BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RELATÓRIO DO TRABALHO PRÁTICO II

INTELIGÊNCIA ARTIFICIAL

PROF. DR. ALMIR OLIVETTE ARTERO

EYMAR FERRARIO DE LIMA

MATHEUS PRACHEDES BATISTA

PRESIDENTE PRUDENTE

2017

# 1 – Execução do Software

## 1.1 – Carregando os Testes

Para iniciar a execução do Software, primeiramente é necessário carregar os arquivos que contém o conjunto de Treinamento e o conjunto de Testes onde ambas as opções se encontram na janela principal do programa no menu “Conjuntos” (o programa apenas reconhece e carrega arquivos com extensão .csv). Após carregar ambos os arquivos, para agilizar a convergência e deixar todos os atributos na mesma escala [0,1], é importante normalizar os dois conjuntos carregados e, para realizar esta ação, basta selecionar a opção normalizar Conjuntos que se encontra também no menu “conjuntos” (não é possível normalizar os conjuntos até que ambos estejam carregados). A Figura 1 ilustra o menu que contém as opções citadas anteriormente.

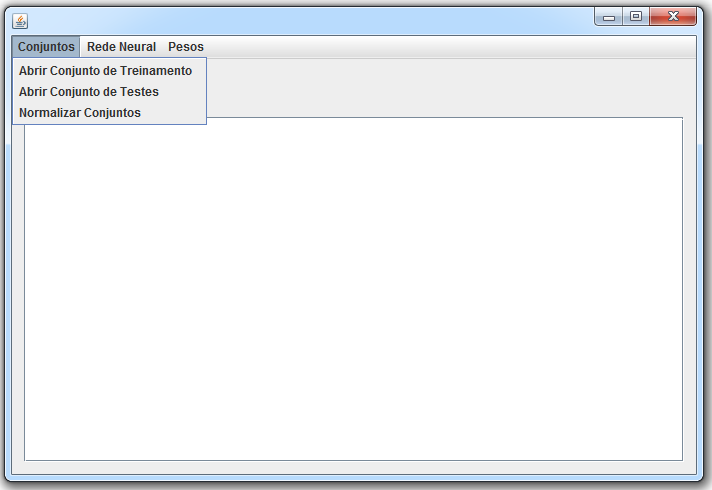


Figura - Menu de Conjuntos.

## 1.2 – Criação da Rede Neural

Após carregar os conjuntos e normaliza-los é necessário criar a Rede Neural e, para realizar esta ação basta acessar o menu “Rede Neural” e escolher a opção “Criar Rede Neural”. A Figura 2 ilustra o menu “Rede Neural”.

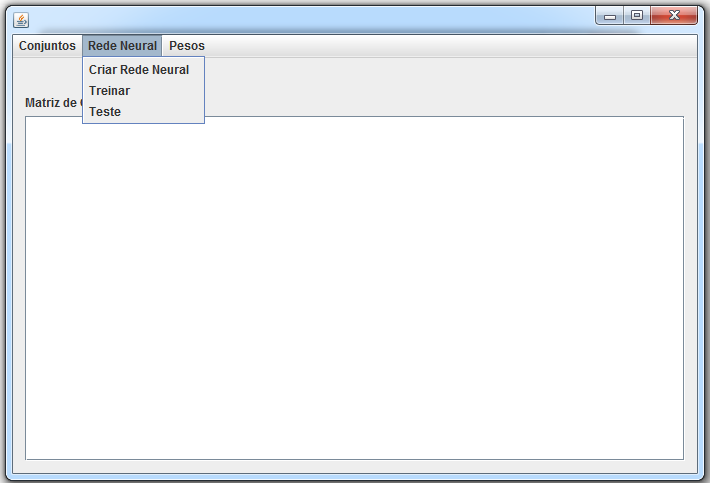


Figura 2 - Menu de Rede Neural

Ao selecionar a Opção de “Criar Rede Neural”, uma nova janela irá se abrir onde o usuário poderá selecionar a quantidade de camadas ocultas, o número de neurônios em cada camada oculta e a função de propagação (podendo ser Tangente Hiperbólica ou Logística). A quantidade de neurônios na camada de entrada e de saída é configurada automaticamente de acordo com os conjuntos de entrada. Após configurar a Rede Neural Basta clicar no botão “Criar”. A Figura 3 ilustra a interface de criação de redes neurais.

