrada Número de neurônios na camada de entrada.

\* **@param** numSaida Número de neurônios na camada de saída.

\* **@param** numOculta Número de neurônios na camada oculta.

\* **@param** tipoPeso Constante número que define o tipo de peso entre

\* aleatório ou gaussiano.

\*/

**public** BackPropagation(**int** numEntrada, **int** numSaida, **int** numOculta, **int** tipoPeso) {

*contadorNeuronio* = 0;

**if** (tipoPeso == *PESO\_GAUSSIANO*) {

BackPropagation.*random* = **new** Random();

}

BackPropagation.*tipoPeso* = tipoPeso;

**this**.numEntrada = numEntrada;

**this**.numOculta = numOculta;

**this**.numSaida = numSaida;

**this**.camadas = **new** Camada[3];

**this**.camadas[Camada.ENTRADA] = **new** Camada(numEntrada, **null**);

**this**.camadas[Camada.OCULTA] = **new** Camada(numOculta, **this**.camadas[Camada.ENTRADA]);

**this**.camadas[Camada.SAIDA] = **new** Camada(numSaida, **this**.camadas[Camada.OCULTA]);

}

/\*\*

\* Atribui informações à rede.

\*

\* **@param** taxaAprendizagem Taxa de modificação da rede.

\* **@param** funcao Função matemática de processamento da rede.

\* **@param** erroMax Erro máximo no processo de treinamento.

\*/

**public** **void** inicializar(**double** taxaAprendizagem, **int** funcao, **double** erroMax) {

**this**.taxaAprendizagem = taxaAprendizagem;

**this**.opcaoErro = **true**;

**this**.erroMax = erroMax;

**this**.numIteracoesMax = 0;

**this**.funcao = funcao;

FuncaoTransferencia.FUNCAO\_ATUAL = **this**.funcao;

}

/\*\*

\* Atribui informações à rede.

\*

\* **@param** taxaAprendizagem Taxa de modificação da rede.

\* **@param** funcao Função matemática de processamento da rede.

\* **@param** numIteracoesMax Número máximo de iterações no treinamento.

\*/

**public** **void** inicializar(**double** taxaAprendizagem, **int** funcao, **int** numIteracoesMax) {

**this**.taxaAprendizagem = taxaAprendizagem;

**this**.opcaoErro = **false**;

**this**.erroMax = 0;

**this**.numIteracoesMax = numIteracoesMax;

**this**.funcao = funcao;

FuncaoTransferencia.FUNCAO\_ATUAL = **this**.funcao;

}

/\*\*

\* O método {@code teste()} executa o teste de um conjunto de dados

\* identificado pelo parâmetro {@code dados}, em seguida, categoriza-os na

\* matriz de confusão da rede.

\*

\* **@param** dados Conjunto de dados para o teste.

\*/

**public** **void** teste(ArrayList<Dados> dados) {

**this**.matrizConfusao = **new** MatrizConfusao(numSaida);

**double** dadosEntrada[], dadosSaida[];

**for** (Dados d : dados) {

dadosEntrada = Arrays.copyOfRange(d.getDados(), 0, d.getDados().length - 1);

dadosSaida = CSVFile.classes.get((**int**) d.getDados()[d.getDados().length - 1]).clone();

**this**.testar(dadosEntrada, dadosSaida);

}

}

/\*\*

\* Este método {@code testar()} executa o processo de teste para uma única

\* entrada e uma única saída esperada.

\*

\* **@param** dadoEntrada Conjunto de dados da entrada.

\* **@param** dadoSaida Conjunto de dados da saída.

