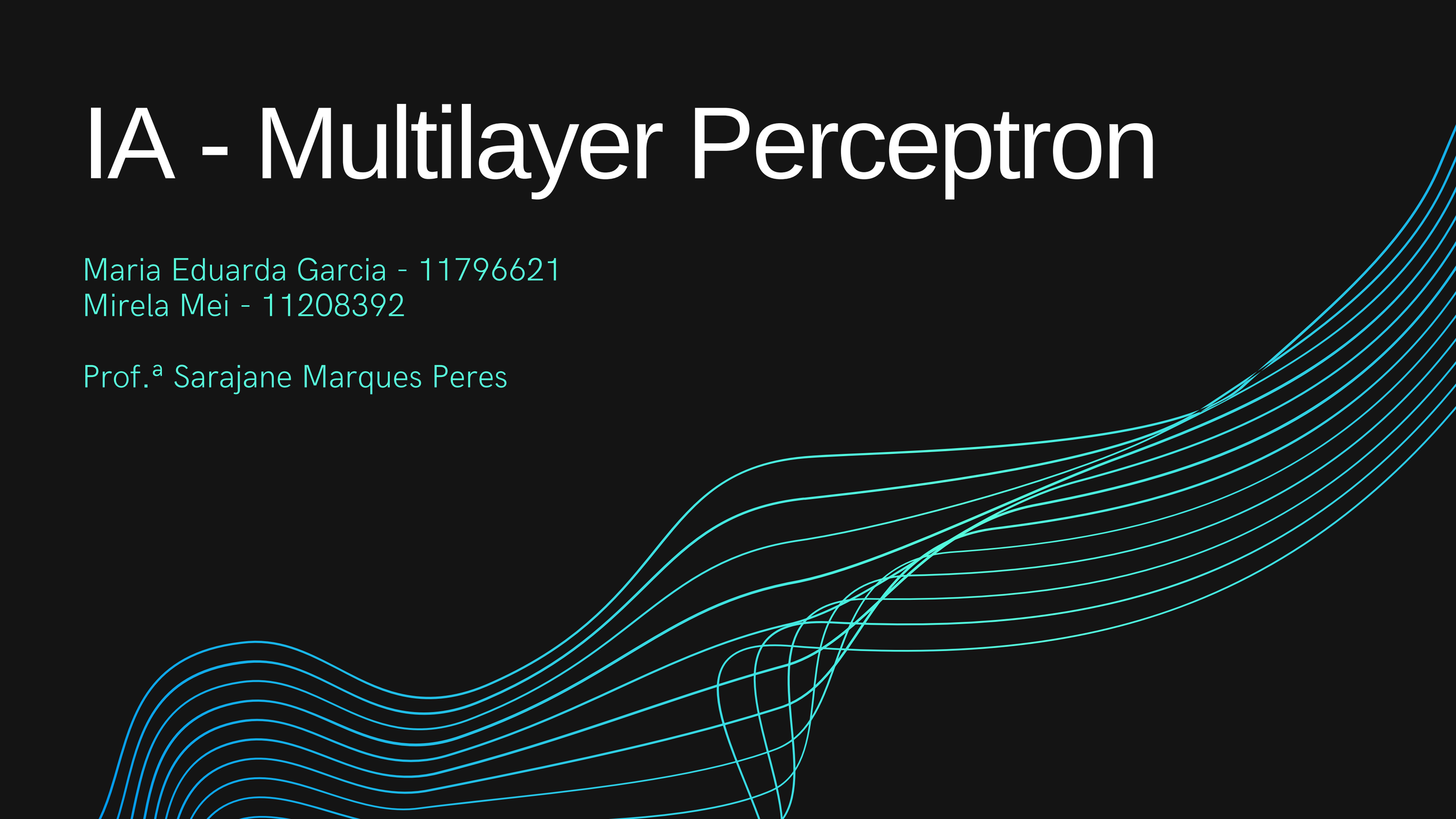
IA - Multilayer Perceptron

Maria Eduarda Garcia - 11796621
Mirela Mei - 11208392

Prof.ª Sarajane Marques Peres



MLP

DESCRIÇÃO

- Java
- Estrutura dividida em 3 classes principais:
  - NeuralNetwork
  - Layer
  - Neuron

