**LISTA – RNA SIMPLES**

**Atividade 1**

**a)** A resposta a essa pergunta depende da separabilidade linear dos dados. Visualmente, os dados parecem ser linearmente separáveis, ou seja, é possível traçar uma linha reta que separa as maçãs das laranjas. Se os dados forem linearmente separáveis, um perceptron de camada simples é capaz de aprender a classificação. No entanto, a separabilidade linear não é garantida apenas pela visualização, e o treinamento do perceptron é necessário para confirmar essa hipótese.

**b)**

A tabela apresenta um processo de treinamento de um perceptron. Cada linha representa uma iteração de treinamento, e as colunas significam:

**x1, x2: Valores de entrada.**

**w1, w2: Pesos sinápticos.**

**ol(o): Saída calculada pelo perceptron.**

**y: Saída desejada.**

**taxa: Taxa de aprendizado.**

**erro: Erro entre a saída desejada e a saída calculada.**

**w1novo, w2novo: Novos valores dos pesos após a atualização.**

Analisando as Iterações

A tabela mostra um processo de treinamento incompleto e com alguns erros nos cálculos. Vamos analisar as primeiras iterações para identificar os problemas:

Iteração 1:

Cálculo da saída: ol(o) = x1w1 + x2w2 = 00,7 + 300,8 = 24. Como a saída é positiva, a função de ativação (não explicitada) provavelmente retorna 1.

Cálculo do erro: O erro não está calculado corretamente. Se a saída desejada (y) for 0 (assumindo que a saída desejada é binária), o erro seria -1.

Atualização dos pesos: Os novos pesos não estão calculados corretamente. A fórmula de atualização dos pesos é:

w1novo = w1 + taxa \* erro \* x1

w2novo = w2 + taxa \* erro \* x2

Iterações subsequentes:

Os mesmos problemas se aplicam às outras iterações. Os cálculos do erro e da atualização dos pesos estão incorretos.

Corrigindo os Cálculos

Para corrigir os cálculos, precisamos:

Definir a função de ativação: A função de ativação não está explicitada na tabela. Vamos assumir que a função de ativação é a função degrau:

se x1w1 + x2w2 >= 0, então ol(o) = 1

caso contrário, ol(o) = 0

Calcular o erro corretamente: O erro é a diferença entre a saída desejada e a saída calculada: erro = y - ol(o).

Atualizar os pesos corretamente: Utilizar a fórmula de atualização dos pesos mencionada anteriormente.

Exemplo Corrigido (primeira iteração):

x1 x2 w1 w2 ol(o) y taxa erro w1novo w2novo

0 30 0,7 0,8 1 0 0,1 -1 -0,03 0,77

Exportar para as Planilhas

Completando o Treinamento

Para completar o treinamento, é necessário continuar as iterações, calculando o erro e atualizando os pesos em cada iteração até que o erro seja menor que um valor predefinido ou até que um número máximo de iterações seja atingido.

Observações:

Função de ativação: A escolha da função de ativação pode influenciar o desempenho do perceptron. Outras funções de ativação comuns são a sigmoide e a tangente hiperbólica.

Taxa de aprendizado: A taxa de aprendizado controla a velocidade de convergência do algoritmo. Um valor muito alto pode levar à divergência, enquanto um valor muito baixo pode levar a um treinamento lento.

Inicialização dos pesos: A inicialização dos pesos pode afetar o desempenho do perceptron. É comum inicializar os pesos com valores aleatórios pequenos.

Normalização dos dados: Normalizar os dados pode melhorar o desempenho do perceptron.

Utilizando uma Ferramenta de Cálculo

Para realizar os cálculos de forma mais eficiente, você pode utilizar uma planilha eletrônica (como o Excel) ou uma linguagem de programação como Python ou MATLAB. Essas ferramentas permitem automatizar os cálculos e visualizar os resultados de forma mais clara.

**Assumindo a Função Degrau**

Vamos assumir que a função de ativação é a função degrau:

* Se io >= 0, então o(o) = 1
* Se io < 0, então o(o) = 0

**Calculando as Primeiras Linhas**

**Linha 1:**

| x1 | x2 | w1 | w2 | io | o(o) | y | n | erro | w1novo | w2novo | |---|---|---|---|---|---|---|---|---|---| | 0.3 | 0.7 | 0.7 | -0.8 | -0.11 | 0 | 0 | 0.5 | 0 | 0.7 | -0.8 |

**Cálculos:**

* io = 0.3 \* 0.7 + 0.7 \* -0.8 = -0.11
* Como io < 0, o(o) = 0
* Erro = y - o(o) = 0 - 0 = 0
* w1novo = w1 + n \* erro \* x1 = 0.7 + 0.5 \* 0 \* 0.3 = 0.7
* w2novo = w2 + n \* erro \* x2 = -0.8 + 0.5 \* 0 \* 0.7 = -0.8

**Linha 2:**

| x1 | x2 | w1 | w2 | io | o(o) | y | n | erro | w1novo | w2novo | |---|---|---|---|---|---|---|---|---|---| | -0.6 | 0.3 | 0.7 | -0.8 | -0.15 | 0 | 0 | 0.5 | 0 | 0.7 | -0.8 |