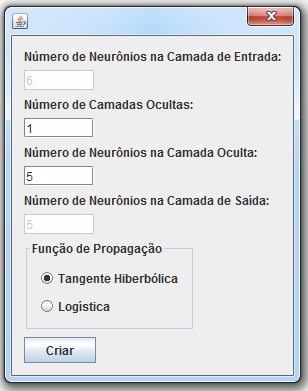


Figura - Interface de Criação de uma Rede Neural

## 1.3 – Ajuste de Pesos

O software gera os pesos aleatoriamente, porém, caso o usuário julgue necessário, ele pode alterar todos os pesos individualmente selecionando a opção “Ajustas Pesos” no menu “Pesos” ilustrados pela Figura 4.

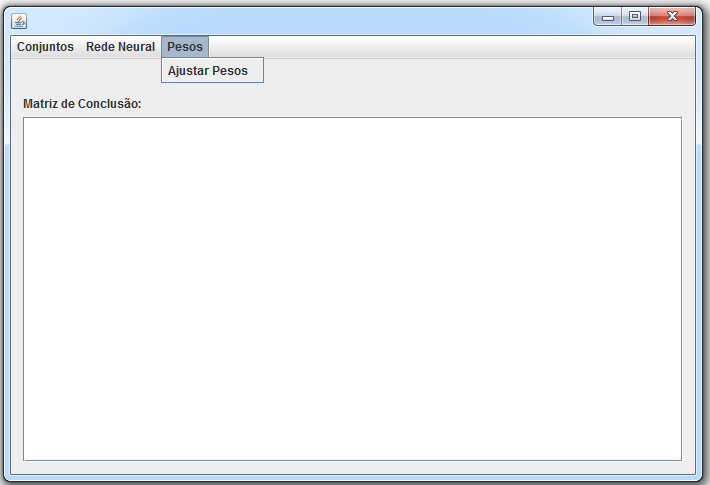


Figura - Menu de Pesos

Após selecionar a opção de ajusta pesos, uma nova janela será aberta aonde o usuário poderá escolher a camada, o neurônio e qual dos pesos ele deseja alterar, após selecionar essas 3 opções, bastar inserir o novo peso no campo “Valor do Peso” e finalizar clicando no botão “Alterar”. A Figura 5 ilustra a interface de alteração de pesos.

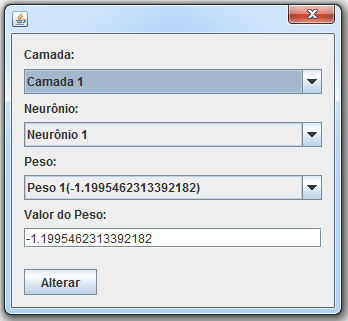


Figura - Interface de Alteração de Pesos

## 1.4 – Treinamento da Cadeia

Para treinar a cadeia é necessário selecionar a opção “Treinar” que se encontra no menu “Rede Neural” (ambos ilustrados pela Figura 2) e, após selecionar esta opção, uma nova janela irá abrir para que o usuário selecione a taxa de aprendizado, o limiar de erro e o número de iterações (caso seja necessário usar números reais, o programa apenas aceita a fração dividida por “.” e não por “,”). Após entrar com os dados necessários basta clicar no botão “Treinar”. A Figura 6 ilustra a interface de treinamento.

