EXERCÍCIO 02

Uma possibilidade de aplicação de redes neurais é treinar um perceptron (neurônio artificial) para realizar alguma operação conhecida de modo que se possa prever e analisar o desempenho do treinamento de maneira objetiva. O objetivo deste segundo exercício é treinar um perceptron para realizar a soma de dois números. Para tal utilizaremos um perceptron com duas entradas e uma saída. As duas entradas estão representando os dois números a serem somados, e a saída representa o resultado da soma.

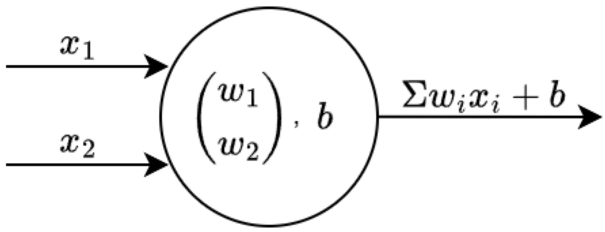


Figura 01: Representação do nosso perceptron que soma dois números.

Claro, é sempre importante notar que o "treino" aqui se refere ao procedimento iterativo de atualização dos pesos e vieses (bias) associados ao nosso perceptron. A ideia é que a saída do perceptron pode ser escrita como:

$$z = W \cdot x + b$$

Onde W é a matriz de pesos de cada ligação, x são os inputs do vetor de entrada e b é o bias associado ao perceptron em si. Escrevendo a operação na sua forma explícita temos:

$$z = \Sigma w_i x_i + b = w_1 x_1 + w_2 x_2 + b$$

Assim, se a nossa intenção é treinar um perceptron que seja capaz de realizar uma soma de dois números x_1 e x_2, é natural que, ao longo do treinamento, se espere que o bias convirja para zero, bem como os pesos w_1 e w_2 convirjam para um. Deste modo, a saída do perceptron tende a se aproximar objetivamente da soma entre os dois números.

Para executar este treinamento eu, primeiramente, defini uma classe de modo a estruturar o perceptron como um objeto da classe Perceptron. Dentro desta classe foram, também, definidos métodos que possibilitam inserir entradas, calcular a saída e atualizar a matriz de pesos cada vez que a rede (neste caso composta por um único perceptron) for executada. As figuras 2 a 5 contém o código, em python utilizado.

No código da figura 2 a classe e os métodos são criados. Na figura 3 alguns parâmetros da rede são definidos e o perceptron é criado utilizando a atribuição da classe. Já na figura 4 uma lista de entradas é elaborada. Por fim, a figura 5 contém o código utilizado para, a partir da lista de entradas, realizar o treinamento do perceptron e criar um gráfico da função de custo à medida que as iterações ocorrem.

```python
import numpy as np
import matplotlib.pyplot as plt

class Perceptron:
    def __init__(self, entradas=3, saida=1, bias=0.5):
        self.entradas = np.zeros(entradas)
        self.saida = np.zeros(saida)
        #self.pesos = np.random.uniform(low = -1, high = +1, size = entradas)
        self.pesos = [0.7, 0.3]
        self.bias = np.random.uniform(low = -1, high = +1, size = 1)
        self.saida_registro = []  # Lista para armazenar os valores de saída
        self.bias_registro = []   # Lista para armazenar os valores de bias
        self.pesos_registro = []  # Lista para armazenar os valores de pesos
        self.entrada_registro = []# Lista para armazenar os valores de entradas

    def input(self, array_in):
        for i, x_i in enumerate(array_in):
            self.entradas[i] = x_i

    def ativa(self):
        self.saida = np.dot(self.entradas, self.pesos) + self.bias
        self.entrada_registro.append(self.entradas)
        self.saida_registro.append(self.saida)  # Registra o valor de saída
        self.bias_registro.append(self.bias)    # Registra o valor do bias
        self.pesos_registro.append(self.pesos.copy())  # Registra os valores de pesos (cópia)
        return self.saida, self.bias, self.pesos

    def calculaCusto(self, entradas, saidas, alvo):
        delta = alvo - self.saida
        custo = np.dot(delta, delta)
        return custo

    def atualizaPesos(self, entradas, taxa_de_aprendizado=0.01, alvo=0):
        erro = alvo - self.saida
        for i in range(len(self.pesos)):
            self.pesos[i] += taxa_de_aprendizado * erro * entradas[i]
        self.bias += taxa_de_aprendizado * erro


    def __str__(self):
        return f'Entradas: {self.entradas}, Pesos: {self.pesos}, Saída: {self.saida}, Bias: {self.bias}'
```

