Perceptron multicamada do zero em Python — Porta lógica XOR

Alef Matias · Follow

6 min read · Jul 24, 2023

Considerações iniciais

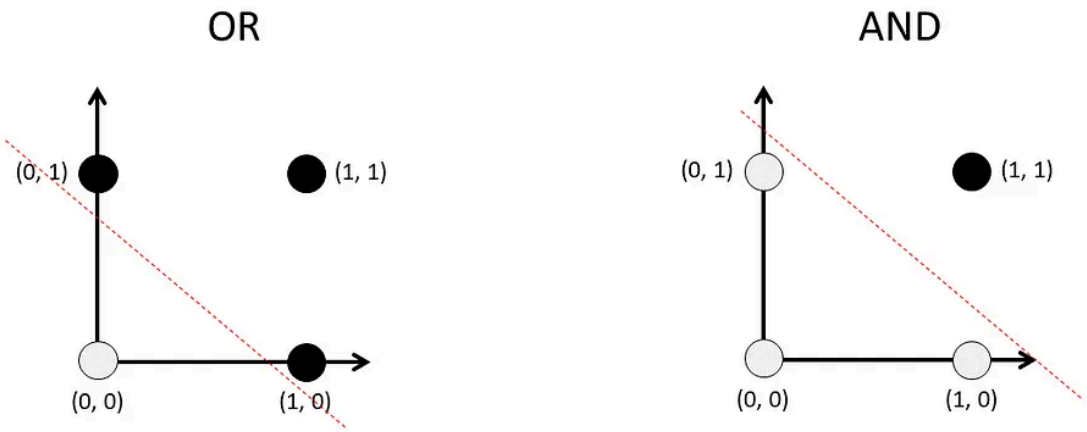
Dando continuidade na saga de estudos sobre Redes Neurais Artificiais, o presente artigo tem a missão de abordar um assunto clássico envolvendo aprendizado de máquina. Na postagem anterior houve menção do funcionamento do Perceptron, a forma mais básica do que conhecemos hoje como Rede Neural Artificial. Por se tratar da forma mais simples de representação do tema, o que facilita o aprendizado, o Perceptron é aplicado na resolução de problemas mais simples, como a separação dos dados em apenas duas classes distintas. O que acontece se precisarmos resolver um problema um pouco mais complexo? É possível utilizar essa rede como ferramenta? É o que será debatido (de forma simples) nos parágrafos seguintes.

O problema

Como citado, o Perceptron é capaz de auxiliar na resolução de problemas mais simples e apenas tem a capacidade de classificar os dados em duas classes. No artigo anterior, pudemos treinar o algoritmo para aprender as portas lógicas AND e OR e, nessa aplicação, o modelo se encaixa perfeitamente. Porém se estendemos o

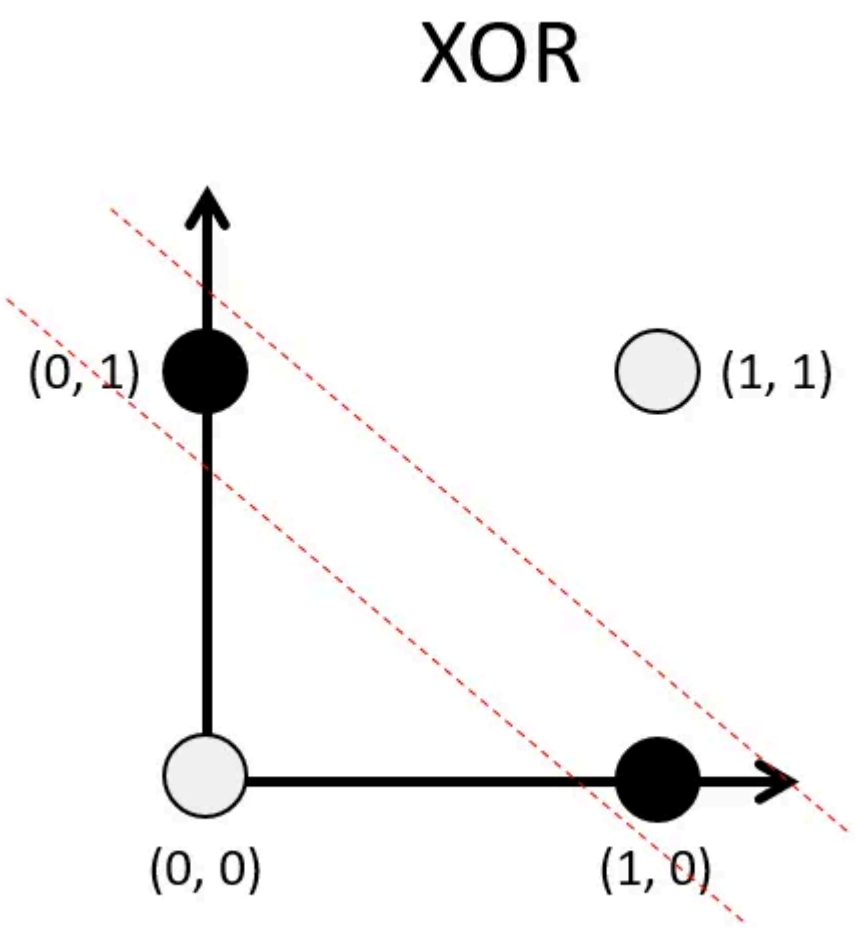
treinamento para outras opções de portas lógicas, vemos que o algoritmo é incapaz de atingir a convergência e estimar os parâmetros necessários para alcançar o melhor resultado possível.