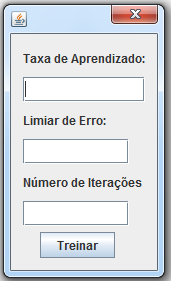


Figura - Interface de Treinamento da Cadeia

## 1.5 – Teste da Cadeia

Para testar a cadeia, apenas é necessário selecionar a opção “Testar” que se encontra no menu “Rede Neural” (ambos ilustrados pela Figura 2) e, após selecionar a opção, o resultado ilustrado através da Matriz de Confusão será carregado na interface principal do programa. A Figura 7 ilustra como a matriz de confusão é exibida para o usuário.

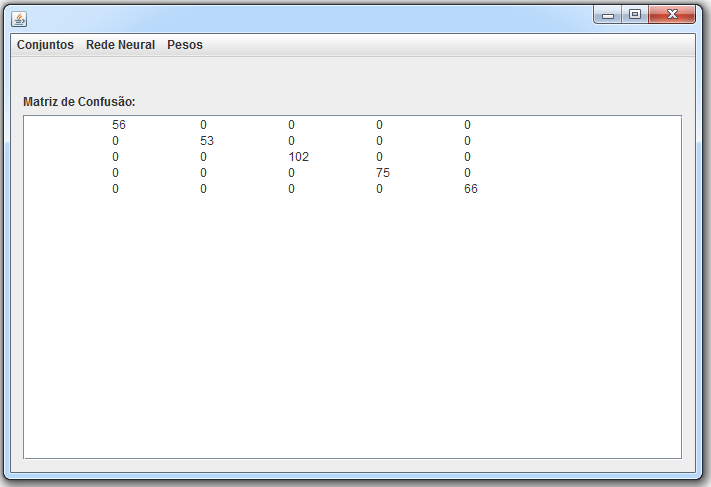


Figura 7 - Matriz de Confusão

# 2 – Resultado dos Testes

Ao utilizar o conjunto de testes e treinamento fornecidos em aula, normalizamos os dois conjuntos e criamos apenas 1 camada oculta com 5 neurônios (todos os pesos foram gerados aleatoriamente). Ao utilizar tanto a Função de Tangente Hiperbólica quanto a Função de Logística, utilizando os parâmetros para treinar:

* Taxa de Aprendizado: 0.1
* Limiar de Erro: 0.01
* Número de Iterações: 1000

Ambas as Funções apresentaram sucesso no treinamento, de tal forma que na hora de executar os testes apresentar a mesma 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 |

Ao tentar realiza o mesmo conjunto de testes porem sem normalizar os conjuntos, ao utilizar a função de Tangente Hiperbólica e Função Logística obtivemos respectivamente as seguintes matrizes:

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

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

Logo concluímos que a quantidade de camadas e neurônios não é suficiente para aprender o treinamento ou talvez seja necessário aumentar o número de iterações pois sem normalização a convergência é mais devagar.

# 3 – Código Fonte

## 3.1 – Classe Abstrata de uma Função

/\*Classe abstrata usada para representar uma função de propagação; \*/

public abstract class Função {

/\*\*Calcula f(x)\*/

public abstract double compute(double x);

/\*\*Calcula f'(x)\*/

public abstract double derivada(double x);

/\*\*Menor valor da imagem da função, usada para definir a saida das camadas\*\*/

public abstract double menorValorImagem();

/\*\*Maior valor da imagem da função, usada para definir a saida das camadas\*\*/

public abstract double maiorValorImagem();

}

## 3.2 – Função Logística

/\* Função logistica para a função de propagação\*/

public class Logistica extends Função{

/\*\*Calcula f(x)\*/

@Override

public double compute(double x) {

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

}

/\*\*Calcula f'(x)\*/

@Override

public double derivada(double x) {

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

}

@Override

/\*\*Menor valor da imagem da função, usada para definir a saida das camadas\*\*/

public double menorValorImagem() {

return 0;

}

@Override

/\*\*Maior valor da imagem da função, usada para definir a saida das camadas\*\*/

public double maiorValorImagem() {

return 1;

}

}

## 3.3 – Função Tangente Hiperbólica

public class TangenteHiberbolica extends Função {

@Override

public double compute(double x) {

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

}

@Override

public double derivada(double x) {

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

}

@Override

/\*Menor valor da imagem da função, usada para definir a saida das camadas\*/

public double menorValorImagem() {

return -1;