\*/

**private** **void** testar(**double**[] dadoEntrada, **double**[] dadoSaida) {

**int** i = 0;

**for** (Neuronio n : **this**.camadas[Camada.ENTRADA].getNeuronios()) {

n.setNet(dadoEntrada[i]);

n.setSaida(dadoEntrada[i]);

i++;

}

**for** (Neuronio n : **this**.camadas[Camada.OCULTA].getNeuronios()) {

**double** soma = 0;

**for** (Neuronio no : **this**.camadas[Camada.ENTRADA].getNeuronios()) {

soma += n.getPeso(no.getId()) \* no.getSaida();

}

n.setNet(soma);

n.setSaida(FuncaoTransferencia.funcao(soma));

}

**for** (Neuronio n : **this**.camadas[Camada.SAIDA].getNeuronios()) {

**double** soma = 0;

**for** (Neuronio no : **this**.camadas[Camada.OCULTA].getNeuronios()) {

soma += n.getPeso(no.getId()) \* no.getSaida();

}

n.setNet(soma);

n.setSaida(FuncaoTransferencia.funcao(soma));

}

**this**.atualizarMatrizConfusao(dadoSaida);

}

/\*\*

\* Este método {@code atualizarMatrizConfusao} classifica a saída da rede e

\* atribui à matriz de confusão.

\*

\* **@param** dadoSaida Conjunto da saída esperada.

\*/

**public** **void** atualizarMatrizConfusao(**double**[] dadoSaida) {

**int** real, obtido;

real = **this**.maior(dadoSaida);

obtido = **this**.maior(camadas[Camada.SAIDA].saidas());

matrizConfusao.add(real, obtido);

}

/\*\*

\* Este método {@code maior} retorna a posição do maior elemento do

\* conjunto.

\*

\* **@param** dadoSaida Conjunto de valores.

\* **@return** Retorna posição do maior elemento.

\*/

**private** **int** maior(**double**[] dadoSaida) {

**double** maior = dadoSaida[0];

**int** posicao = 0;

**for** (**int** a = 0; a < dadoSaida.length; a++) {

**if** (dadoSaida[a] > maior) {

maior = dadoSaida[a];

posicao = a;

}

}

**return** posicao;

}

/\*\*

\* Este método {@code treinamento} aplica o treinamento na rede a partir de

\* um conjunto de dados.

\*

\* **@param** dados Conjunto de dados.

\*/

**public** **void** treinamento(ArrayList<Dados> dados) {

**int** numIteracoes = 0;

**double** dadosEntrada[], dadosSaida[];

**boolean** flag;

**do** {

flag = **false**;

**for** (Dados d : dados) {

dadosEntrada = Arrays.copyOfRange(d.getDados(), 0, d.getDados().length - 1);

dadosSaida = CSVFile.classes.get((**int**) d.getDados()[d.getDados().length - 1]).clone();

**this**.treinar(dadosEntrada, dadosSaida);

**if** (**this**.pararTreinamento(opcaoErro, numIteracoes)) {

flag = **true**;

}

}

numIteracoes++;

} **while** (flag);

}

/\*\*

\* Aplica o treinamento para cada entrada e saída esperada, individualmente.

\*

\* **@param** dadoEntrada Conjunto de entrada.

\* **@param** dadoSaida Conjunto de saída esperado.

\*/

**private** **void** treinar(**double**[] dadoEntrada, **double**[] dadoSaida) {

**int** count = 0;

**for** (Neuronio n : **this**.camadas[Camada.ENTRADA].getNeuronios()) {

n.setNet(dadoEntrada[count]);

n.setSaida(dadoEntrada[count]);

count++;

}

**for** (Camada camada : **this**.camadas) {

**if** (camada.temCamadaAnterior()) {

**for** (Neuronio no : camada.getNeuronios()) {

**double** soma = 0;

**for** (Neuronio n : camada.getAnterior().getNeuronios()) {

soma += n.getSaida() \* no.getPeso(n.getId());

}

no.setNet(soma);

no.setSaida(FuncaoTransferencia.funcao(no.getNet()));

}

}

}

count = 0;

**for** (Neuronio n : **this**.camadas[Camada.SAIDA].getNeuronios()) {

n.setErro((dadoSaida[count] - n.getSaida()) \* FuncaoTransferencia.derivada(n.getNet()));

count++;

}

**for** (Neuronio no : camadas[Camada.OCULTA].getNeuronios()) {

**double** erro = 0;

**for** (Neuronio n : camadas[Camada.SAIDA].getNeuronios()) {

erro += n.getErro() \* n.getPeso(no.getId());

}

no.setErro(erro \* FuncaoTransferencia.derivada(no.getNet()));

}

**for** (Camada camada : **this**.camadas) {

**if** (camada.temCamadaAnterior()) {

**for** (Neuronio no : camada.getNeuronios()) {

**for** (Neuronio n : camada.getAnterior().getNeuronios()) {

no.setPeso(n.getId(), no.getPeso(n.getId()) + taxaAprendizagem \* no.getErro() \* n.getSaida());

}

}

}

}

}

/\*\*

\* Este método verifica a condição de término do treinamento.

