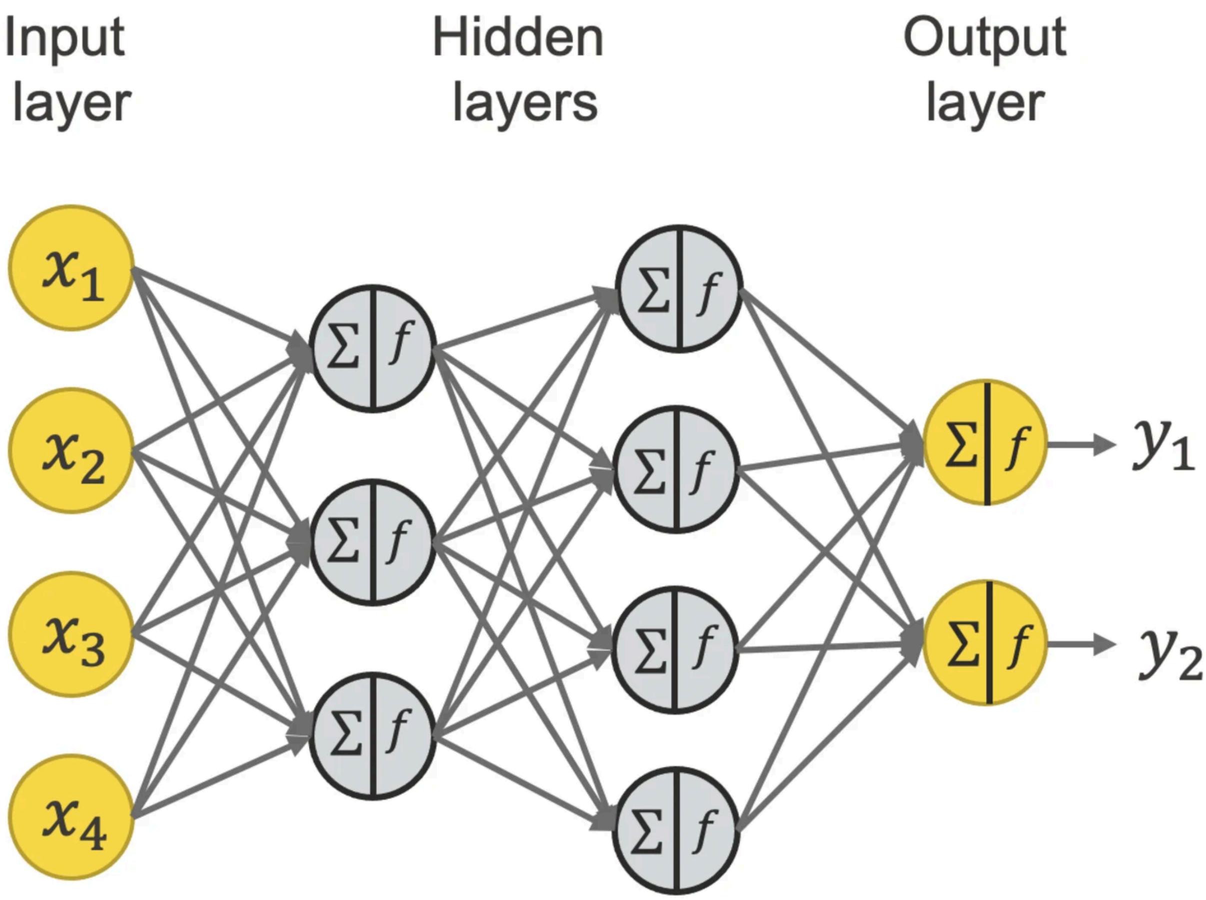
Redes Neurais Artificiais



links uteis:

- https://array-3d-viz.vercel.app/
- https://arrayvis.netlify.app/

---

1. O que são redes neurais?

As redes neurais são um tipo de modelo computacional inspirado no funcionamento do cérebro humano, especificamente na forma como os neurônios se conectam e processam informações. Elas são uma das técnicas mais poderosas e populares no campo do machine learning (aprendizado de máquina) e são usadas para resolver uma ampla variedade de problemas, como reconhecimento de imagens, processamento de linguagem natural, previsão de séries temporais e muito mais.