}

@Override

/\*Maior valor da imagem da função, usada para definir a saida das camadas\*/

public double maiorValorImagem() {

return 1;

}

}

## 3.4 – Instância de Dados

/\*Representa uma instancia de dados com os atributos e o rotulo da classe\*/

public class Instancia {

public double[] atributos;

public String classe;

public Instancia(double[] atributos, String classe){

this.atributos = atributos;

this.classe = classe;

}

/\*\*

\* Normaliza os atributos desta instancia de modo que os atributos do conjunto

\* estejam no intervalo [limiteMin, limiteMax]

\* @param min menor valor de cada atributo no conjunto

\* @param max maior valor de cada atributos no conjunto

\* @param limiteMin limiteMin da normalização

\* @param limiteMax limiteMax da normalização

\*/

protected void normalizar(double[] min, double[] max,double limiteMin, double limiteMax) {

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

atributos[i] = ((atributos[i]-min[i])/(max[i]-min[i]))\*(limiteMax-limiteMin)+limiteMin;

}

}

}

## 3.5 – Conjunto de Instâncias

/\*Classe que representa um conjunto de instancias\*/

public class Instancias {

public static final int NORMALIZAR\_ENTRE\_0E1 = 1;

public static final int NORMALIZAR\_ENTRE\_M1E1 = 2;

/\*Instancias do conjunto\*/

private ArrayList<Instancia> instancias = new ArrayList<>();

/\*\*Saida esperada da rede para cada classe

\*Armazenada para evitar calcular a saida toda hora\*/

private HashMap<String, Double[]> mapeamentoSaidas = new HashMap<>(10);

/\*Classes do conjunto de instancias\*/

private ArrayList<String> classes = new ArrayList<>();

private int numClasses = 0;

private int numAtributos = 0;

public Instancias(){}

public Instancia getInstancia(int i) {

return instancias.get(i);

}

public int size() {

return instancias.size();

}

/\*Retorna os atributos da instancia i\*/

public double[] getAtributos(int i) {

return instancias.get(i).atributos;

}

/\*Embaralha o conjunto de instancias\*/

public void embaralhar() {

ArrayList<Instancia> novaInstancias = new ArrayList<>();

Random r = new Random();

while (instancias.size() > 0) {

novaInstancias.add(instancias.remove(r.nextInt(instancias.size())));

}

instancias = novaInstancias;

}

/\*\*Retorna a saida esperada da rede para a instancia i\*\*/

public Double[] getSaida(int i) {

return mapeamentoSaidas.get(instancias.get(i).classe);

}

/\*\*Abre o arquivo especificado pelo parametro e carrega as instancias\*\*/

public boolean abrirArquivo(File arquivo) {

instancias.clear();

BufferedReader reader = null;

try {

int i, cont = 0;

String classe;

reader = new BufferedReader(new FileReader(arquivo));

String linha;

String[] tokens;

tokens = reader.readLine().split(",");

numAtributos = tokens.length - 1;

while ((linha = reader.readLine()) != null) {

double[] atributos = new double[numAtributos];

tokens = linha.split(",");

for (i = 0; i < numAtributos; i++) {

atributos[i] = Double.valueOf(tokens[i]);

}

classe = tokens[i];

instancias.add(new Instancia(atributos, classe));

mapeamentoSaidas.putIfAbsent(classe, null);

}

numClasses = mapeamentoSaidas.size();

reader.close();

} catch (IOException ex) {

instancias.clear();

if (reader != null) {

try {

reader.close();

} catch (IOException ex1) {

}

}

}

return instancias.isEmpty();

}

/\*\*

\* Recupera o minimo e maximo de cada atributo do conjunto 'c' e retorna nos vetores

\* de entrada

\*/

private static void recuperaMinMax(double[] min, double[]max, Instancias c){

if(c==null)return;

for(Instancia i : c.instancias){

double[] atributos = i.atributos;

for (int cont = 0; cont < atributos.length; cont++) {

if (atributos[cont] < min[cont]) {

min[cont] = atributos[cont];