Isso ocorre por um motivo específico: no Perceptron de apenas uma camada, os dados assumem o status de serem linearmente separáveis, ou seja, uma única linha é capaz de dividir os dados em duas classes. Quando utilizamos as portas AND e OR, conseguimos visualmente realizar essa divisão.



Divisão das portas OR e AND

Já para a porta XOR (ou exclusivo), não é possível realizar a separação dos dados apenas com uma linha, como demonstrado na imagem abaixo, são necessárias duas linhas para fazer a classificação corretamente. O problema da porta XOR foi o problema clássico que tornou necessário a evolução do Perceptron para o que conhecemos como Perceptron multicamada.

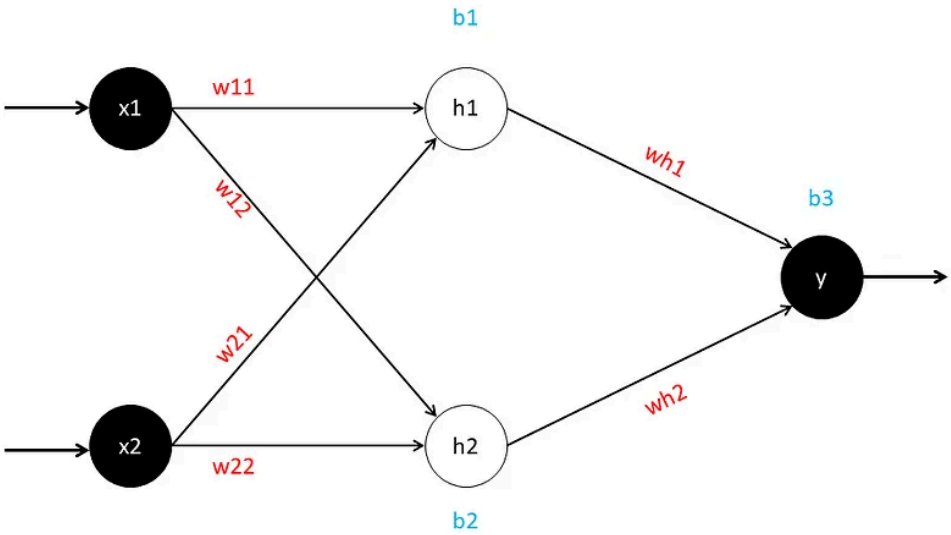


Divisão da porta XOR

Criando o modelo

Como na postagem anterior, aqui iremos criar o modelo da forma mais simples, sem o uso de bibliotecas específicas para realização dos cálculos (exceto a biblioteca *Numpy* para gerar os números aleatórios e executar o cálculo exponencial), lembrando também que o código não está 100% otimizado e serve para visualizarmos passo a passo como os cálculos são realizados, ou seja, foi modelado propositalmente para entendimento de cada etapa do algoritmo.

Antes de nos adentrarmos no código em si, é importante visualizarmos a arquitetura da rede, facilitando o entendimento.



Arquitetura da rede neural.

Onde:

- **x1** e **x2**: dados de entrada.
- **w11**, **w12**, **w21**, **w22**, **wh1** e **wh2**: pesos.
- **b1**, **b2**, e **b3**: bias.

Importamos a biblioteca *Numpy* e inicializamos a classe MLP.

```
import numpy as np

class MLP:
  def __init__(self):
    pass
```

Definimos a função (método) de treinamento do algoritmo e inicializamos as variáveis.

```
def train(self, inputs, outputs, alpha, epochs):
  # Aqui incluímos os parâmetros que a função train deverá receber
  self.inputs = inputs
```

```
    self.outputs = outputs
    self.alpha = alpha # taxa de aprendizado do algoritmo
    self.epochs = epochs # iterações na etapa de treinamento

    # Neste bloco definimos os pesos e bias que serão inicializados aletatoriamer
    w11 = np.random.uniform(0, 1)
    w12 = np.random.uniform(0, 1)
    w21 = np.random.uniform(0, 1)
    w22 = np.random.uniform(0, 1)

    wh1 = np.random.uniform(0, 1)
    wh2 = np.random.uniform(0, 1)

    b1 = np.random.uniform(0, 1)
    b2 = np.random.uniform(0, 1)
    b3 = np.random.uniform(0, 1)
```

