Rede Neural Perceptron para a classificação do salmão e robalo.

Daniel C Anselmo[1]

[1]Instituto Federal de Educação, Ciência e Tecnologia do Ceará.

[2]Programa de Pós-Graduação em Ciência da Computação do IFCE.

Abstract. *This report describes the implementation of a perceptron neural network for the classification of two types of fish, salmon and sea bass.*

Resumo. *Esse relatório descreve a implementação de uma rede neural perceptron para a classificação de dois tipos de peixes, salmão e robalo.*

1. Introdução

O Perceptron é um modelo de rede neural artificial criado em 1958 por Frank Rosenblatt, com o objetivo de simular o comportamento de um neurônio biológico. As diferenças entre um neurônio artificial e um neurônio biológico é que o primeiro possui um somador de pesos com valores de entradas e uma função de ativação, e processamento acontece por meio de operações matemáticas. O neurônio biológico por sua vez possui um corpo celular, dendritos, axônios e sinapses e processa tudo através da bioquímica(neurotransmissores).

Uma rede neural artificial funciona como um classificador linear, recebendo um conjunto de entradas, que são multiplicadas, cada entrada, por um peso. O somatório desses produtos de entradas pelos pesos é passada para uma função de ativação, que decide a saída do neurônio, geralmente sendo 0 ou 1, indicando assim a classe.

Com base no erro calculado, os pesos são ajustados durante o treinamento da rede, um processo conhecido como backpropagation. O erro é igual à subtração da saída prevista pela saída esperada, e nesse processo iterativo, com um determinado número de "épocas", ocorre a aprendizagem. Os pesos são importantes porque eles determinam como o modelo interpreta os dados de entrada. Então, o ajuste deles, durante o processo de aprendizagem, é importante pois ajuda o modelo a se adaptar melhor aos dados e a realizar mais classificações corretas. O processo de backpropagation faz com que o perceptron aprenda com os erros.

2. Dataset do salmão e robalo.

O conjunto de dados possui 131 entradas, com dois atributos: luminosidade e largura. O conjunto tem 74 classes 0 e 57 classes 1.

```
In [18]: (df['species']).value_counts()

Out[18]: species
         0    74
         1    57
         Name: count, dtype: int64
```

Para abrir o conjunto de dados foi utilizado a biblioteca pandas do python, com o método *read_csv()*. Os dados foram divididos em dados de entrada e dados de saída e foram colocados em formato de lista com o método *tolist()*.

```
import pandas as pd

peixes = pd.read_csv('robalo-salmao.csv')


entrada1 = peixes['lightness'].tolist()
entrada2 = peixes['width'].tolist()

entrada = [[x, y] for x, y in zip(entrada1, entrada2)]
saida = peixes['species'].tolist()

X = entrada
y = saida
```

3. Perceptron

Antes de iniciar a explicação do código, é preciso explicar os conceitos de bias(viés) e taxa de aprendizagem.

O bias é um valor extra somado ao somatório dos produtos das entradas pelos pesos, fazendo assim um deslocamento na função de ativação, ajudando a rede a se ajustar melhor aos dados. Já a taxa de aprendizagem é um dos hiperparâmetros mais importantes de uma rede neural, ele controla o ajuste dos pesos durante o treinamento. Caso essa taxa seja muito alta, a rede neural pode pular a melhor solução e nunca convergir. Se ela for muito baixa, o aprendizado será lento, correndo o risco de ficar preso em mínimos locais ou demorar muito para chegar em uma ótima solução.

A classe Perceptron recebe como parâmetros o input_size que se trata da quantidade de atributos do conjunto de dados, no caso o *lightness* e o *width* e a taxa_aprendizado. O construtor da classe recebe a taxa de aprendizado, inicializa os pesos e o bias aleatoriamente, com o método *random()*, com valores entre -1 e 1.

A seguir é apresentado o código da classe Perceptron.

```python
# Perceptron
class Perceptron:
    def __init__(self, input_size, taxa_aprendizado=0.1):
        self.taxa_aprendizado = taxa_aprendizado
        # Inicializando os pesos aleatoriamente
        self.pesos = [random.uniform(-1, 1) for _ in range(input_size)]
        # Inicializando o bias aleatoriamente
        self.bias = random.uniform(-1, 1)

    # Função para treinar o modelo
    def treinar(self, X, y, epocas):
        for _ in range(epocas):
            for i in range(len(X)):
                # Passo 1: cálculo da saída (com a fórmula de ativação)
                entrada = X[i]
                soma = sum(entrada[j] * self.pesos[j] for j in range(len(entrada))) + self.bias
                previsao = sigmoide(soma)

                # Passo 2: cálculo do erro
                erro = y[i] - previsao


                # Passo 3: ajuste dos pesos
                for j in range(len(self.pesos)):
                    self.pesos[j] += self.taxa_aprendizado * erro * derivada_sigmoide(previsao) * entrada[j]

                # Ajuste do bias
                self.bias += self.taxa_aprendizado * erro * derivada_sigmoide(previsao)

            print(f'Erro: {erro}')
```