}

if (atributos[cont] > max[cont]) {

max[cont] = atributos[cont];

}

}

}

}

/\*\*

\* Normaliza cada instancia do conjunto 'c'

\* @param min menor valor de cada atributo do conjunto

\* @param max maior valor de cada stributo do conjunto

\* @param limiteMin menor valor do intervalo após normalização

\* @param limiteMax maior valor do intervalo após normalização

\* @param c conjunto que será normalizado

\*/

private static void normaliza(double[] min, double[] max, double limiteMin, double limiteMax, Instancias c){

if(c == null)return;

for(Instancia i : c.instancias){

i.normalizar(min, max, limiteMin, limiteMax);

}

}

/\*\*

\* Normaliza este conjunto para que os atributos estejam entre [limiteMin,limiteMax]

\* @param c Conjunto que será normalizado junto com este.

\*/

public void normalizar(double limiteMin, double limiteMax, Instancias c) {

int numAtr = instancias.get(0).atributos.length;

double min[] = new double[numAtr];

double max[] = new double[numAtr];

for (int i = 0; i < numAtr; i++) {

min[i] = Double.MAX\_VALUE;

max[i] = -Double.MAX\_VALUE;

}

recuperaMinMax(min, max, this);

recuperaMinMax(min,max,c);

normaliza(min,max,limiteMin,limiteMax,c);

normaliza(min,max,limiteMin,limiteMax,this);

}

/\*\*

\* Define as saidas esperadas pela rede para cada classe do conjunto.

\* @param funçãoPropagação A função é usada para definir o minimo e o maximo da saida.

\*/

public void definirSaidasClasses(Função funçãoPropagação) {

Set<String> valorClasses = mapeamentoSaidas.keySet();

int cont = 0;

for (String i : valorClasses) {

Double[] saidaClasses = new Double[mapeamentoSaidas.size()];

for (int j = 0; j < mapeamentoSaidas.size(); j++) {

saidaClasses[j] = funçãoPropagação.menorValorImagem();

}

saidaClasses[cont++] = funçãoPropagação.maiorValorImagem();

mapeamentoSaidas.put(i, saidaClasses);

classes.add(i);

}

}

public int getNumClasses() {

return numClasses;

}

public int getIndexClasse(int i) {

return classes.indexOf(instancias.get(i).classe);

}

}

## 3.6 – Classe Neurônio

/\*Classe que representa um neuronio da rede\*/

public class Neuronio {

/\*\*Função de propagação utilizada neste neuronio\*/

protected Função funçãoPropagação;

/\*\*Pesos das entradas deste neuronio\*/

protected double[] pesos;

/\*\*Net calculado por este neuronios\*/

protected double net;

/\*\*Valor calculada da propagação deste neuronio\*/

protected double propagação;

/\*\*Erro calculado por este neuronio\*/

protected double erro;

/\*\*Valor dos sinais que veio como entrada para este neuronio\*/

protected double[] inputs;

/\*\*

\* Constroi este neuronio com a determinada função de propagação.

\* O parametro numPesos deve bater com o numero de neuronios da camada anterior.

\* Os pesos são setados aleatoriamente de acordo com uma distribuição gaussiana

\* com media 0 e desvio padrão 0

\* @param funçãoPropagação função de propagação deste neuronio

\* @param numPesos numero de pesos que este neuronio possui

\*/

public Neuronio(Função funçãoPropagação, int numPesos){

this.funçãoPropagação = funçãoPropagação;

pesos = new double[numPesos];

for(int i=0;i<numPesos;i++){

pesos[i] = RedeNeural.rand.nextGaussian();

}

}

/\*\*

\* Calcula o valor de propagação deste neuronio.

\* @param entradas sinais de entrada

\* @return retonar o valor de propagação.