O próximo passo se dá por iniciar os laços de repetição necessários para realizar o treinamento com os dados de entrada. É nessa etapa onde todos os cálculos são feitos para depois obtermos os valores necessários para a etapa de teste.

```
for i in range(epochs):
  for j in range(len(inputs)):
    # Execução dos cálculos lineares da camada de entrada que serão passados ac
    # Utilizada a função de ativação sigmoid
    h1 = 1 / (1 + np.exp(- ((inputs[j][0] * w11) + (inputs[j][1] * w21) + b1)))
    h2 = 1 / (1 + np.exp(- ((inputs[j][0] * w12) + (inputs[j][1] * w22) + b2)))

    # A partir dos valores obtidos na camada oculta, aplicamos a função de ativ
    y = 1 / (1 + np.exp(- ((h1 * wh1) + (h2 * wh2) + b3)))

    # Calculamos o erro
    error = outputs[j][0] - y

    # Nessa etapa entram alguns conceitos mais densos que envolvem o temido cál
    # Por se tratar de um assunto extenso, não entrarei em detalhes, pois esse
    # Efetuamos o cálculo das derivadas parciais
    derivative_y = y * (1 - y) * error
    derivative_h1 = h1 * (1 - h1) * wh1 * derivative_y
    derivative_h2 = h2 * (1 - h2) * wh2 * derivative_y

    # Cálculo dos deltas que serão usados para atualização dos pesos
    delta_w11 = alpha *derivative_h1 * inputs[j][0]
    delta_w12 = alpha *derivative_h2 * inputs[j][0]
    delta_w21 = alpha *derivative_h1 * inputs[j][1]
    delta_w22 = alpha *derivative_h2 * inputs[j][1]
```

```
        delta_b1 = alpha * derivative_h1
        delta_b2 = alpha * derivative_h2
        delta_b3 = alpha * derivative_y

        delta_wh1 = alpha * derivative_y * h1
        delta_wh2 = alpha * derivative_y * h2

        # Atualizando os pesos e retornando os pesos obtidos após o treinamento
        w11 += delta_w11
        w12 += delta_w12
        w21 += delta_w21
        w22 += delta_w22

        wh1 += delta_wh1
        wh2 += delta_wh2

        b1 += delta_b1
        b2 += delta_b2
        b3 += delta_b3

    return w11, w12, w21, w22, wh1, wh2, b1, b2, b3
```

Com todos os cálculos realizados, podemos prosseguir para a etapa de teste do algoritmo. Nesta função, apenas definimos o mesmo cálculo a partir da função de ativação (*sigmoid*) usando os valores retornados na função de treinamento.

```
def predict(self, weights, x1, x2):
  hidden1 = 1 / (1 + np.exp(- ((x1 * weights[0]) + (x2 * weights[2]) + weights[
  hidden2 = 1 / (1 + np.exp(- ((x1 * weights[1]) + (x2 * weights[3]) + weights[

  # Note que nos cálculos anteriores da função sigmoid, retornávamos o valor pu
  return 1 if 1 / (1 + np.exp(- ((hidden1 * weights[4]) + (hidden2 * weights[5]
```

Após toda a parte árdua de construção do modelo, finalmente podemos testá-lo.

```
# Dados de entrada e saída baseados na tabela verdade usando a porta lógica XOR
inputs = [[0, 0], [0, 1], [1, 0], [1, 1]]
outputs = [[0], [1], [1], [0]]

mlp = MLP()
```

```
# Executamos o treinamento da rede com uma taxa de aprendizado de 5% e 10.000 é
tr = mlp.train(inputs, outputs, 0.05, 10000)
# Testamos o algoritmo utilizando as entradas 1 e 1, se o algoritmo foi treinac
y = mlp.predict(tr, 1, 1)
print(y)
```

Considerações finais

Considerando todo o conhecimento necessário de cálculo e álgebra linear para entendermos de fato como as Redes Neurais Artificiais funcionam, podemos ver que criar uma rede do zero é totalmente possível com poucas linhas de código e sem bibliotecas dedicadas. A princípio parece um bicho de sete cabeças (ainda mais quando envolvemos derivadas e álgebra linear), mas quando analisamos as entranhas do seu funcionamento, podemos ver que uma rede simples se torna nossa aliada no quesito de aprendizado.

O exemplo aqui utilizado para exemplificação é amplamente usado para demonstrar o real funcionamento das redes neurais e trazer a tona o motivo da necessidade da inclusão de mais camadas. Outros exemplos de dados podem ser aplicados nesse modelo, ficando de lição de casa testar e realizar os aprimoramentos necessários para cada tipo de uso.

Até mais!

