Trabaho de Redes Neurais

Luiz Felipe Pierre Pestana
João Pedro Paiva Cardoso
Marcos Paulo Sousa Santos

29 de maio 2023

Github https://github.com/Mr-marcs/NeuralNetwork/

1 Descrição

Este projeto teve como objetivo utilizar redes neurais para realizar análises de classificações. Para isso, foram empregados dois neurônios em uma arquitetura de rede neural. A função de ativação utilizada foi a sigmoide, e o algoritmo de treinamento utilizado foi o gradiente descendente.

As redes neurais são treinadas com um conjunto de dados contendo exemplos de diferentes classes ou categorias. O objetivo é ensinar a rede neural a identificar corretamente a classe de novos exemplos que não foram vistos durante o treinamento.

O gradiente descendente é um método de otimização que ajusta os pesos dos neurônios com base na taxa de erro calculada durante o treinamento. A função sigmoide é comumente usada como função de ativação em redes neurais, pois mapeia os valores de entrada para um intervalo entre 0 e 1, facilitando a interpretação probabilística dos resultados.

2 O que queremos classificar

Nessa seção iremos explicar a base de dados e todos os processos que utilizamos de pré-processamento

2.1 Sobre dataset

O conjunto de dados Iris, também conhecido como conjunto de dados Iris Flower, utiliza dados sobre características de três espécies diferentes de flores Iris (setosa, virginica e versicolor). Esses dados incluem medidas das pétalas e sépalas das flores. Mais especificamente, o conjunto de dados Iris consiste em 150 entradas com as seguintes medidas para cada amostra de flor:

- Comprimento da sépala (em centímetros)
- Largura da sépala (em centímetros)
- Comprimento da pétala (em centímetros)
- Largura da pétala (em centímetros)
- Espécie

Nosso objetivo é classificar, através dessas informações, uma flor como setosa ou versicolor.

2.2 Adaptações

Para realizar uma classificação binária usando nossa rede neural, foi necessário realizar um pré-processamento nos dados do conjunto iris, que contém informações de três tipos de flores.Sendo assim retiramos do datasets flores que eram classificadas como virginica. Após isso reduzimos os valores

restantes pela sua média para normaliza-los e dividimos por 10 para a rede neural ter uma melhor interpretação.

Além disso, para reduzir a dimensionalidade do conjunto de dados, aplicamos a técnica de análise de componentes principais (PCA) para obter apenas uma característica relevante.

Tabela 1: Dados antes e depois do pré-processamento

Tabela 2: Dados antes do pré-processamento

Index	**Sepal Length (cm)**	**Sepal Width (cm)**	**Petal Length (cm)**	**Petal Width (cm)**	**Target**
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
⋮	⋮	⋮	⋮	⋮	⋮
95	5.7	3.0	4.2	1.2	1
96	5.7	2.9	4.2	1.3	1
97	6.2	2.9	4.3	1.3	1
98	5.1	2.5	3.0	1.1	1
99	5.7	2.8	4.1	1.3	1

Tabela 3: Dados pós PCA

Index	**Componente principal**	**Target**
0	-0.1461	0
1	-0.1461	0
2	-0.1561	0
3	-0.1361	0
4	-0.1461	0
⋮	⋮	⋮
95	0.1339	1
96	0.1339	1
97	0.1439	1
98	0.0139	1
99	0.1239	1

2.3 Código utilizado

Nesse projeto foi utilizado o scikit-learn para carregar os dataset, numpy para álbebra linear e pandas para facilitar a criação da base de dados final.

```
from sklearn.datasets import load_iris
import pandas as pd
import numpy as np

def PCA(X):
    #Normalizacao
    X = X - X.mean()
    cov = np.cov(X.T)
    eigval, eigvec = np.linalg.eig(cov)
    indexes = np.argsort(eigval)[::-1]
    eigvec_sorted = eigvec[:, indexes]
    componentes_principais = eigvec_sorted[:, :1]
    dados_projetados = np.dot(X, componentes_principais)
    return dados_projetados

iris = load_iris()
df = pd.DataFrame(data=iris.data,columns=iris.feature_names)
df["target"] = iris.target
df = df[df["target"].isin([0,1])]
df_f = df.drop(columns=["target"]) - df.drop(columns=["target"]).mean()
dp = PCA(df.drop(columns=["target"]))
df_dp = pd.DataFrame({'PC': dp[:, 0], 'target': df["target"]})
#Divisao por 10 para facilitar a leitura da rede neural
df_dp["PC"] = df_dp["PC"].apply(lambda x: x/10)
```

3 Função de erro

Nessa etapa, iremos mostrar a função de erro utilizada, nível de tolerancia e código implementado todas essas funcionalidades.