1.1. Inspiração biológica

As redes neurais são inspiradas no sistema nervovo humano:

- **Neurônios biológicos:** No cérebro, os neurônios são células que processam e transmitem informações por meio de conexões chamadas *sinapses*.
- **Neurônios artificiais:** Em redes neurais, os neurônios artificiais (também chamados de unicades ou nós) são modelos matemáticos que simulam o comportamento dos neurônios biológicos.

2. Estrutura de uma rede neural artificial

Uma rede neural é composta por camadas de neurônios artificiais interconectados. Essas camadas são organizadas da seguinte forma:

1. **Camada de entrada (Input Layer):**

- Recebe os dados de entrada (por exemplo: pixels de uma imagem ou palavras de um texto)
- Cada neurônio na camada de entrada representa uma característica (*feature*) dos dados

2. **Camadas Ocultas (Hidden Layers):**

- São camadas intermediárias entre a entrada e a saída
- Cada neurônio em uma camada oculta processa as entradas recebidas e passa o resultado para a próxima camada

3. **Camada de Saída (Output Layers):**

- Produz o resultado final da rede (por exemplo, a classe de uma imagem ou o valor de uma previsão)
- O número de neurônios na camada de saída depende do problema (por exemplo, 1 neurônio para regressão ou vários neurônios para classificação)

2.1. Funcionamento

1. **Propagação (Feed Forward)**:

- Os dados de entrada são passados pela rede, camada por camada
- Cada neurônio calcula uma soma ponderada das entradas, aplica uma *função de ativação* e passa o resultado adiante

2. **Função de Ativação:**

- Introduz não-linearidade na rede, permitindo que ela aprenda padrões complexos
- Exemplos comuns: ReLU, sigmoide, tanh

3. **Cálculo da Perda (Loss):**

- A saída da rede é comparada com o valor real (rótulo) para calcular o erro (ou perda)

4. **Retropropagação (Backpropagation):**

- O erro é propagado de volta pela rede, ajustando os pesos e viéses dos neurônios para minimizar a perda

5. **Atualização dos Pesos:**

- Os pesos são ajustados usando algoritmos de otimização, como grandiende descendente (gradient descent)

3. Tipos de redes neurais

1. **Redes Neurais Feedforward**:

- A informação flui em uma única direção, da entrada para a saída.
- Usadas para problemas simples, como classificação e regressão.

2. **Redes Neurais Convolucionais (CNNs)**:

- Especializadas em processar dados com estrutura espacial, como imagens.
- Usam **filtros convolucionais** para extrair características.

3. **Redes Neurais Recorrentes (RNNs)**:

- Projetadas para trabalhar com sequências de dados, como texto ou séries temporais.
- Têm "memória" para processar informações ao longo do tempo.

4. **Redes Neurais de Memória de Longo Prazo (LSTMs)**:

- Uma variação das RNNs, capazes de aprender dependências de longo prazo.

5. **Redes Neurais Generativas (GANs)**:

- Compostas por duas redes (geradora e discriminadora) que competem entre si.
- Usadas para gerar dados novos, como imagens ou textos.

4. Underfitting e Overfitting

4.1. Overfitting (Sobreajuste)

O **overfitting** ocorre quando o modelo aprende demais os detalhes e ruídos dos dados de treinamento, a ponto de prejudicar seu desempenho em dados novos (não vistos).

Características do Overfitting:

- O modelo tem um desempenho excelente nos dados de treinamento, mas ruim nos dados de teste/validação.
- O modelo "decora" os dados de treinamento em vez de aprender padrões gerais.
- Geralmente acontece quando o modelo é muito complexo (muitos parâmetros) em relação à quantidade de dados disponíveis.