\*/

public double calcularPropagação(double[] entradas){

net = 0;

inputs = entradas;

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

net += pesos[i]\*entradas[i];

}

propagação = funçãoPropagação.compute(net);

return propagação;

}

/\*\*

\* Calcula o erro deste neuronio.

\* Deve ser usado para neuronios da camada oculta.

\* @param soma

\* @return

\*/

public double calculaErro(double soma) {

erro = funçãoPropagação.derivada(net)\*soma;

return erro;

}

/\*\*

\* Calcula o erro deste neuronio.

\* Deve ser usado para neuronios da camada de saida.

\* @param desejado valor desejado deste neuronio de saida.

\* @return retorna o erro deste neuronio.

\*/

public double calculaErroSaida(double desejado) {

erro = (desejado - propagação)\*funçãoPropagação.derivada(net);

return erro;

}

/\*\*

\* Ajusta os pesos deste neuronio de acordo com a taxa do parametro e com

\* os erros já calclados.

\* Deve ser chamado após os erros deste neuronio terem sidos calculados e

\* de preferencia após toda a rede ou a camada anterior a esta ter calculado

\* seus erros.

\* @param taxaAprendizado taxa de aprendizado usado para ajustar os pesos.

\*/

public void ajustarPesos(double taxaAprendizado) {

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

pesos[i] = pesos[i] + taxaAprendizado\*erro\*inputs[i];

}

}

}

## 3.7 – Camada de Processamento

/\*\*

\* Representa uma cada de processamento, podendo ser tanto uma camada oculta

\* quanto uma camada de saida.

\*/

public class CamadaProcessamento {

/\*Conjunto de neuronios desta camada\*/

protected Neuronio[] neuronios;

/\*Armazena os erros calculados pelos neuronios desta camada para uso futuro\*/

protected double[] erros;

/\*\*

\* Constroi a camada de neuronios e instancia os neuronios, setando os pesos

\* aleatoriamente seguinda uma distribuição gaussiana com média 0 e desvio

\* padão 1

\* @param numNeuroniosOcultos Numero de neuronios desta camada

\* @param propagação Função de propagação usadas nos neuronios

\* @param numPesos Numero de pesos que cada neuronios deve ter. Este numero

\* Deve corresponder com o numero de neuronios da camada anterior a esta.

\*/

public CamadaProcessamento(int numNeuroniosOcultos, Função propagação, int numPesos) {

neuronios = new Neuronio[numNeuroniosOcultos];

erros = new double[numNeuroniosOcultos];

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

neuronios[i] = new Neuronio(propagação,numPesos);

}

}

/\*\*

\* Processo de feedFoward desta camada.

\* @param inputs entrada recebida por esta camada

\* @return retonar os sinais propagados por esta camada

\*/

public double[] feedFoward(double[] inputs) {

double[] sinais = new double[neuronios.length];

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

sinais[i] = neuronios[i].calcularPropagação(inputs);

}

return sinais;

}

/\*\*

\* Calcula os erros destes neuronios considerandos as saidas desejadas.

\* Esse método deve ser chamado apenas para a camada de saida.

\* @param outputEsperado saidas esperadas por essa camada

\*/

public void calculaErros(Double[] outputEsperado) {

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

erros[i] = neuronios[i].calculaErroSaida(outputEsperado[i]);

}

}

public double[] getErros(){

return erros;

}

/\*\*

\* Calcula os erros destes neuronios considerandos os erros propagados pela

\* camada posterior a esta.

\* Esse método deve ser chamado apenas para camadas ocultas.

\* @param saida Camada posterior a esta

\*/

public void calculaErros(CamadaProcessamento saida) {

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

double soma = saida.somaErros(i);

neuronios[i].calculaErro(soma);

}

}

/\*\*

\* Retorna a soma dos erros destes neuronios "i" multiplicados pelo peso Wij

\* Formula: Σ (de i=0 até m)(erro(i)\*Wij)

\* @param j indice do neuronio da camada oculta anterior para o qual está sendo

\* calculando o erro

\* @return retorna a soma

\*/

private double somaErros(int j) {

double soma =0;

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

soma+=neuronios[i].erro\*neuronios[i].pesos[j];

}

return soma;

}

/\*\*

\* Ajusta os pesos desse neuronio utilizando a taxa de aprendizado vindo

\* pelo parametro.