NeuralNetwork

```java
public class NeuralNetwork{
    private Layer[] layers;
    private int[] layerInfo;
    private double[] inputs;
    private double[] finalOutputs;
    private double[] expectedOutputs;
    double MSE;

    public NeuralNetwork(int[] layerInfo){
        this.layerInfo = layerInfo;
        layers = new Layer[layerInfo.length];
        /* inicializa a rede neural criando cada camada com:
         * o seu respectivo indice,
         * o número de neurônios especificado no vetor layerInfo,
         * o numero de inputs por neuronio
         * (número de neurônios da camada anterior,
         * ou apenas 1 se for a camada de entrada),
         * a camada anterior (null se for a camada de entrada)
         * e a camada seguinte (no caso sempre null)
         */
        for (int i = 0; i < layerInfo.length; i++){
            int numNeurons = layerInfo[i];
            int numInputsPerNeuron = (i == 0) ? 1 : layerInfo[i - 1];
            Layer previousLayer = (i == 0) ? null : layers[i - 1];
            layers[i] = new Layer(i, numNeurons, numInputsPerNeuron, previousLayer);
        }
    }
```

Layer

```java
public class Layer{
    private int layerIndex;
    private Layer previousLayer;
    private Layer nextLayer;
    private Neuron[] neurons;
    private double[] outputs;

    public Layer(int layerIndex, int numNeurons, int numInputsPerNeuron, Layer previousLayer){
        this.layerIndex = layerIndex;
        this.neurons = new Neuron[numNeurons];
        this.previousLayer = previousLayer;
        /* inicializa a camada criando cada neurônio com:
         * o indice da camada que ele está
         * o indice do neuronio
         * e o número de inputs que ele vai ter
         * setta a nextLayer da camada anterior como 'this'
         */
        for (int i = 0; i < numNeurons; i++){
            neurons[i] = new Neuron(layerIndex, i, numInputsPerNeuron);
        }
        if (previousLayer != null){
            previousLayer.setNextLayer(this);
        }
    }
```

Neuron

```java
public class Neuron{
    private int neuronIndex;
    private int layerIndex;
    private double[] inWeights;
    private double bias;
    private double[] inputs;
    private double sum;
    private double output;
    private double errorInfo;
    private double[] delta;
    private double biasDelta;

    public Neuron(int layerIndex, int neuronIndex, int numInputs){
        this.layerIndex = layerIndex;
        this.neuronIndex = neuronIndex;
        this.inWeights = new double[numInputs];
        this.delta = new double[numInputs];
        /* caso não seja a camada de entrada,
        * inicializa o neurônio
        * percorre o vetor de pesos de entrada (+ bias)
        * e inicializa cada um com um valor aleatório
        * entre -1.0 e 1.0
        */
        if(layerIndex != 0){
            this.bias = Math.random() - 1.0;
            for (int i = 0; i < numInputs; i++){
                inWeights[i] = Math.random() - 1.0;
            }
        }
    }
```

Camadas

ENTRADA

No código, é tratada apenas como uma “porta”, ou seja, não faz cálculos e nem possui pesos, apenas repassa os valores que são enviados ao rodar o feedforward

- 120 neurônios - vetor de pixels

Camadas

OCULTA

É inicializada com pesos e bias aleatórios entre -1 e +1

Backpropagation:

- *auxErrorInfo* é a somatória *(errorInfo * respectivoPeso)* de todos os neuronios da camada seguinte
- *errorInfo* é o *auxErrorInfo* multiplicado pela *derivada da sigmoid*
- *delta* é o *errorInfo* multiplicado pelo *learning rate* e pelo *input* ligado ao peso em questão (output do neurônio anterior)