Perceptron | Multilayer Perceptron | Machine Learning | Neural Networks

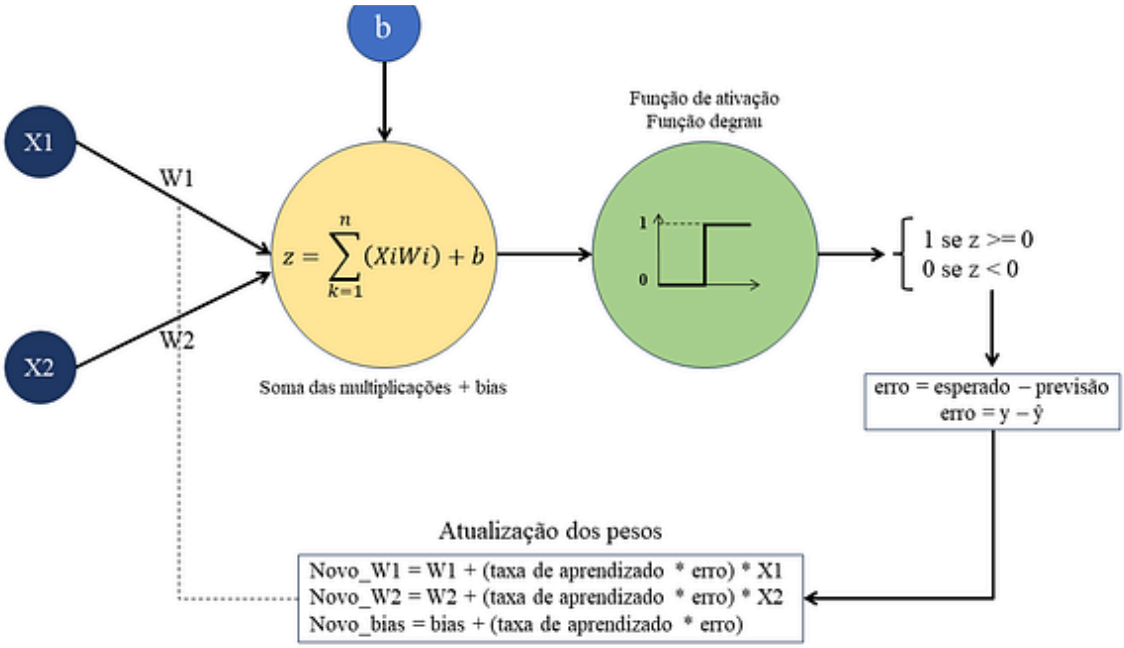
Follow

Written by Alef Matias

3 Followers

Cientista de Dados e Pós Graduando em Inteligência Artificial.