**Cálculos:**

* io = -0.6 \* 0.7 + 0.3 \* -0.8 = -0.15
* Como io < 0, o(o) = 0
* Erro = y - o(o) = 0 - 0 = 0
* w1novo = w1 + n \* erro \* x1 = 0.7 + 0.5 \* 0 \* -0.6 = 0.7
* w2novo = w2 + n \* erro \* x2 = -0.8 + 0.5 \* 0 \* 0.3 = -0.8

**c) código pedido abaixo:**

**using System;**

**namespace Perceptron**

**{**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**double[,] inputs = { { 113, 6.8 }, { 122, 4.7 }, { 107, 5.2 }, { 98, 3.6 }, { 115, 2.9 }, { 120, 4.2 } };**

**int[] outputs = { 1, -1, 1, 1, -1, -1 }; // 1 para maçã, -1 para laranja**

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

**double[] weights = new double[inputs.GetLength(1) + 1]; // +1 para o bias**

**for (int i = 0; i < weights.Length; i++)**

**{**

**weights[i] = random.NextDouble() \* 2 - 1; // Valores aleatórios entre -1 e 1**

**}**

**double learningRate = 0.1;**

**for (int epoch = 0; epoch < 100; epoch++)**

**{**

**for (int i = 0; i < inputs.GetLength(0); i++)**

**{**

**double sum = 0;**

**for (int j = 0; j < inputs.GetLength(1); j++)**

**{**

**sum += inputs[i, j] \* weights[j];**

**}**

**sum += weights[weights.Length - 1]; // Bias**

**int output = sum >= 0 ? 1 : -1;**

**double error = outputs[i] - output;**

**for (int j = 0; j < weights.Length; j++)**

**{**

**weights[j] += learningRate \* error \* (j == weights.Length - 1 ? 1 : inputs[i, j]);**

**}**

**}**

**}**

**double[,] newInputs = { { 110, 5.5 }, { 125, 3.0 }, { 105, 6.0 }, { 95, 4.0 }, { 118, 4.5 } };**

**foreach (double[] input in newInputs)**

**{**

**double sum = 0;**

**for (int i = 0; i < input.Length; i++)**

**{**

**sum += input[i] \* weights[i];**

**}**

**sum += weights[weights.Length - 1];**

**int output = sum >= 0 ? 1 : -1;**

**Console.WriteLine($"Classificação: {(output == 1 ? "Maçã" : "Laranja")}");**

**}**

**}**

**}**

**}**

**d) Atividade 4**

Em problemas mais complexos com maior variabilidade de características, são necessários mais dados de treinamento para que a rede neural possa generalizar bem para novos dados. Os motivos são:

Maior espaço de busca: Um espaço de características maior implica em um espaço de parâmetros maior para a rede neural explorar, o que exige mais dados para encontrar os melhores valores para os pesos.

Maior complexidade: Problemas mais complexos podem exigir redes neurais mais complexas com mais camadas e neurônios, o que aumenta a quantidade de parâmetros a serem ajustados.

Overfitting: Com poucos dados, a rede neural pode se ajustar demais aos dados de treinamento (overfitting), aprendendo o ruído presente nos dados em vez das características relevantes. Ao aumentar a quantidade de dados, é possível reduzir o risco de overfitting.

um conjunto de treinamento maior permite que a rede neural aprenda um modelo mais robusto e generalizável, capaz de lidar com a variabilidade presente em dados reais.

Generalização:

Capturar a complexidade: Redes neurais aprendem padrões complexos nos dados. Um conjunto de treinamento maior permite que a RNA capture uma variedade maior de padrões, tornando-a mais capaz de generalizar para dados nunca antes vistos.

Evitar overfitting: Com poucos dados, a RNA pode "decorar" os exemplos de treinamento ao invés de aprender as relações subjacentes. Um conjunto de treinamento maior ajuda a evitar esse problema, pois a rede é forçada a encontrar padrões mais gerais.

2. Robustez:

Lidar com ruído: Dados reais geralmente contêm ruído e outliers. Um conjunto de treinamento maior permite que a RNA seja mais robusta a essas variações, tornando suas predições mais confiáveis.

Aumentar a confiança: Quanto mais dados a RNA "ver", mais confiante podemos estar em suas predições.

3. Otimização dos parâmetros:

Melhor ajuste: Um conjunto de treinamento maior permite que os algoritmos de otimização (como o gradiente descendente) encontrem valores mais precisos para os parâmetros da rede (pesos e vieses).

Evitar mínimos locais: Com mais dados, a probabilidade de a otimização ficar presa em mínimos locais é menor.

4. Avaliação mais precisa:

Validação cruzada: Um conjunto de treinamento maior permite realizar uma validação cruzada mais robusta, dividindo os dados em conjuntos de treinamento e validação de forma mais significativa. Isso ajuda a avaliar o desempenho da rede e escolher os melhores hiperparâmetros.

Em resumo:

Quanto mais dados de treinamento, mais a RNA pode aprender sobre o problema, tornando-a mais precisa, robusta e generalizável. No entanto, é importante balancear a quantidade de dados com a complexidade da rede neural. Uma rede muito complexa treinada com poucos dados pode levar ao overfitting, enquanto uma rede muito simples treinada com muitos dados pode não ser capaz de capturar todas as nuances do problema.