- 1 camada oculta com 60 neurônios
- 1 bias pra cada neurônio

CONJUNTO DE PESOS SINÁPTICOS

Matriz de dimensão 60 x 120

$$W_1 = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1,120} \\ w_{21} & w_{22} & \cdots & w_{2,120} \\ \vdots & \vdots & \ddots & \vdots \\ w_{60,1} & w_{60,2} & \cdots & w_{60,120} \end{bmatrix}$$

Camadas

SAÍDA

É inicializada com pesos e bias aleatórios entre -1 e +1

Backpropagation:

- *errorInfo* é o *erro* do output multiplicado pela *derivada da sigmoid*
- *delta* é o *errorInfo* multiplicado pelo *learning rate* e pelo *input* ligado ao peso em questão (output do neurônio anterior)

- 26 neurônios - 1 pra cada letra
- 1 bias pra cada neurônio

CONJUNTO DE PESOS SINÁPTICOS

Matriz de dimensão 26 x 60

$$W_2 = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1,60} \\ w_{21} & w_{22} & \cdots & w_{2,60} \\ \vdots & \vdots & \ddots & \vdots \\ w_{26,1} & w_{26,2} & \cdots & w_{26,60} \end{bmatrix}$$

Função de ativação

SIGMÓIDE

```
double sigmoidDerivative(){
    /* derivada da Sigmoid */
    return sigmoid(sum) * (1 - sigmoid(sum));
}

double sigmoid(double x){
    /* função de ativação Sigmoid */
    return 1 / (1 + Math.exp(-x));
}
```

- Usada em todos os neurônios do treinamento

Taxa de Aprendizado

Utilizada no cálculo do Delta (método backpropagation) para atualizar os pesos e bias

- o peso novo é o peso antigo mais o delta relacionado ao peso
- o bias novo é o bias antigo mais o biasDelta

- learningRate = 0.5

Camada escondida

$$\delta\_in_j = \sum_{k=1}^{m} \delta_k w_{jk} \text{ e } \delta_j = \delta\_in_j f'(z\_in_j)$$

$$\Delta v_{ij} = \alpha \delta_j x_i \text{ e } \Delta v_{0j} = \alpha \delta_j$$

Camada de saída

$$\delta_k = (t_k - y_k) f'(y\_in_k)$$

$$\Delta w_{jk} = \alpha \delta_k z_j \text{ e } \Delta w_{0k} = \alpha \delta_k$$

Alteração de pesos e bias

$$w_{jk}(new) = w_{jk}(old) + \Delta w_{jk}$$

$$v_{ij}(new) = v_{ij}(old) + \Delta v_{ij}$$

Treinamento - exemplo

DEFINIÇÃO DOS VALORES

```
int[] layerInfo = {6, 5, 2};
NeuralNetwork neuralNetwork = new NeuralNetwork(layerInfo);

double[] inputs = {0, 1, 0, 0, 0, 1};
double[] expectedOutputs = {1.0, 0.0};

double targetMSE = 0;
int maxIterations = 200000;
```

Treinamento - exemplo

LOOP

```
while (currentMSE > targetMSE && iteration < maxIterations) {
    neuralNetwork.runFeedForward(inputs);
    neuralNetwork.calculateMSE(expectedOutputs);
    currentMSE = neuralNetwork.getMSE();
    System.out.println("Iteration " + iteration + ", MSE: " + currentMSE);

    neuralNetwork.runBackpropagation(expectedOutputs, learningRate:0.01);

    iteration++;
}

System.out.println("Final MSE: " + currentMSE);
neuralNetwork.printOutputs();
```