More from Alef Matias

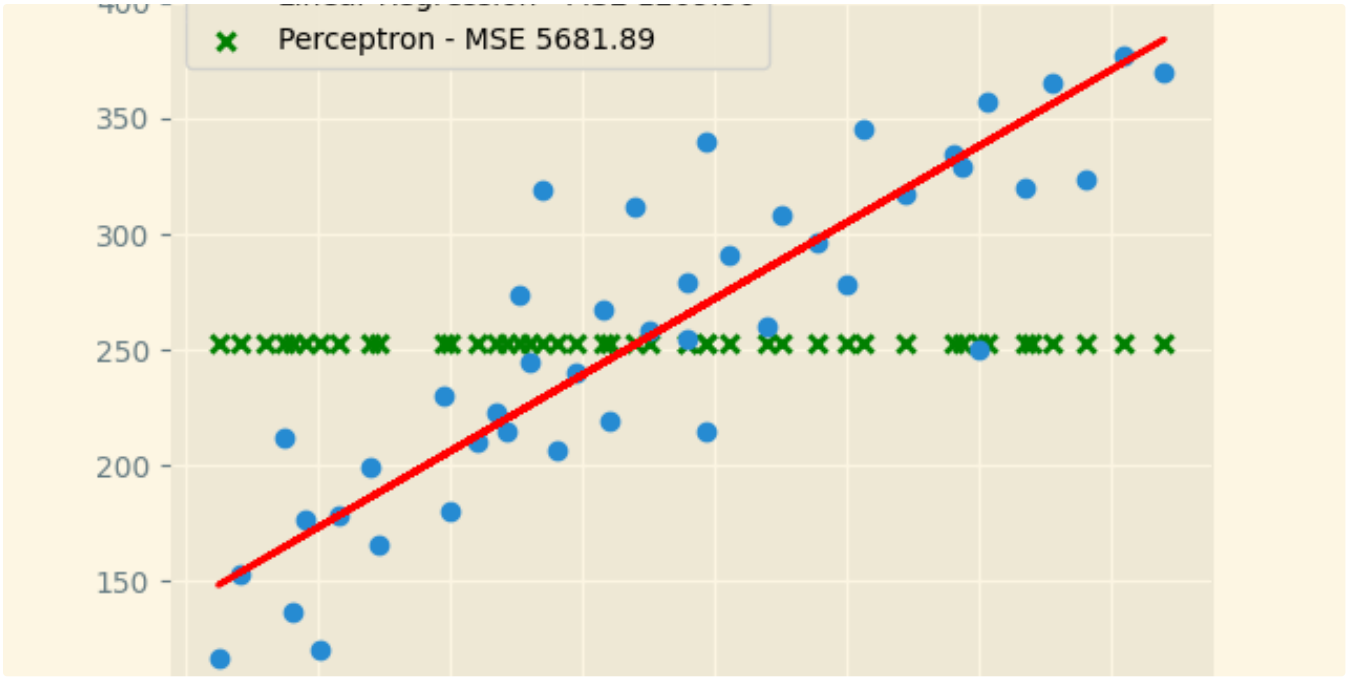


Alef Matias

Criando um Perceptron do zero com Python

Introdução

7 min read · Jul 11, 2023



Alef Matias

Comparação entre Regressão Linear e Rede Neural (Perceptron)