Exemplo:

Imagine um modelo que tenta prever se uma fruta é uma maçã ou uma laranja. Se o modelo se ajustar demais aos dados de treinamento, ele pode começar a considerar características irrelevantes, como pequenas marcas ou manchas nas frutas, em vez de focar em características gerais como cor e formato.

Como evitar o Overfitting:

- **Regularização**: Técnicas como L1/L2 regularization penalizam pesos muito grandes no modelo.
- **Dropout**: Desativa aleatoriamente neurônios durante o treinamento para evitar dependência excessiva em características específicas.
- **Aumento de Dados (Data Augmentation)**: Gera novas versões dos dados de treinamento (por exemplo, rotacionar imagens) para aumentar a diversidade.
- **Validação Cruzada**: Avalia o desempenho do modelo em diferentes subconjuntos dos dados.
- **Redução da Complexidade do Modelo**: Usar menos camadas ou neurônios em redes neurais.
- **Early Stopping**: Interromper o treinamento quando o desempenho no conjunto de validação para de melhorar.

4.2. Underfitting (Subajuste)

O **underfitting** ocorre quando o modelo é muito simples para capturar os padrões subjacentes nos dados, resultando em um desempenho ruim tanto nos dados de treinamento quanto nos dados de teste/validação.

Características do Underfitting:

- O modelo tem desempenho ruim tanto nos dados de treinamento quanto nos dados de teste.
- O modelo não consegue aprender relações importantes nos dados.
- Geralmente acontece quando o modelo é muito simples (poucos parâmetros) ou o treinamento é insuficiente.

Exemplo:

Usando o mesmo exemplo das frutas, um modelo subajustado pode não conseguir distinguir entre maçãs e laranjas porque não aprendeu características básicas como cor e formato.

Como evitar o Underfitting:

- **Aumentar a Complexidade do Modelo**: Adicionar mais camadas ou neurônios em redes neurais.
- **Treinar por Mais Tempo**: Permitir que o modelo aprenda por mais épocas.
- **Engenharia de Features**: Criar características mais relevantes para o modelo.
- **Reduzir Regularização**: Diminuir a penalização sobre os pesos do modelo.

---

5. Pesos

Em uma rede neural, pesos são números que irão ser multiplicados pelos valores de entrada. Os pesos, basicamente, representam a importância relativa de cada entrada na rede neural.

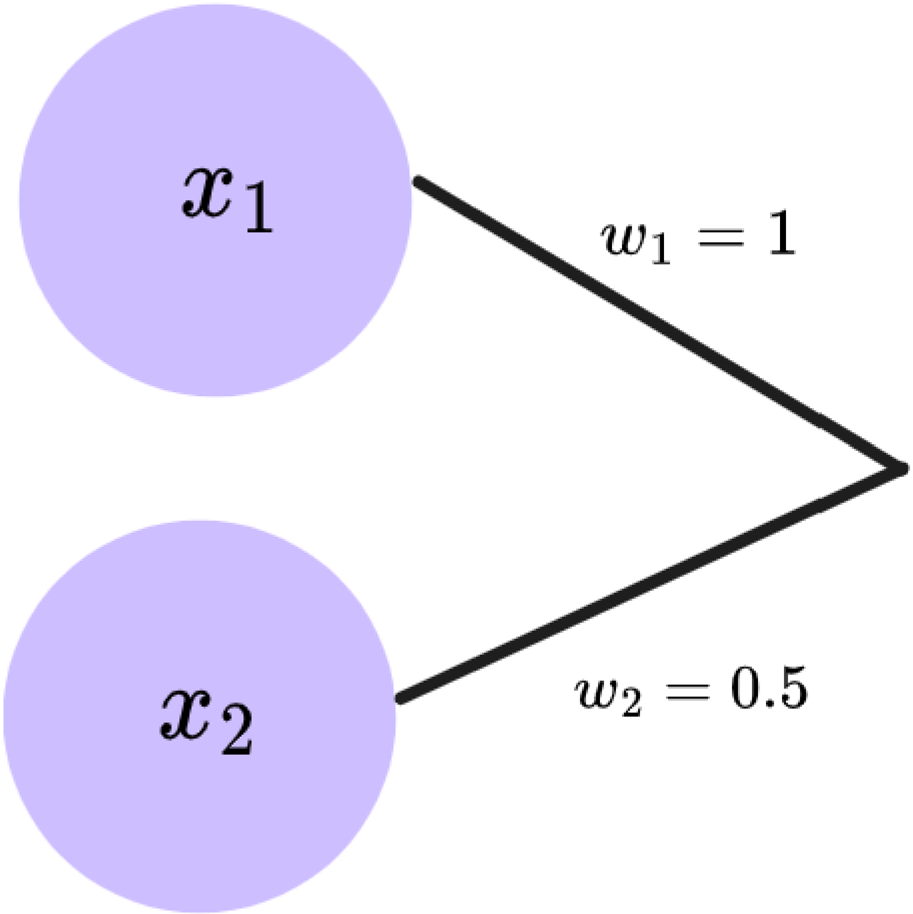
Os pesos determinam como as entradas influenciam os valores de saída do modelo.