Treinamento - exemplo

SAÍDA

```
Iteration 199974, MSE: 1.1538742163151338E-4
Iteration 199975, MSE: 1.1538672420063182E-4
Iteration 199976, MSE: 1.1538602677754438E-4
Iteration 199977, MSE: 1.1538532936225343E-4
Iteration 199978, MSE: 1.153846319547512E-4
Iteration 199979, MSE: 1.1538393455504676E-4
Iteration 199980, MSE: 1.1538323716313214E-4
Iteration 199981, MSE: 1.1538253977901231E-4
Iteration 199982, MSE: 1.1538184240268359E-4
Iteration 199983, MSE: 1.1538114503414959E-4
Iteration 199984, MSE: 1.1538044767340765E-4
Iteration 199985, MSE: 1.1537975032045763E-4
Iteration 199986, MSE: 1.1537905297530094E-4
Iteration 199987, MSE: 1.1537835563793486E-4
Iteration 199988, MSE: 1.1537765830836308E-4
Iteration 199989, MSE: 1.1537696098658031E-4
Iteration 199990, MSE: 1.1537626367259186E-4
Iteration 199991, MSE: 1.1537556636639475E-4
Iteration 199992, MSE: 1.1537486906798902E-4
Iteration 199993, MSE: 1.1537417177737361E-4
Iteration 199994, MSE: 1.1537347449455191E-4
Iteration 199995, MSE: 1.1537277721952014E-4
Iteration 199996, MSE: 1.1537207995227944E-4
Iteration 199997, MSE: 1.1537138269282973E-4
Iteration 199998, MSE: 1.1537068544117362E-4
Iteration 199999, MSE: 1.1536998819730438E-4
Final MSE: 1.1536998819730438E-4
---- OUTPUTS ----
0.9883143651969997
0.00970494284607159
```

FeedForward

CLASSE NEURAL NETWORK

```java
private double[] feedForward(double[] inputs){
    this.inputs = inputs;
    double[] outputs = inputs;
    /* calcula os valores de saida da rede neural
    * percorre as camadas e calcula a saída de cada uma
    * -> a saída de uma camada é a entrada da próxima
    */
    for (Layer layer : layers){
        outputs = layer.calculateOutputs(outputs);
    }
    this.finalOutputs = outputs;
    return finalOutputs;
}
```

FeedForward

CLASSE LAYER

```java
double[] calculateOutputs(double[] inputs){
    this.outputs = new double[neurons.length];
    /* calcula a saída de cada neurônio da camada
     * se for a camada de entrada, o input de cada neurônio é apenas 1 valor enviado na chamada do método
     * se não, o input é o vetor de saída da camada anterior
     */
    if (layerIndex == 0){
        for (int i = 0; i < neurons.length; i++){
            outputs[i] = neurons[i].calculateOutput(new double[]{inputs[i]});
        }
    }else{
        for (int i = 0; i < neurons.length; i++){
            outputs[i] = neurons[i].calculateOutput(inputs);
        }
    }
    return outputs;
}
```

FeedForward

CLASSE NEURON

```
double calculateOutput(double[] inputs){
    this.inputs = inputs;
    this.sum =  this.bias; // somando o bias antes da multiplicação dos pesos
    /* se for a camada de entrada,
     * a saída do neurônio é o input, sem cálculos
     * se não, calcula a saída do neurônio
     * percorrendo o vetor de pesos de entrada e multiplica cada peso pelo respectivo input
     * soma o resultado
     * aplica a função de ativação (sigmoid) ao resultado da soma
     */
    if(layerIndex == 0){
        this.output = inputs[0];
    }else{
        for (int i = 0; i < inputs.length; i++){
            this.sum += inputs[i] * inWeights[i];
        }
        this.output = sigmoid(sum);
    }
    return output;
}
```

Atualização dos pesos

CLASSE NEURAL NETWORK

```java
private void backpropagation(double[] expectedOutputs, double learningRate){
    /* calcula o erro na camada de saída,
     * propaga o erro de volta para as camadas ocultas
     * e atualiza os pesos e bias em todas as camadas
     */
    for (int i = layers.length - 1; i > 0; i--){
        layers[i].backpropagate(expectedOutputs, learningRate);
    }

    for (int i = layers.length - 1; i > 0; i--){
        layers[i].updateWeightsAndBiases();
    }
}
```