Conceitos iniciais

5 min read · Jan 21, 2024

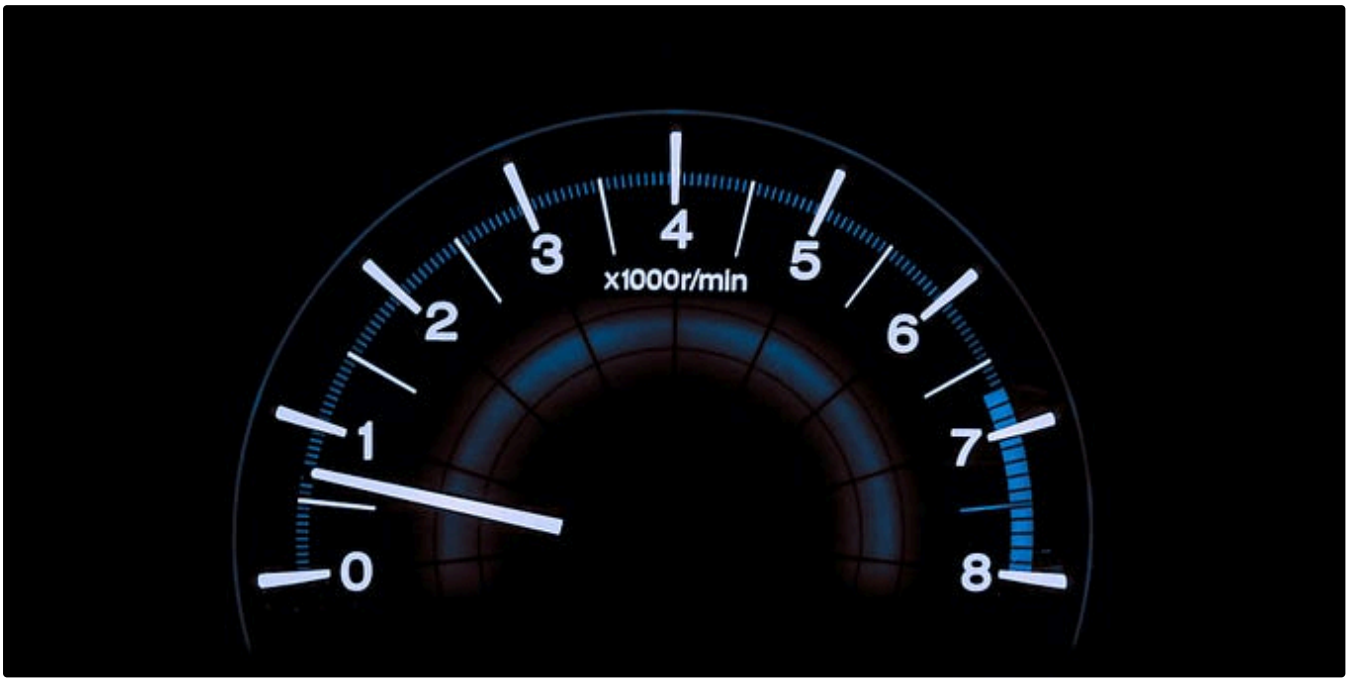
Alef Matias

Como um CPF é validado?—Algoritmo em Python

Introdução

4 min read · Oct 23, 2023

6



Alef Matias

Comparativo de velocidade entre linguagens de programação — Python vs. C

Blá blá blá inicial

3 min read · Sep 13, 2023

1

See all from Alef Matias

Recommended from Medium

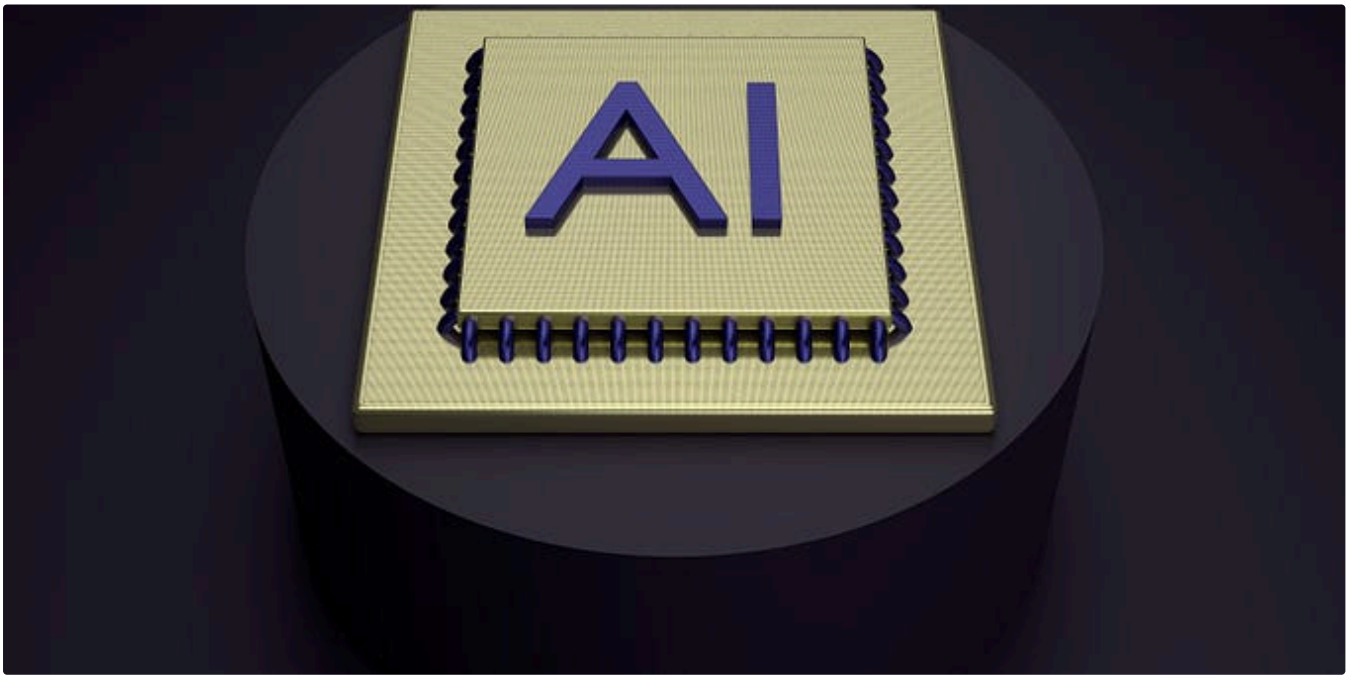
Mansi in Code Like A Girl

Coding Your First Neural Network FROM SCRATCH

A step by step guide to building your own Neural Network using NumPy.

9 min read · Oct 26, 2023

192 1



buddy(talk with me at patelharsh7458@gmail.com)

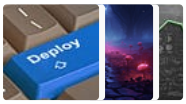
AI—Simple implementation of a feedforward neural network with one hidden layer (XOR Problem)

Creating an artificial neural network (ANN) from scratch in Python can be quite an involved process, but I'll provide you with a simple...