3.1 Erro quadrático médio (MSE)

O erro quadrado médio é uma métrica que quantifica a diferença média entre os valores previstos e os valores reais em um modelo. É calculado como a média dos quadrados das diferenças entre os valores previstos e os valores reais. É amplamente utilizado como uma medida de desempenho para avaliar o quão bem um modelo está fazendo previsões. A fórmula do MSE é dada por:

$$\frac{1}{n} \cdot \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2$$

3.2 Nível de tolerância

Para reduzir o tempo de execução do modelo, podemos definir um valor a partir do qual consideramos o modelo "suficientemente bom". Essa seleção foi empírica e o valor que se mostrou mais eficaz foi um erro abaixo de 10**-9.

3.3 Código utilizado

```
err = (1/len(self.X) * np.sum((np.array(self.variables) - np.
    array(MyGrad)) ** 2))

        if err < self.tol:
            break
```

Essa função está dentro do algoritmo "Back-propagation", e será explicada junto com suas variáveis posteriormente.

4 Sigmoid

Nesta seção, será apresentada a função de ativação utilizada no modelo, juntamente com sua curva obtida a partir do modelo treinado.

4.1 Função de ativação

Uma função de ativação é uma função matemática aplicada em cada neurônio de uma rede neural. A sigmoid é uma função de ativação que mapeia um valor de entrada para um valor no intervalo entre 0 e 1, proporcionando uma interpretação com base em probabilidade. A fórmula da sigmoid se dá por:

$$\frac{1}{1 + \exp^{-x}}$$

4.2 Gráfico da sigmoid

A seguir está o gráfico gerado a partir dos pesos e vieses ajustados pelo modelo. A função geradora e seus parâmetros serão apresentados posteriormente.

4.3 Funções Sigmoid código

As funções para cálcular a sigmoid e o gráfico são respectivamente:

```
def sigmoid(self, x):
    return 1 / (1 + np.exp(-x))

def plot_sigmoid(self):
    x = []
    y = []
    targets = np.linspace(-1, 1, 1000)
    for target in targets:
        x.append(target)
        y.append(self.forward(target))

    plt.plot(x, y)
    plt.title("Sigmoid of the equation")
    plt.xlabel('target')
    plt.ylabel("sigmoid(a6)")
    plt.grid(True)
    plt.show()
    plt.savefig("graph.png")
```

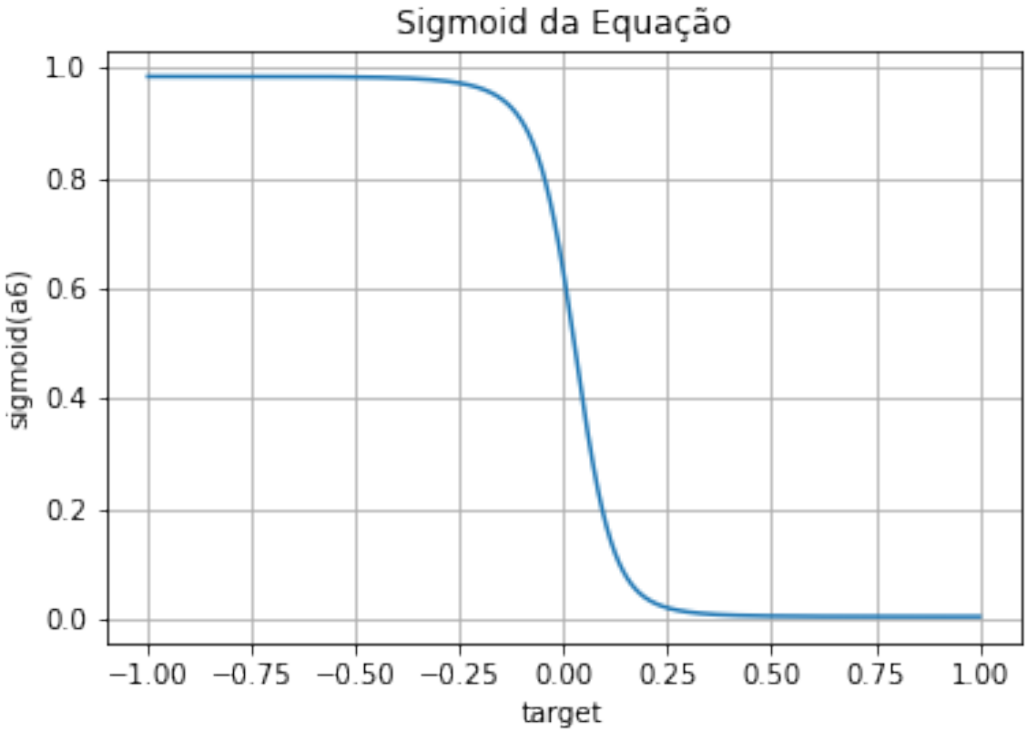

Figura 1: Gráfico sigmoid da função já treinada

5 Modelo

Nessa seção será apresentado a função geradora e explicação do modelo de forma geral.