Figura 02: Definição da classe “Perceptron()” em python.

```python
# Informações do treino
taxa_de_aprendizado = 0.001

# Informações do perceptron criado
num_entradas = 2
num_saidas = 1
perceptron = Perceptron(num_entradas, num_saidas)
```

Figura 03: Parâmetros de entrada do perceptron

```python
lista_de_entradas = []
# Cria lista de entradas
for i1 in range(10):
    for i2 in range(10):
        lista_de_entradas.append([i1*1.0,i2*1.0])
print(f'Lista de entradas:{lista_de_entradas}')
```

Figura 04: Criação da lista de entrada para treino

```python
# Iterando sobre várias entradas
for k in range(5):
    for i, entrada in enumerate(lista_de_entradas):
        perceptron.input(entrada)
        saida, bias, pesos = perceptron.ativa()
        alvo = entrada[0]+entrada[1]
        custo = perceptron.calculaCusto(entrada,saida,alvo)

        print(f'\n# Lote ({k}) # Rodada ({i+1}) #\n')
        print(f'Entrada: {entrada}')
        print(f'Pesos: {pesos}, Bis: {bias}')
        print(f'Saida: {saida}, Alvo: {alvo}')
        print(f'\nPesos antigos: {perceptron.pesos}')

        perceptron.atualizaPesos(entrada,taxa_de_aprendizado,alvo)

        print(f'Pesos atualizados: {perceptron.pesos}\n')

        # Adicionando os dados para o plot
        x_dados.append(i+1 + k*len(lista_de_entradas))
        y_dados.append(perceptron.entradas[1])
        custo_dados.append(custo)
```

Figura 05: Processo de treinamento do perceptron.

O índice i se refere a um dos inputs da lista de entradas, enquanto o índice k se refere à reexecução para o mesmo conjunto de entradas (chamo, aqui, este conjunto de “lote”).

		x_1	x_2
k = 1	i = 1	0	0
	i = 2	1	0
	i = 3	0	1
	i = 4	1	1
k = 2	i = 1	0	0
	i = 2	1	0
	i = 3	0	1
	i = 4	1	1

Tabela 01: Exemplo de entradas

Neste exemplo, o loop interno será executado 4 vezes, uma para cada par de números, e os parâmetros x_1 e x_2 serão passados para o perceptron, mas o perceptron pode ser treinado utilizando novamente este mesmo conjunto de entradas. Esta estratégia não me parece, em geral, ideal para o treinamento de uma rede, visto que é interessante que o treino ocorra com uma vasta quantidade de dados distintos, de modo que o procedimento de backpropagation garanta uma utilização adequada dos pesos.

Opto, no entanto, por utilizar o mesmo conjunto de número mais de uma vez como estratégia para visualizar a dinâmica do erro de saída do perceptron. Assim é possível verificar que o erro diminui cada vez que as mesmas operações são executadas.

Ao executar o código eu crio uma lista com 100 entradas (pares de números) da forma:

$$\textbf{Entrada: } [x_1, x_2] \mid x_i \in [0, 10],$$

Estas entradas são passadas para o perceptron é um valor de saída calculado. O "custo" (erro) é calculado a partir da diferença entre o valor obtido no perceptron e o valor esperado. E em seguida o perceptron tem seus pesos corrigidos baseados no custo. A intenção é que os pesos sejam ajustados de modo a minimizar a função de custo.

O seguinte gráfico foi obtido acompanhando a dinâmica do custo em função na iteração.