\*

\* **@param** opcaoErro Define a operação entre erro máximo e número de

\* iterações

\* **@param** numIteracoes Total de iterações executadas até o momento.

\* **@return** Retorna {@code true} para continuar repetindo, caso contrário

\* {@code false}.

\*/

**private** **boolean** pararTreinamento(**boolean** opcaoErro, **int** numIteracoes) {

**boolean** flag = **false**;

**if** (opcaoErro) {

**double** soma = 0;

**for** (Neuronio n : **this**.camadas[Camada.SAIDA].getNeuronios()) {

soma += (n.getErro() \* n.getErro());

}

soma \*= 0.5;

**if** (soma > **this**.erroMax) {

flag = **true**;

}

} **else** {

**if** (numIteracoes < **this**.numIteracoesMax) {

flag = **true**;

}

}

**return** (flag);

}

@Override

**public** String toString() {

StringBuilder s = **new** StringBuilder("");

**for** (Camada c : camadas) {

s.append("\n").append(c);

}

s.append("\n");

**return** s.toString();

}

/\*\*

\* Recupera a camada pela identificação.

\*

\* **@param** i Número da camada.

\* **@return** Retorna a instancia da camada escolhida.

\*/

**public** Camada getCamada(**int** i) {

**return** camadas[i];

}

/\*\*

\* Recupera a matriz de confusão da rede.

\*

\* **@return** Retorna a instancia atual da matriz de confusão.

\*/

**public** MatrizConfusao getMatrizConfusao() {

**return** matrizConfusao;

}

**public** **int** getNumOculta() {

**return** numOculta;

}

**public** **void** setNumOculta(**int** numOculta) {

**this**.numOculta = numOculta;

}

**public** **void** modificarCamadaOculta(**int** num) {

**this**.camadas[Camada.OCULTA].modificarCamadaOculta(num);

**for** (Neuronio noSaida : **this**.camadas[Camada.SAIDA].getNeuronios()) {

**for** (Neuronio noOculta : **this**.camadas[Camada.OCULTA].getNeuronios()) {

**if** (noSaida.getPesos().get(noOculta.getId()) == **null**) {

**if** (BackPropagation.*tipoPeso* == BackPropagation.*PESO\_GAUSSIANO*) {

noSaida.getPesos().put(noOculta.getId(), BackPropagation.*random*.nextGaussian());

} **else** {

noSaida.getPesos().put(noOculta.getId(), Math.*random*() \* (Math.*random*() > 0.5 ? 1 : -1));

}

}

}

}

}

}

**public** **class** Camada {

**public** **static** **final** **int** ***ENTRADA*** = 0;

**public** **static** **final** **int** ***OCULTA*** = 1;

**public** **static** **final** **int** ***SAIDA*** = 2;

**private** Neuronio[] nos;

**private** Camada anterior;

**public** Camada(**int** tamanho, Camada anterior) {

**this**.nos = **new** Neuronio[tamanho];

**this**.anterior = anterior;

**for** (**int** i = 0; i < **this**.nos.length; i++) {

nos[i] = **new** Neuronio(BackPropagation.contadorNeuronio++, 0);

nos[i].inicializarPesos(anterior);

}

}

**public** **void** modificarCamadaOculta(**int** numNeuronios) {

Neuronio[] novos = **new** Neuronio[numNeuronios];

**int** i = 0;

**for** (; i < numNeuronios && i < **this**.nos.length; i++) {

novos[i] = **this**.nos[i];

}

**for**(; i < numNeuronios; i++){

novos[i] = **new** Neuronio(BackPropagation.contadorNeuronio++, 0);

novos[i].inicializarPesos(anterior);

}

**this**.nos = novos;

}

**public** Neuronio[] getNeuronios() {

**return** nos;

}

**public** Neuronio getNeuronio(**int** id) {

**for** (**int** i = 0; i < **this**.nos.length; i++) {

**if** (nos[i].getId() == id) {

**return** (nos[i]);

}

}

**return** (**null**);

}

/\*\*

\* Verifica se a camada atual possui camada antecedente.