\* Este metodo deve ser chamado após está camada ter calculado seus erros e

\* de preferencia após toda a rede ter calculada seus erros

\* @param taxaAprendizado Taxa de aprendizado utilizado para ajustar os pesos

\*/

public void ajustarPesos(double taxaAprendizado) {

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

neuronios[i].ajustarPesos(taxaAprendizado);

}

}

/\*\*

\* Retorna metade da soma dos quadrados dos erros dos neuronios desta camada

\* Formula: 1/2 \* Σ (de i=0 até o)(erro(i)^2)

\* Deve ser chamado apenas para a camada de saida.

\* @return Retorna metade da soma.

\*/

public double erroRede() {

double soma = 0;

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

soma+=Math.pow(neuronios[i].erro, 2);

}

return soma/2;

}

/\*\*

\* Retorna os pesos do neuronio

\* @param neuronio Neuronio no qual será retornado os pesos

\*/

public double[] getPesos(int neuronio) {

return neuronios[neuronio].pesos;

}

}

## 3.8 – Rede Neural

public class RedeNeural {

/\*\*Usado para gerar os pesos aleatorios dos neuronios\*/

public static final Random rand = new Random();

/\*\*Função de propagação usada pelos neuronios da rede\*/

protected Função funçãoPropagação;

/\*\*Camada de neuronios de entrada\*/

protected CamadaEntrada entrada;

/\*\*Camadas de neuronios ocultos usados para o processamento\*/

protected CamadaProcessamento[] oculta;

/\*\*Camada de neuronios na saida\*/

protected CamadaProcessamento saida;

/\*\*Taxa de aprendizado para esta rede\*/

protected double taxaAprendizado = 0.1;

/\*\*

\* Numero de iterações para parar o treinamento caso a rede não convergir.

\* Caso o erro da rede seja menor que <code>limiar</code> o treinamento para

\* antes

\*/

protected int numIteraçõesLimite = 20000;

/\*\*Limiar usado para parar o treinamento antes do limite de iterações.

\* O treinamento para caso o valor do erro seja menor que o limiar

\*/

protected double limiar = 0.000001f;

/\*\*

\* Cria a rede neural e instancia as camadas e os neuronios da rede.

\* Os pesos dos neuronios são decididos aleatoriamente seguindo uma distribuição

\* gaussiana com media 0 e desvio padrão 1.

\*

\* As camadas ocultas possuem a mesma quantidade de neuronios.

\*

\* @param numNeuroniosEntrada Numero de neuronios na camada de entrada

\* @param numNeuroniosOcultos Numero de neuronios na camada oculta

\* @param numCamadasOcultas Numero de camadas ocultas

\* @param numNeuroniosSaida Numero de neuronios na camada de saida

\* @param propagação Função de propagação que será usado na rede

\*/

public RedeNeural(int numNeuroniosEntrada, int numNeuroniosOcultos, int numCamadasOcultas, int numNeuroniosSaida, Função propagação){

entrada = new CamadaEntrada(numNeuroniosEntrada);

oculta = new CamadaProcessamento[numCamadasOcultas];

oculta[0] = new CamadaProcessamento(numNeuroniosOcultos, propagação,numNeuroniosEntrada);

for(int i=1;i<oculta.length;i++){

oculta[i] = new CamadaProcessamento(numNeuroniosOcultos,propagação,numNeuroniosOcultos);

}

saida = new CamadaProcessamento(numNeuroniosSaida,propagação,numNeuroniosOcultos);

funçãoPropagação = propagação;

}

/\*\*

\* Realiza a alimentação na rede com a entrada especificada no parametro

\* @param inputs entrada da rede

\* @return retorna a saida da rede

\*/

public double[] feedFoward(double[] inputs){

double[] sinais = inputs;

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

sinais = oculta[i].feedFoward(sinais);

}

sinais = saida.feedFoward(sinais);

return sinais;

}

/\*\*

\* Realiza uma iteração na rede.