A classe possui o método de treinar o modelo, que recebe como parâmetro os dados de entrada e de saída, e o número de épocas, ou seja, o número de iterações que irá acontecer para que ocorra a aprendizagem, à medida que os pesos são atualizados com o objetivo de diminuir o erro. O método realiza o somatório das entradas pelos pesos, e depois esse resultado é somado ao bias. Esse resultado é passado para a função de ativação sigmoide, que retornará o valor da classe prevista. A função de ativação sigmoide transforma a saída de um neurônio em um valor entre 0 e 1. Se a entrada for muito negativa, o resultado tende a 0, se for muito positiva, então o resultado tende a 1. A fórmula dela é definida dessa forma:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Foi utilizado o método *math()* para a implementação dela em python.

```python
import math

# Função de ativação (sigmoide)
def sigmoide(x):
    return 1 / (1 + math.exp(-x))
```

Após isso, é calculado o erro do modelo, comparando a saída prevista pela função sigmoide com classe real. O erro é o resultado da subtração da classe real pela saída prevista.

Com o erro calculado, agora é necessário ajustar os pesos e o bias. O peso novo será igual a: taxa de aprendizagem * erro * derivada da sigmoide(previsão) * entrada relativa àquele peso que está sendo ajustado. A taxa de aprendizagem indica o quanto o peso será ajustado. A derivada da sigmoide mostra o quanto a saída muda com uma pequena mudança na entrada. Ela direciona o gradiente, mostrando qual deverá ser a inclinação da curva. A fórmula dela é a seguinte:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

A implementação da função é a seguinte:

```python
# Derivada da sigmoide
def derivada_sigmoide(x):
    return x * (1 - x)
```

E o novo bias é igual a: taxa de aprendizagem * erro * derivada da sigmóide(previsão)

Esse fluxo é iterado em toda a lista de entradas, e feito por um determinado número de épocas.

4. Resultados.

Nessa seção é apresentada a acurácia da rede e a matriz de confusão, mostrando assim a precisão da rede para classificação.

4.1. Acurácia

A acurácia mede o quão bem o modelo está acertando em suas previsões. Sua fórmula é a seguinte:

$$\text{Acurácia} = \frac{TP + TN}{TP + TN + FP + FN}$$

Onde TP são os verdadeiros positivos, ou seja os resultados que a rede previu que eram verdadeiros e realmente eram verdadeiros. Exemplo: Robalo - 1, Salmão - 0. O modelo previu que era 1 e realmente era 1. TN são os resultados que o modelo previu 0, e realmente era 0. FP são os resultados que o modelo previu 1, mas era 0. E FN são os resultados que o modelo previu 0 e era 1.

A implementação em python ficou assim:

```python
def acuracia(y_real, y_predito):
    # Inicializando os valores de TP, FP, TN, FN
    TP = FP = TN = FN = 0

    # Percorrendo as listas de valores reais e previstos
    for real, predito in zip(y_real, y_predito):
        if real == 1 and predito == 1:
            TP += 1  # True Positive
        elif real == 0 and predito == 1:
            FP += 1  # False Positive
        elif real == 0 and predito == 0:
            TN += 1  # True Negative
        elif real == 1 and predito == 0:
            FN += 1  # False Negative

    # Calculando a acurácia
    acuracia = (TP + TN) / (TP + TN + FP + FN)
    return acuracia
```

A rede apresentou uma acurácia de 0.8550.

```
Previsões: [0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Acurácia: 0.8550
```

4.2 Matriz de Confusão.

A matriz de confusão tem a seguinte estrutura:

	Previsão Positiva	Previsão Negativa
Real Positivo	True Positive (TP)	False Negative (FN)
Real Negativo	False Positive (FP)	True Negative (TN)

Implementação da matriz de confusão:

```python
def matriz_de_confusao(y_real, y_predito):
    # Inicializando os valores de TP, FP, TN, FN
    TP = FP = TN = FN = 0

    # Percorrendo as listas de valores reais e previstos
    for real, predito in zip(y_real, y_predito):
        if real == 1 and predito == 1:
            TP += 1  # True Positive
        elif real == 0 and predito == 1:
            FP += 1  # False Positive
        elif real == 0 and predito == 0:
            TN += 1  # True Negative
        elif real == 1 and predito == 0:
            FN += 1  # False Negative

    # Retorna a matriz de confusão
    return [[TP, FP], [FN, TN]]
```

Resultado da matriz de confusão após a execução da rede:

```
Matriz de Confusão:
[56, 18]
[1, 56]

In [2]:
```