\*

\* **@return**

\*/

**public** **boolean** temCamadaAnterior() {

**return** **this**.anterior != **null**;

}

**public** Camada getAnterior() {

**return** **this**.anterior;

}

@Override

**public** String toString() {

StringBuilder s = **new** StringBuilder("");

**for** (Neuronio no : nos) {

s.append(no);

}

**return** s.toString();

}

/\*\*

\* Este método extrai o conjunto de valores da saída dos neurônios desta

\* camada.

\*

\* **@return** Retorna um array com os valores de saída.

\*/

**public** **double**[] saidas() {

**double**[] lista = **new** **double**[**this**.getNeuronios().length];

**int** count = 0;

**for** (Neuronio n : **this**.getNeuronios()) {

lista[count] = n.getSaida();

count++;

}

**return** lista;

}

}

**public** **class** Neuronio {

**private** **int** id;

**private** **double** net;

**private** **double** saida;

**private** **double** erro;

**private** HashMap<Integer, Double> pesos;

**public** Neuronio(**int** id, **double** net) {

**this**.id = id;

**this**.net = net;

**this**.pesos = **new** HashMap<>();

**this**.saida = 0;

}

/\*\*

\* Inicializa os pesos do neurônio de forma aletória ou gaussiana.

\* **@param** anterior Refere-se a camada antecedente na rede.

\*/

**public** **void** inicializarPesos(Camada anterior) {

**if** (anterior == **null**) {

**return**;

}

**for** (Neuronio n : anterior.getNeuronios()) {

**if** (BackPropagation.tipoPeso == BackPropagation.PESO\_GAUSSIANO) {

**this**.pesos.put(n.getId(), BackPropagation.random.nextGaussian());

} **else** {

**this**.pesos.put(n.getId(), Math.*random*() \* (Math.*random*() > 0.5 ? 1 : -1));

}

}

}

**public** HashMap<Integer, Double> getPesos() {

**return** pesos;

}

**public** **void** setPesos(HashMap<Integer, Double> pesos) {

**this**.pesos = pesos;

}

**public** **double** getPeso(**int** id) {

**return** pesos.get(id);

}

**public** **void** setPeso(**int** numPeso, **double** valor) {

**if** (pesos.containsKey(numPeso)) {

pesos.replace(numPeso, valor);

} **else** {

pesos.put(numPeso, valor);

}

}

**public** **double** getErro() {

**return** erro;

}

**public** **void** setErro(**double** erro) {

**this**.erro = erro;

}

**public** **void** setNet(**double** net) {

**this**.net = net;

}

**public** **double** getSaida() {

**return** saida;

}

**public** **void** setSaida(**double** saida) {

**this**.saida = saida;

}

**public** **double** getNet() {

**return** net;

}

**public** **int** getId() {

**return** id;

}

@Override

**public** String toString() {

StringBuilder s = **new** StringBuilder("");

s.append("(").append(**this**.net).append(") => [");

**for** (Integer i : **this**.pesos.keySet()) {

s.append(String.*format*("%.2f", pesos.get(i))).append(",");

}

**if** (s.toString().charAt(s.length() - 1) == '[') {

s.append(",");

}

s.append("\b] ");

**return** s.toString();

}

}

**public** **class** Normalizacao {

**public** **static** Dados *max*;

**public** **static** Dados *min*;

**public** **static** **int** *funcaoTreinamento*;

/\*\*

\* Este método obtém informações do conjunto de dados, como valores mínimo e máximo.