\* A iteração é composta de um feedfoward e um backpropagation para uma entrada.

\* @param inputs entrada para a iteração

\* @param outputEsperado resultado esperado

\* @return retorna o erro da rede nesta iteração.

\*/

public double iteration(double[] inputs, Double[] outputEsperado){

double[] output = feedFoward(inputs);

backPropagation(output,outputEsperado);

return erroRede();

}

/\*\*

\* Realiza o processo de treinamento da rede com as instancias do parametro

\* @param instancias instancias utilizadas para o treinamento

\*/

public void treinamento(Instancias instancias){

boolean houveErro;

double maiorErro;

instancias.definirSaidasClasses(funçãoPropagação);

for(int i=0;i<numIteraçõesLimite;i++){

houveErro = false;

double erroAtual;

for(int j=0;j<instancias.size();j++){

erroAtual=iteration(instancias.getAtributos(j),instancias.getSaida(j));

if(erroAtual > limiar)houveErro = true;

}

if(!houveErro)return;

}

}

/\*\*

\* Testa a rede com o conjunto do parametro e retorna a matriz de confusão

\* @param instancias conjunto de testes

\* @return matriz de confusão

\*/

public int[][] testarRede(Instancias instancias){

int numClasses = instancias.getNumClasses();

instancias.definirSaidasClasses(funçãoPropagação);

int[][] matrizConfusão = new int[numClasses][numClasses];

for(int i=0;i<numClasses;i++){

for(int j=0;j<numClasses;j++){

matrizConfusão[i][j] = 0;

}

}

for(int i=0;i<instancias.size();i++){

double[] saida = feedFoward(instancias.getAtributos(i));

int classeCalculada = indexMaiorSinalSaida(saida);

int classeDesejada = instancias.getIndexClasse(i);

matrizConfusão[classeCalculada][classeDesejada]++;

}

return matrizConfusão;

}

/\*\*

\* Processo de backpropagation da rede

\* @param outputs Saida calculada no final do processo feedFoward

\* @param outputEsperado Saida esperada pela rede

\*/

public void backPropagation(double[] outputs, Double[] outputEsperado){

saida.calculaErros(outputEsperado);

oculta[oculta.length-1].calculaErros(saida);

for(int i=oculta.length-2;i>=0;i--){

oculta[i].calculaErros(oculta[i+1]);

}

saida.ajustarPesos(taxaAprendizado);

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

oculta[i].ajustarPesos(taxaAprendizado);

}

}

/\*\*

\* @return Retorna o erro da rede para a iteração atual

\*/

private double erroRede() {

return saida.erroRede();

}

/\*\*

\* Verifica em qual indice se encontra a maior valor na saida da rede

\* @param saida

\* @return

\*/

private int indexMaiorSinalSaida(double[] saida) {

int pos=0;

double maior = Integer.MIN\_VALUE;

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

if(maior < saida[i]){

maior = saida[i];

pos = i;

}

}

return pos;

}

/\*\*

\* Retorna os pesos de um neuronio

\* @param camada indice da camada

\* @param neuronio indice do neuronio

\* @return

\*/

public double[] getPesos(int camada, int neuronio) {

if(oculta.length == camada) return saida.getPesos(neuronio);

return oculta[camada].getPesos(neuronio);

}

public double getTaxaAprendizado() {

return taxaAprendizado;

}

public void setTaxaAprendizado(double taxaAprendizado) {

this.taxaAprendizado = taxaAprendizado;

}

public int getNumIteraçõesLimite() {

return numIteraçõesLimite;

}

public void setNumIteraçõesLimite(int numIteraçõesLimite) {

this.numIteraçõesLimite = numIteraçõesLimite;

}

public double getLimiar() {

return limiar;

}

public void setLimiar(double limiar) {

this.limiar = limiar;

}

}