Backpropagate

CLASSE LAYER

```
void backpropagate(double[] expectedOutputs, double learningRate){
    /* se for a camada de saída, chama o backpropagateOutputLayer
     * se for uma camada oculta, chama o backpropagateHiddenLayer
     * a camada de entrada não roda o backpropagate
     */
    if (nextLayer == null){
        backpropagateOutputLayer(expectedOutputs, learningRate);
    } else if (previousLayer != null){
        backpropagateHiddenLayer(learningRate);
    }
}
```

Backpropagate

CAMADA DE SAÍDA - CLASSE LAYER

$$\delta_k = (t_k - y_k) f'(y\_in_k)$$

$$\Delta w_{jk} = \alpha \delta_k z_j \text{ e } \Delta w_{0k} = \alpha \delta_k$$

```
void backpropagateOutputLayer(double[] expectedOutputs, double learningRate){
    /* se for a camada de saida,
     * a Informação de Erro (errorInfo) será o erro do output multiplicado pela derivada da sigmoid
     * e o delta será o errorInfo multiplicado pelo learning rate
     * e pelo input ligado ao peso em questão (output do neurônio anterior)
     */
    for (int i = 0; i < neurons.length; i++){
        double[] inWeights = neurons[i].getInWeights();
        double[] inputs = neurons[i].getInputs();
        double[] errorInfo = new double[neurons.length];
        double[] delta = new double[inWeights.length];
        double biasDelta;

        errorInfo[i] = neurons[i].outputGradient(expectedOutputs[i]) * neurons[i].sigmoidDerivative();
        for (int j = 0; j < inWeights.length; j++){
            delta[j] = learningRate * errorInfo[i] * inputs[j];
        }
        biasDelta = learningRate * errorInfo[i];
        neurons[i].setErrorInfo(errorInfo[i]);
        neurons[i].setDelta(delta);
        neurons[i].setBiasDelta(biasDelta);
    }
}
```

Backpropagate

CAMADA OCULTA - CLASSE LAYER

$$\delta\_in_j = \sum_{k=1}^{m} \delta_k w_{jk} \text{ e } \delta_j = \delta\_in_j f'(z\_in_j)$$

$$\Delta v_{ij} = \alpha \delta_j x_i \text{ e } \Delta v_{0j} = \alpha \delta_j$$

```java
void backpropagateHiddenLayer(double learningRate){
    /* se for uma camada oculta,
     * o auxErrorInfo será a somatória (errorInfo * respectivoPeso) de todos os neuronios da camada seguinte
     * o errorInfo será o auxErrorInfo multiplicado pela derivada da sigmoid
     * e o delta será o errorInfo multiplicado pelo learning rate
     * e pelo input ligado ao peso em questão (output do neurônio anterior)
     */
    Neuron[] nextLayerNeurons = nextLayer.getNeurons();
    for (int i = 0; i < neurons.length; i++){
        double auxErrorInfo = 0;
        double[] inWeights = neurons[i].getInWeights();
        double[] inputs = neurons[i].getInputs();
        double[] errorInfo = new double[neurons.length];
        double[] delta = new double[inWeights.length];
        double biasDelta;

        for (int j = 0; j < nextLayerNeurons.length; j++){
            auxErrorInfo += nextLayerNeurons[j].getErrorInfo() * nextLayerNeurons[j].getInWeights()[i];
        }
        errorInfo[i] = auxErrorInfo * neurons[i].sigmoidDerivative();
        for (int k = 0; k < inWeights.length; k++){
            delta[k] = learningRate * errorInfo[i] * inputs[k];
        }
        biasDelta = learningRate * errorInfo[i];
        neurons[i].setErrorInfo(errorInfo[i]);
        neurons[i].setDelta(delta);
        neurons[i].setBiasDelta(biasDelta);
    }
}
```

updateWeightsAndBiases

CLASSE LAYER

```java
void updateWeightsAndBiases(){
    /* percorre cada neurônio da camada e atualiza os pesos e bias */
    for (Neuron neuron : neurons){
        neuron.updateWeightsAndBias();
    }
}
```