\* **@param** treinamento

\* **@param** funcaoTreinamento

\*/

**public** **static** **void** analisar(ArrayList<Dados> treinamento, **int** funcaoTreinamento){

Normalizacao.*funcaoTreinamento* = funcaoTreinamento;

Dados max = treinamento.get(0).clone();

Dados min = treinamento.get(0).clone();

//Os dois primeiros FOR calculam o valor maximo de cada coluna de elementos, considerando Treinamento e Teste

**for**(Dados d: treinamento){

**for**(**int** i = 0; i < d.getDados().length-1; i++){

**if** (d.get(i) > max.get(i)) max.set(i, d.get(i));

**else** **if** (d.get(i) < min.get(i)) min.set(i, d.get(i));

}

}

Normalizacao.*max* = max;

Normalizacao.*min* = min;

}

/\*\*

\* Aplica a normalização no conjunto de dados com base no domínio de dados da função de transferência escolhida para a rede,

\* **@param** conjunto Conjunto de dados.

\* **@return** Retorna o conjunto de dados normalizado.

\*/

**public** **static** ArrayList<Dados> normalizar(ArrayList<Dados> conjunto){

ArrayList<Dados> result = **new** ArrayList<>();

**double** a, b;

**if** (*funcaoTreinamento* == FuncaoTransferencia.LOGISTICA) {

a = 0; b = 1;

}

**else**{

a = -1; b = 1;

}

**double** valor;

//Aplica a normalização com base no valor maximo obtido anteriormente

**for**(Dados d: conjunto){

Dados novo = **new** Dados(d.getDados().length);

**for**(**int** i = 0; i < d.getDados().length-1; i++){

valor = ((d.get(i) - *min*.get(i))\*(b-a))/(*max*.get(i) - *min*.get(i)) + a;

novo.set(i, valor);

//novo.set(i, d.get(i)/max.get(i));

}

novo.set( d.getDados().length-1, d.get(d.getDados().length-1));

result.add(novo);

}

**return** result;

}

}

**public** **class** FuncaoTransferencia {

**public** **static** **final** **int** ***LINEAR*** = 1;

**public** **static** **final** **int** ***LOGISTICA*** = 2;

**public** **static** **final** **int** ***TANGENTE\_HIPERBOLICA*** = 3;

**public** **static** **int** *FUNCAO\_ATUAL*;

**public** **static** **double** funcao(**double** x){

**switch**(*FUNCAO\_ATUAL*){

**case** ***LINEAR***:

**return** x;

**case** ***LOGISTICA***:

**return** 1/(1+Math.*exp*(-x));

**case** ***TANGENTE\_HIPERBOLICA***:

**return** (1-Math.*exp*(-2\*x))/(1+Math.*exp*(-2\*x));

**default**:

**return** 1;

}

}

**public** **static** **double** derivada(**double** x){

**switch**(*FUNCAO\_ATUAL*){

**case** ***LINEAR***:

**return** 1;

**case** ***LOGISTICA***:

**return** *funcao*(x)\*(1-*funcao*(x));

**case** ***TANGENTE\_HIPERBOLICA***:

**return** 1 - Math.*pow*(*funcao*(x), 2);

**default**:

**return** 1;

}

}

}

**public** **class** MatrizConfusao {

**private** **int** dimensao;

**private** **int**[][] dados;

**public** MatrizConfusao(**int** dimensao){

**this**.dimensao = dimensao;

**this**.dados = **new** **int**[dimensao][dimensao];

}

**public** **void** add(**int** real, **int** obtida){

dados[real][obtida]++;

}

@Override

**public** String toString() {

String s = "\n";

**for**(**int** i = 0; i < **this**.dimensao; i++){

**for**(**int** j = 0; j < **this**.dimensao; j++){

s += dados[i][j] + " ";

}

s += "\n";

}

**return** s;

}

}

# **REFERÊNCIAS BIBLIOGRÁFICAS**

[1] ARTERO, Almir Olivette. Inteligência Artificial: teórica e prática. 1ª Edição, São Paulo: Editora Livraria da Física, 2009.