5.1 Pesos, Viés e Função Geradora

Peso e viés são os elementos principais que utilizamos para ajustar a curva e permitir que o modelo aprenda. No nosso modelo com 2 neurônios, a função geradora seria representada da seguinte forma:

$$\sum_{i=1}^{n}(y_i - (\sigma(w_2 \cdot \sigma(w_1 \cdot x_i + b_1) + b_2))^2$$

Onde σ representa a função de ativação sigmoid, como já apresentada anteriormente.

5.2 Gradiente,função geradora e derivadas parciais

Aqui está o mesmo conjunto de equações representado em notação LaTeX:

Para facilitar na visualizzação, foram usadas as variáveis a_1 e a_2:

$$a_1 = \sigma(w_1 \cdot x + b_1)$$
$$a_2 = \sigma(w_2 \cdot x + b_2)$$

5.2.1 Função Geradora

A função é usada para medir a discrepância entre a saída prevista a_2 e o valor real y durante o treinamento de uma rede neural para problemas de regressão.

$$f(x) = (y - a_2)^2$$

5.2.2 Derivadas Parciais

A derivada parcial é fundamental para determinar a direção e a magnitude das atualizações dos parâmetros durante o treinamento de redes neurais. Ela desempenha um papel crucial na otimização dos modelos e no ajuste dos parâmetros para melhorar o desempenho da rede neural na tarefa específica em que está sendo treinada. Os colchetes significam a equação que será derivada

$$\frac{\partial f}{\partial w_2} = -2(y - a_2)[a_2]$$
$$\frac{\partial f}{\partial w_2} = -2(y - a_2)a_2(1 - a_2)[w_2 a_1 + b_2]$$
$$\frac{\partial f}{\partial w_2} = -2(y - a_2)a_2(1 - a_2)a_1$$
$$\frac{\partial f}{\partial b_2} = -2(y - a_2)[a_2]$$
$$\frac{\partial f}{\partial b_2} = -2(y - a_2)a_2(1 - a_2)[w_2 a_1 + b_2]$$
$$\frac{\partial f}{\partial b_2} = -2(y - a_2)a_2(1 - a_2)$$

$$\frac{\partial f}{\partial w_1} = -2(y - a_2)[a_2]$$
$$\frac{\partial f}{\partial w_1} = -2(y - a_2)a_2(1 - a_2)[w_2a_1 + b_2]$$
$$\frac{\partial f}{\partial w_1} = -2(y - a_2)a_2(1 - a_2)w_2[a_1]$$
$$\frac{\partial f}{\partial w_1} = -2(y - a_2)a_2(1 - a_2)w_2a_1(1 - a_1)[w_1x + b_1]$$
$$\frac{\partial f}{\partial w_1} = -2(y - a_2)a_2(1 - a_2)w_2a_1(1 - a_1)x$$
$$\frac{\partial f}{\partial b_1} = -2(y - a_2)[a_2]$$
$$\frac{\partial f}{\partial b_1} = -2(y - a_2)a_2(1 - a_2)[w_2a_1 + b_2]$$
$$\frac{\partial f}{\partial b_1} = -2(y - a_2)a_2(1 - a_2)w_2[a_1]$$
$$\frac{\partial f}{\partial b_1} = -2(y - a_2)a_2(1 - a_2)w_2a_1(1 - a_1)$$

5.2.3 Gradiente

O gradiente é crucial para a otimização e treinamento de redes neurais, fornecendo informações sobre como os parâmetros devem ser ajustados para melhorar o desempenho da rede e convergir para uma solução ótima.

$$\nabla f(x) = (-2 \cdot (y - a_2) \cdot a_2 \cdot (1 - a_2) \cdot a_1, -2 \cdot (y - a_2) \cdot a_2 \cdot (1 - a_2),$$
$$-2 \cdot (y - a_2) \cdot a_2 \cdot (1 - a_2) \cdot w_2 \cdot a_1 \cdot (1 - a_1) \cdot x, -2 \cdot (y - a_2) \cdot a_2 \cdot (1 - a_2) \cdot w_2 \cdot a_1 \cdot (1 - a_1))$$

5.3 Definição de peso e viés dentro do código

Toda a estrutura da rede neural é feita a partir de uma class, feita em python, nessa, que ao executar o método de iniciação da classe acaba por selecionar peso é viés "aleatórios"

```
class NeuralNetwork:
 def __init__(self, lr=0.1, epochs=10**4, seed=42, tol=10**-6):
     self.lr = lr
     self.epochs = epochs
     self.initialize_parameters()
     np.random.seed(seed)
     self.tol = tol


 def initialize_parameters(self):
     self.w1 = np.random.randn()
     self.w2 = np.random.randn()
     self.b1 = np.random.randn()
     self.b2 = np.random.randn()
     self.variables = [self.w1, self.w2, self.b1, self.b2]
```