CLASSE NEURON

```java
void updateWeightsAndBias(){
    /* o peso novo será o peso antigo mais o delta relacionado ao peso
     * o bias novo será o bias antigo mais o biasDelta
     */
    for (int i = 0; i < inWeights.length; i++){
        double newWeight = inWeights[i] + delta[i];
        this.inWeights[i] = newWeight;
    }
    this.bias = bias + biasDelta;
}
```

Cálculo de Erro

CLASSE NEURON

```
double outputGradient(double expectedOutput){
    /* valor esperado menos a saída */
    return (expectedOutput - output);
}
```

CLASSE NEURAL NETWORK - MSE

```
void calculateMSE(double[] expectedOutputs){
    /* calcula o erro quadrático médio
     * percorre os valores esperados e os valores de saída
     * calcula o erro para cada saída e soma os quadrados
     * divide a soma pelo número de saídas
     */
    if (expectedOutputs.length != finalOutputs.length){
        throw new IllegalArgumentException("Number of expected outputs must be equal to number of neurons in output-layer");
    }
    double sumSquaredErrors = 0.0;
    for (int i = 0; i < finalOutputs.length; i++){
        double error = expectedOutputs[i] - finalOutputs[i];
        sumSquaredErrors += Math.pow(error, 2);
    }
    this.MSE = sumSquaredErrors / finalOutputs.length;
}
```

Reconhecimento de caractere

CLASSE LETTERPROCESSOR

```
private void handleTestFold(double[] outputs, int linha, int aux, char[] expectedResponses, ch
    /* armazena as respostas esperadas e as respostas finais
     * para gerar a matriz de confusão e calcular a acurácia
     */
    double[] response = new double[26];
    int index = findMaxIndex(outputs);

    for (int i = 0; i<outputs.length; i++){
        response[i] = (i == index) ? 1 : 0;
    }

    char actualResponse = findOutResponseLetter(response);
    expectedResponses[aux] = AlphabetVectors.getExpectedResponse(linha);
    finalResponses[aux] = actualResponse;
}
```

Reconhecimento de caractere

CLASSE LETTERPROCESSOR

```
try (PrintWriter pw = new PrintWriter(new BufferedWriter(new FileWriter(mseFilePath)))){
    while ((line = br.readLine()) != null){
        linha = linha % 26;
        linha = linha == 0 ? 26 : linha;
        double[] expectedOutputs = AlphabetVectors.getLetter(linha);
        double[] inputs = parseInputLine(line);
        double[] outputs = neuralNetwork.runFeedForward(inputs);

        if (!isTestFold){
            neuralNetwork.runBackpropagation(expectedOutputs, learningRate);
        }else{
            if(isEarlyStopping){
                validationLabels[index] = expectedOutputs;
                validationPredictions[index] = outputs;
                index++;
```

Reconhecimento de caractere

CLASSE ALPHABETVECTORS

```java
public class AlphabetVectors{
    private static final int ALPHABET_SIZE = 26;
    private static final Map<Character, double[]> charToVectorMap = new HashMap<>();
    private static final Map<Integer, Character> vectorToCharMap = new HashMap<>();

    static{
        /* cria um mapa de caracteres para vetores e um mapa de vetores para caracteres
         * onde cada caractere é mapeado para um vetor de 26 posições (um para cada letra do al
         * e cada vetor é mapeado para um caractere
         */
        for (char c = 'A'; c <= 'Z'; c++){
            double[] vector = new double[ALPHABET_SIZE];
            vector[c - 'A'] = 1.0;
            charToVectorMap.put(c, vector);
            vectorToCharMap.put(Arrays.hashCode(vector), c);
        }
    }
```