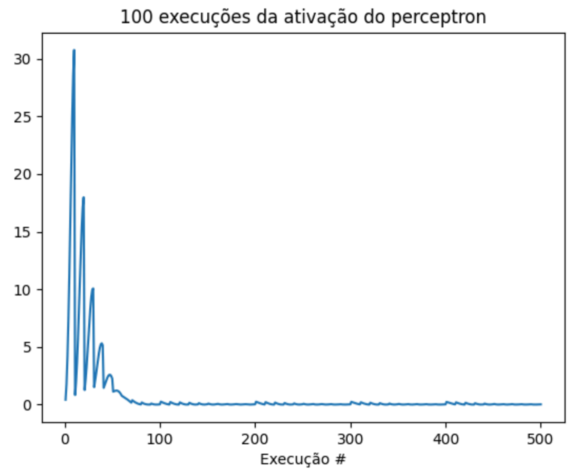


Figura 06: Função custo plotada para cada época do perceptron.

No gráfico da figura acima podemos notar que foram executadas 500 épocas do perceptron (uma época consiste em um ciclo completo de input, output e atualização dos pesos). No entanto, utilizei, $k = 10$ de modo que o perceptron recebe 5 vezes o mesmo conjunto de 100 entradas.

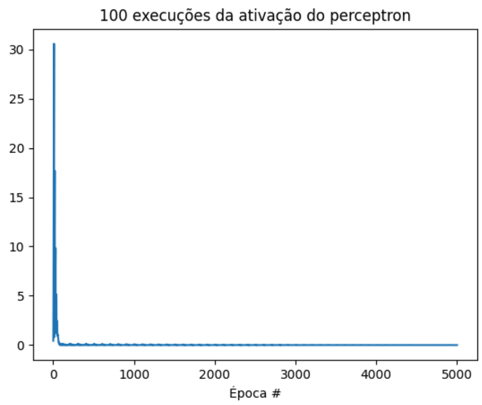
É possível perceber, portanto, que o custo da rede converge adequadamente para zero.

Em cada iteração do código eu executo um print com informações úteis no seguinte formato:

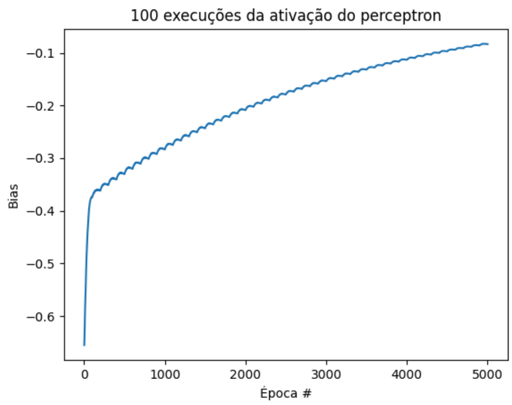
k = 1 , **i** = 1	**Entrada**: [0.0, 0.0] **Pesos**: [0.7, 0.3], **Bias**: [-0.33927183] **Saída**: [-0.33927183] **Alvo**: 0.0
k = 10 , **i** = 100	**Entrada**: [9.0, 9.0] **Pesos**: [1.00548057, 1.00422378], **Bias**: [-0.06788874] **Saída**: [18.01945041] **Alvo**: 18.0

Tabela 02: Primeiro e último retorno do código no prompt de comando

Executei também para k = 50:



Convergência da função custo para 0



Convergência do bias para 0

Figura 07: Teste de convergência para 50 lotes de treino com 100 inputs

Na figura 7 (B) é possível perceber que a minha hipótese inicial estava correta: utilizar o mesmo conjunto de inputs para continuar o treino não apresenta muita eficiência. Podemos notar que durante o treino no primeiro lote o bias inicia em $b =- 0.65643806$e finaliza em$b =- 0.37393168$. Posteriormente todos os próximos 49 lotes é notável que a inclinação da curva diminui substancialmente e, ao final dos 50 lotes o valor do bias converge para $b =- 0.08338204$.