Além disso ele contém outro método para carregar os dados de treinamento e chamar a função de "Back Porpagation"que será apresentada mais a frente nessa seção.

```
 def fit(self, X, y):
     self.X = X
```

```
        self.y = y
        self.back_prop()
```

5.4 Gradiente no código

O gradiente no código é similar ao apresentado anteriormente, eis aqui a função:

```
    def grad_sig(self):
        a1 = self.sigmoid(self.w1 * self.X + self.b1)
        a2 = self.sigmoid(self.w2 * a1 + self.b2)

        delw2 = -2 * np.sum((self.y - a2) * a2 * (1 - a2) * a1)
        delb2 = -2 * np.sum((self.y - a2) * a2 * (1 - a2))

        delw1 = -2 * np.sum((self.y - a2) * a2 * (1 - a2) * self.w2 *
            a1 * (1 - a1) * self.X)

        delb1 = -2 * np.sum((self.y - a2) * a2 * (1 - a2) * self.w2 *
            a1 * (1 - a1))

        return delw1, delw2, delb1, delb2
```

5.5 Função Foward

A função foward executa a prórpia função geradora. Ela é utilizada para realizar a predição dos valores.

```
    def forward(self, target):
        a1 = self.sigmoid(self.w1 * target + self.b1)
        a2 = self.sigmoid(self.w2 * a1 + self.b2)
        return a2
```

5.6 Função de Back propagation

Esse método realiza o gradiente descendente, o que faz o ajuste da função.

```
    def back_prop(self):
        count = 0
        for _ in range(self.epochs):
            count += 1

            # Calculo dos gradientes
            delw1, delw2, delb1, delb2 = self.grad_sig()

            # Atualizacaoo dos pesos e vieses
            self.w1 -= self.lr * delw1
            self.w2 -= self.lr * delw2
            self.b1 -= self.lr * delb1
            self.b2 -= self.lr * delb2
            MyGrad = [self.w1, self.w2, self.b1, self.b2]

            err = np.sum((np.array(self.variables) - np.array(MyGrad))
                ** 2) * 1/len(self.X)
            if err < self.tol:
                break
            self.variables = MyGrad
        print(err,count)
```

Nele cálculamos os valores de gradiente descendente e realizamos a função de custa (também chamada de função de erro), com os valores anteriores e os valores do gradiente descendente

6 Parâmetros

Nessa seção irei apresentar todas os parâmetros iniciais dos pesos e vieses.

6.1 Pesos e vieses inicias

Os parâmetros utilizados na rede neural (pesos e vieses) são "aleatórias", porém através do metódo seed do numpy é possível replica-lo, os valores iniciais são:

- w2 = 0.6363051083451703
- w1 = -0.9067206685799298
- b2 = 0.4760425874269718
- b1 = 1.303661268398584

6.2 Parâmetros inicias da classe

A classe da rede neural recebe alguns parâmetros opcionais, sendo eles o learning rate, epochs (número de interação máximo) seed (seed de aleatoriedade) e tol (tolerância), por padrão esses valores são:

- lr = 0.1
- epochs = 10**4
- seed = 42
- tol = 10**-6

7 Resultado e discussão

Nessa seção irei apresentar os resultados e discussões.

7.1 Parâmetros finais

- w2 = -9.090309342824183
- w1 = 9.163017310075196
- b2 = -0.44300472918309697
- b1 = 4.069485700042025

7.2 Resultados

Para testar os resultados peguei uma amostra de valores para utilizar como parâmetro de acerto:

Tabela 4: Valores de PC e Target

Index	**PC**	**Target**
48	0.153612	0
28	0.160399	0
85	-0.181492	1
83	-0.245625	1
14	0.168659	0
9	0.159598	0
42	0.191528	0
21	0.153420	0
75	-0.192389	1
93	-0.043732	1

Prevendo com esses valores os resultados são:

Tabela 5: Valores de PC e Target

Index	**Previsão**
48	0.0750090556304501
28	0.06763777115060508
85	0.9561889331270907
83	0.9705061383062704
14	0.05982039669938942
9	0.06846088538328919
42	0.043427940678462715
21	0.07523044539460501
75	0.9595928610113245
93	0.7917581881483361

Isso demonstra que o modelo está de fato capaz de classificar com alta precisão. Apesar de alguns resultados mais próximos de zero, o modelo apresenta maior dificuldade em classificar nesses casos, mas ainda assim indica uma probabilidade relativamente alta.