4 min read · Nov 8, 2023

3

Lists



Predictive Modeling w/ Python

20 stories · 1035 saves

Practical Guides to Machine Learning

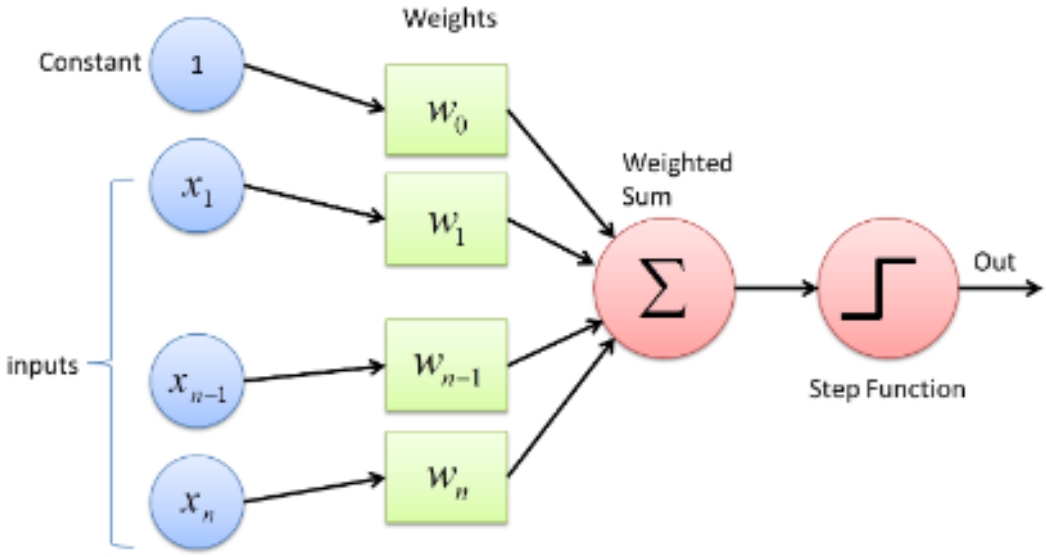
10 stories · 1237 saves

Natural Language Processing

1320 stories · 811 saves

The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 348 saves



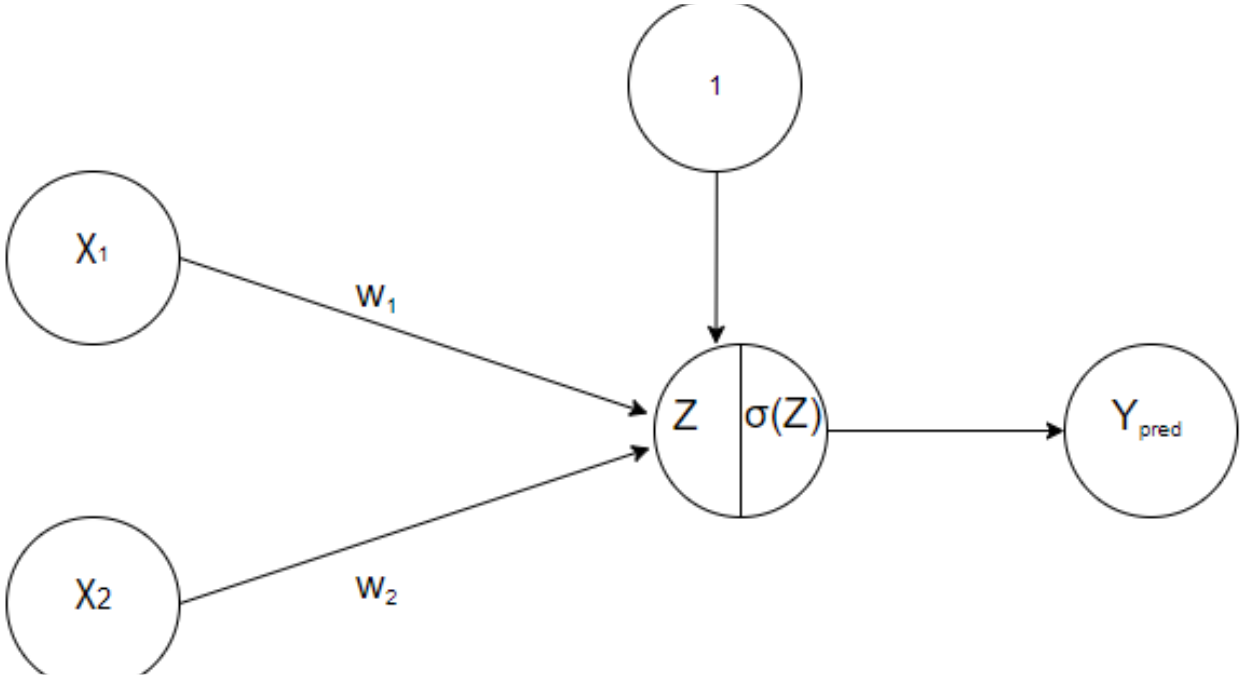
Joseph Robinson, Ph.D. in Towards Data Science