Reconhecimento de caractere

CLASSE ALPHABETVECTORS

```java
public static double[] getLetter(int line){
    /* retorna o vetor correspondente a um número específico (1 para A, 2 para B, etc)
     * se a linha fornecida for menor que 1 ou maior que o tamanho do alfabeto, ele retorna nu
     */
    if (line < 1 || line > ALPHABET_SIZE){
        return null;
    }
    return charToVectorMap.get((char) ('A' + line - 1));
}

public static char getExpectedResponse(int line){
    /* retorna o caractere correspondente a um número específico (1 para A, 2 para B, etc)
     * se a linha fornecida for menor que 1 ou maior que o tamanho do alfabeto, ele retorna '-
     */
    if (line < 1 || line > ALPHABET_SIZE){
        return '-';
    }
    return (char) ('A' + line - 1);
}
```

```java
public static char decodeResponse(double[] response){
    /* decodifica um vetor para retornar o caractere correspondente
     * ele utiliza o hash do vetor para encontrar a correspondência no mapa
     * se o vetor não tiver uma correspondência, ele retorna '-'
     */
    return vectorToCharMap.getOrDefault(Arrays.hashCode(response), '-');
}

public static int returnLetterNumber(char letter){
    /* retorna o número correspondente a uma letra específica (0 para A, 1 para B, etc)
     * se a letra fornecida estiver fora do intervalo de 'A' a 'Z', ele retorna -1
     */
    if (letter < 'A' || letter > 'Z'){
        return -1;
    }
    return letter - 'A';
}
```

Matriz de Confusão

CLASSE LETTER PROCESSOR

```
private void evaluateResults(int numTestEntrance, char[] expectedResponses, char[] finalResponses, Str
    /* gera a matriz de confusão e calcula a acurácia */
    Evaluator.generateConfusionMatrix(finalResponses, expectedResponses, numTestEntrance, fileNameSuff
    double accuracy = Evaluator.calculateAccuracy(numTestEntrance);
    double error = Evaluator.calculateError(accuracy);

    System.out.println("Accuracy: " + accuracy + " // error: " + error);
}
```

CLASSE EVALUATOR

```
public class Evaluator{
    public static final int[][] confusionMatrix = new int[26][26];

    public static void generateConfusionMatrix(char[] finalResponses, char[] expectedResponses, int numTest
        /* gera a matriz de confusão */
        clearPreviousMatrix();
        updateConfusionMatrix(finalResponses, expectedResponses, numTestEntrance);
        printConfusionMatrix(fileNameSuffix);
    }

    public static void clearPreviousMatrix(){
        /* limpa a matriz de confusão */
        for (int i = 0; i<26; i++){
            for (int j = 0; j < 26; j++){
                confusionMatrix[i][j] = 0;
            }
        }
    }
```

Matriz de Confusão

CLASSE EVALUATOR

```java
private static void updateConfusionMatrix(char[] finalResponses, char[] expectedResponses
    /* recebe as respostas finais e as esperadas
     * e atualiza a matriz de confusão
     * onde a linha é a resposta real da rede e a coluna é a resposta esperada
     * a matriz é preenchida com os valores respondidos pela rede, e a diagonal
     * representa a quantidade de vezes que a resposta final foi a resposta esperada
     */
    for (int i = 0; i<numTestEntrance; i++){
        int finalResponse = AlphabetVectors.returnLetterNumber(finalResponses[i]);
        int expectedResponse = AlphabetVectors.returnLetterNumber(expectedResponses[i]);
        if (finalResponse != -1 && expectedResponse != -1){
            confusionMatrix[expectedResponse][finalResponse]++;
        }
    }
}
```

Matriz de Confusão

CLASSE EVALUATOR

```java
private static void printConfusionMatrix(String fileNameSuffix){
    /* imprime a matriz de confusão no arquivo confusion_matrix_<suffix>.csv */
    System.out.println("entrou");
    int totalEntries = 0;
    try (FileWriter writer = new FileWriter("plot/matrix/confusion_matrix_" + fileNameSuffix + ".csv"))
        for (int i = 0; i<26; i++){
            for (int j = 0; j < 26; j++){
                writer.append(Integer.toString(confusionMatrix[i][j]));
                if (j<25){
                    writer.append(",");
                }
                totalEntries += confusionMatrix[i][j];
            }
            writer.append("\n");
        }
    } catch (IOException e){
        e.printStackTrace();
    }
    System.out.println("Total de entradas: " + totalEntries);
}
```

Obrigada!