From Basic Gates to Deep Neural Networks: The Definitive Perceptron Tutorial

Demystifying Mathematics, Binary Classification, and Logic Gates

· 21 min read · Apr 28, 2023

466 3



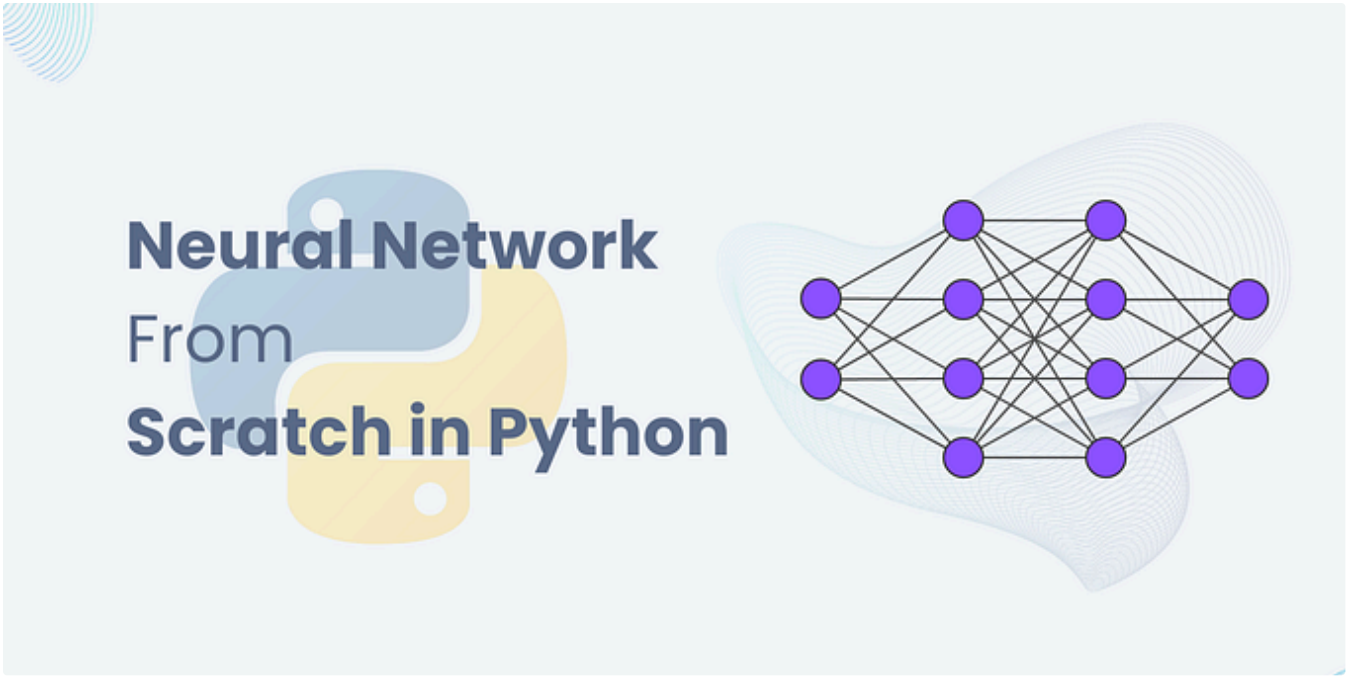
Kamel SOUDANI

Building an Artificial Neural Network Model by hand

ANN with one hidden layer and one formal neuron trained using backpropagation algorithm

8 min read · Jan 8, 2024

5



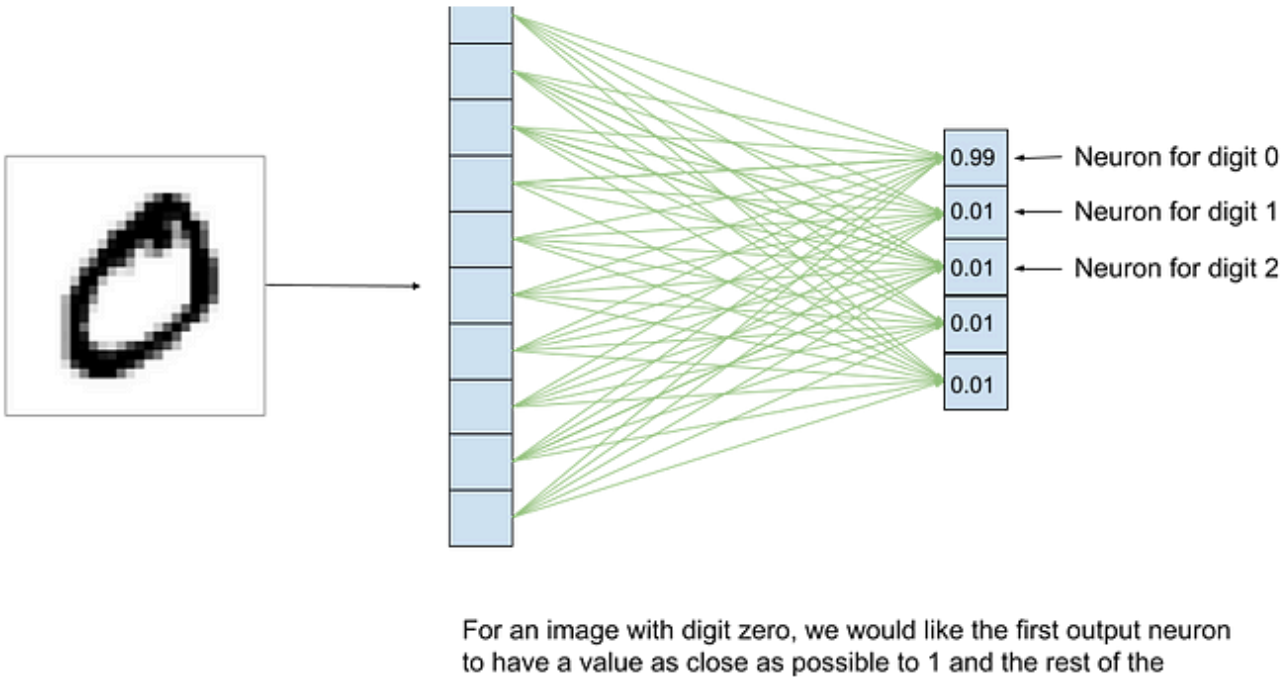
BILAL_AI

Building a Neural Network from Scratch: Your Step-by-Step Guide

Learn the fundamentals of deep learning and build your very own neural network

6 min read · Oct 1, 2023

27 1



Serban Liviu

Backpropagation step by step

Backpropagation

11 min read · Nov 30, 2023

1