Exemplo

Vamos imaginar que temos um modelo que prevê preços de uma casa em base:

$$x_1 \to Tamanho$$
$$x_2 \to Quantidade\ de\ quartos$$

se $w_1 = 100\ e\ w_2 = 50$, isso significa que o tamanho da casa tem mais influência no preço dela do que a quantidade de quartos. Visualmente isso seria algo como:

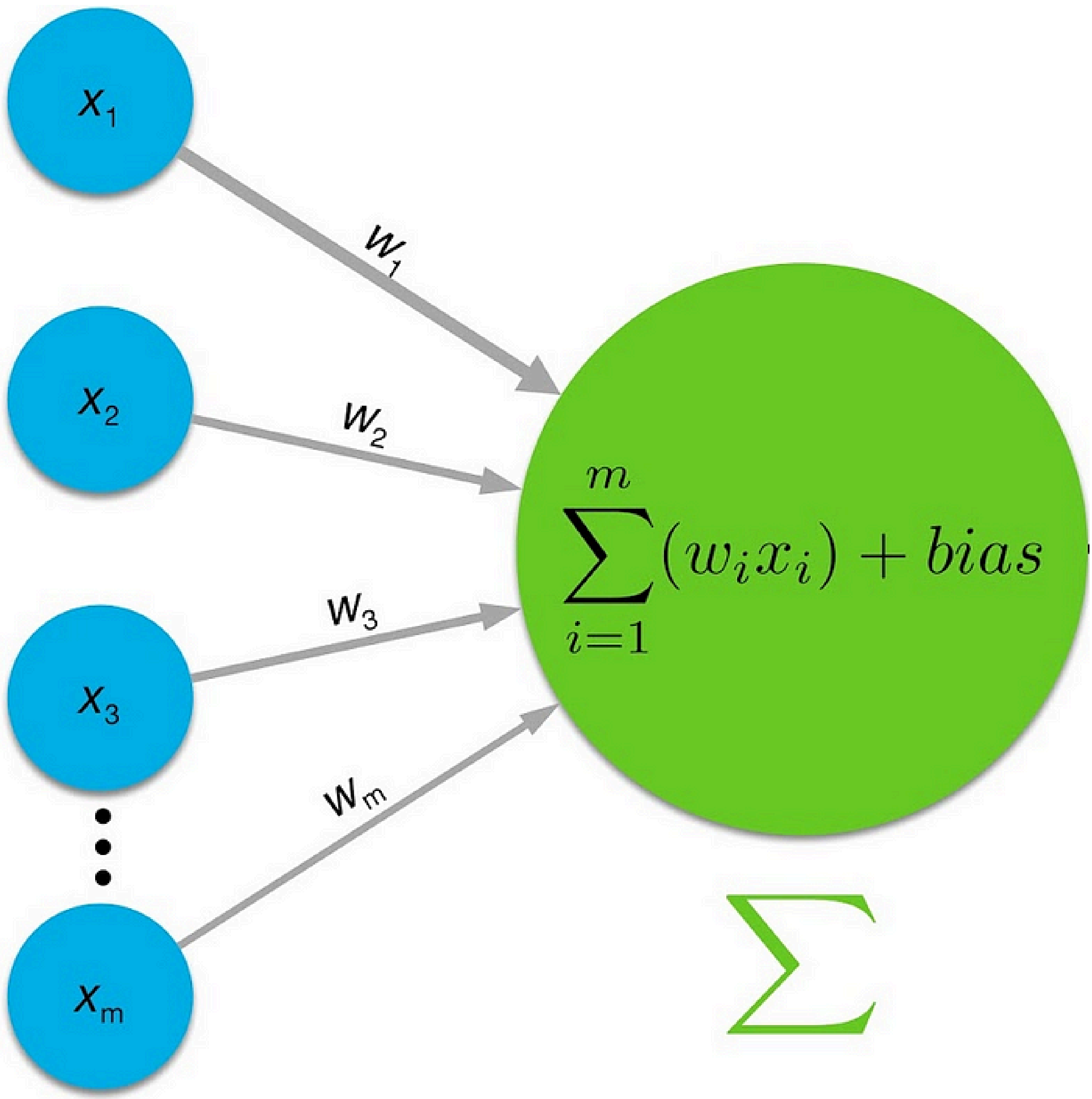


É claro que ao treinar o modelo, ajustamos esses pesos com alguns outros algoritmos e equações.

6. Bias (viés)

O viés é basicamente um valor que ajusta a saída sem depender da entrada. O viés ajuda o modelo a se ajustar melhor aos dados, especialmente quando as entradas são zero.

Na equação do perceptron: $z = x_1 * w_1 + x_2 * w_2 + x_3 * w_3 \cdots + b$ ou $\sum_{i=1}^{n}(x_i * w_i) + b$



No exemplo da previsão do preço da casa viso em Pesos, o viés pode representar um custo base (por exemplo, o valor do terreno) que não depende do tamanho da casa ou do número de quartos.

Assim como os pesos, o viés pode ser ajustado durante o treinamento do modelo para torná-lo mais preciso.

7. Array shape (numpy)

O array shape ou "formato do array", por assim dizer, refere-se a forma ou estrutura do array, ou seja, às suas dimensões e ao tamanho em cada dimensão. O shape é a propriedade fundamental para entender e manipular arrays, especialmente em operações matemáticas e de machine learning.

O que é um array shape?

- O shape descreve as dimensões de um array, ou seja, como os seus elementos são organizados
- É representado por uma tupla, onde cada valor indica o tamanho da dimensão correspondente
- Por exemplo, um array com o shape de (3, 4) tem 3 linhas e 4 colunas, ou seja, é uma lista com três listas dentro e cada lista tem quatro elementos:

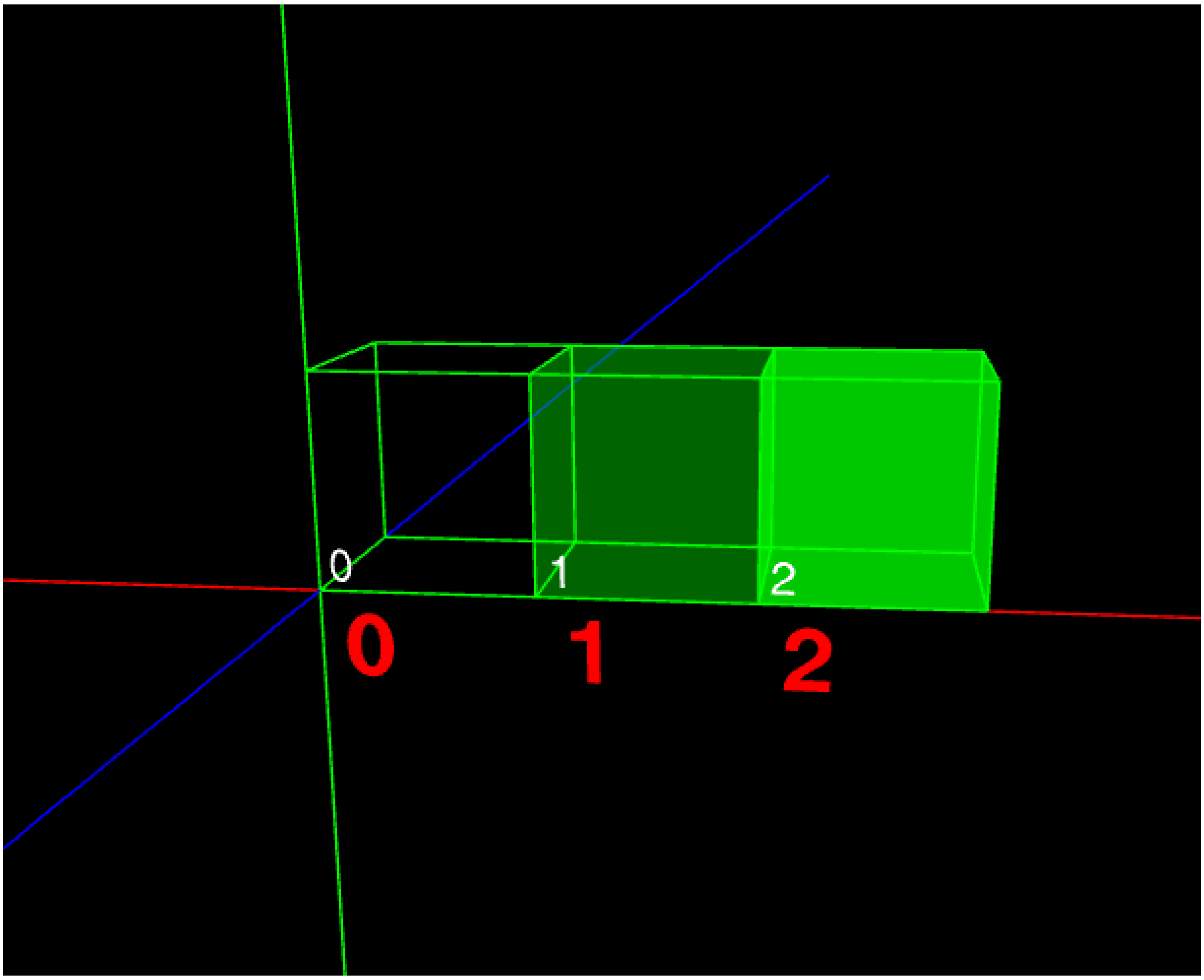
```
# Shape(3, 4):
arrayShape34 = [[0, 1, 2, 3],
                [0, 1, 2, 3],
                [0, 1, 2, 3]]
```

7.1.Vetor

Um array de apenas uma dimensão `(3,)`, por exemplo, pode ser chamado de vetor:

```
import numpy as np
arrayShape1D = np.array([0, 1, 2])

print(arrayShape1D.shape) #(3,)
```

Visualmente, seria algo como:

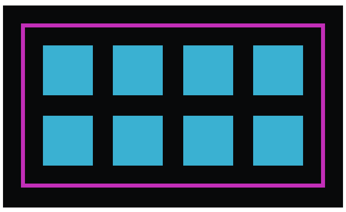


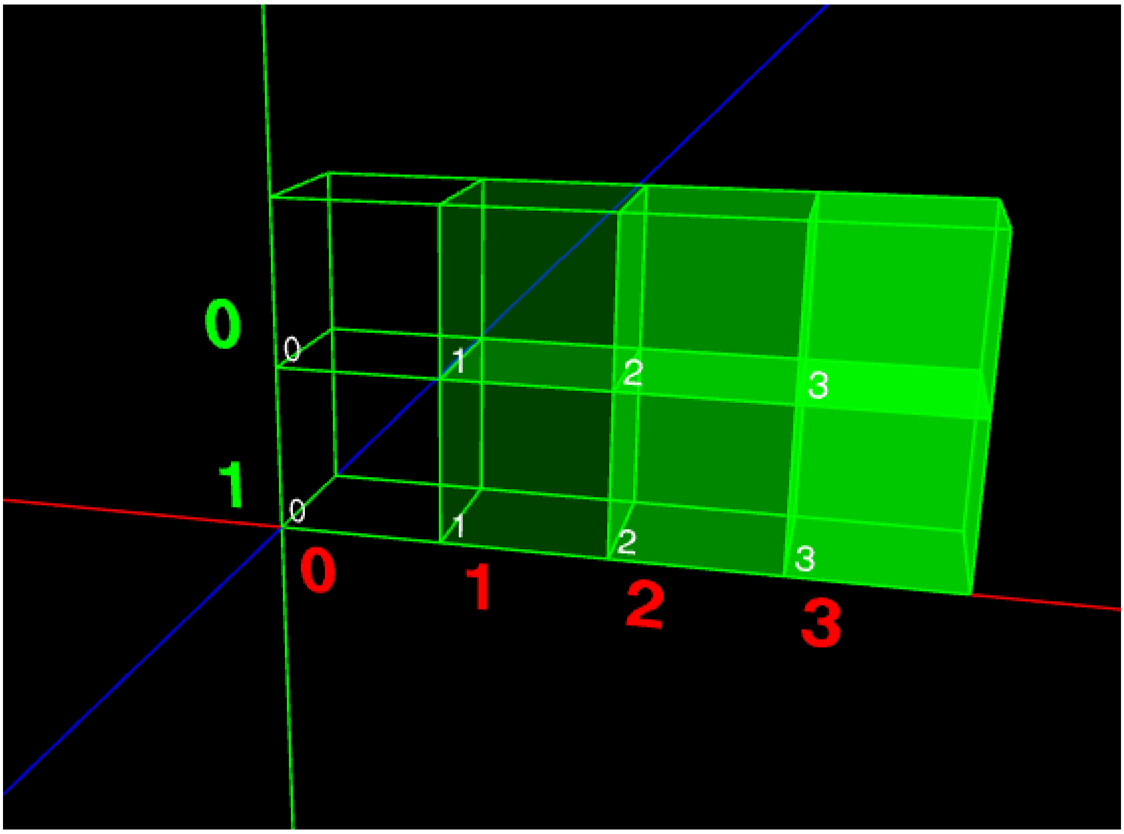
7.2. Matriz

Um array com duas dimensões sendo linha e coluna, `(2, 4)`, pode ser chamado de Matriz:

```
import numpy as np
arrayShape2D = np.array([0, 1, 2, 3],
                        [0, 1, 2, 3])

print(arrayShape2D.shape) #(2, 4)
```

Visualmente, seria algo como:



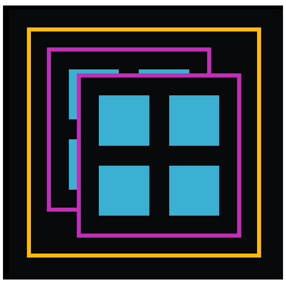


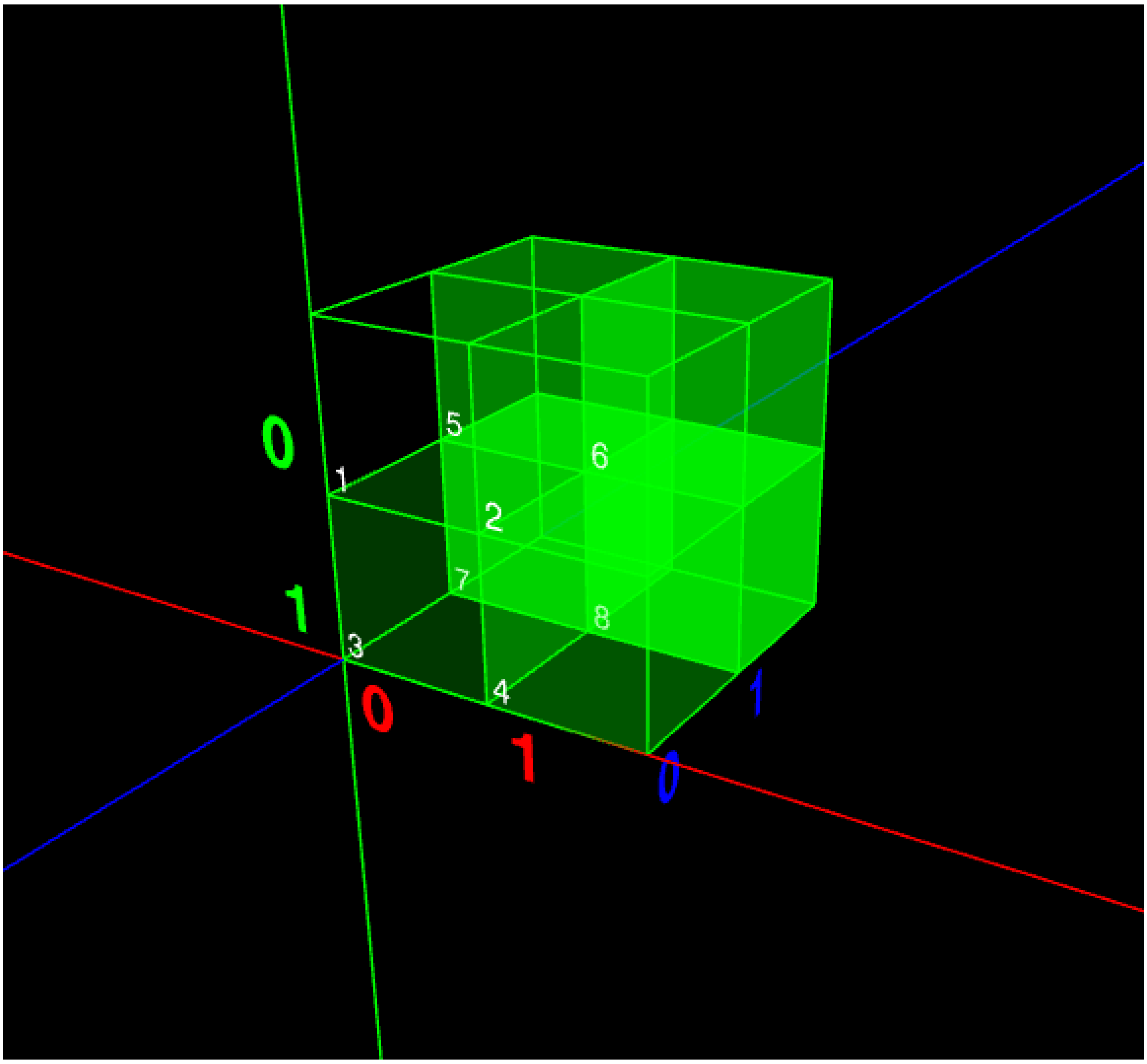
7.3. Tensor

Um array com três dimensões sendo profundidade, linha e coluna `(2, 2, 2)`, pode ser chamado de Tensor:

```
import numpy as np
arrayShape3D = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

print(arrayShape3D.shape) #(2, 2, 2)
```

Visualmente, seria algo como:



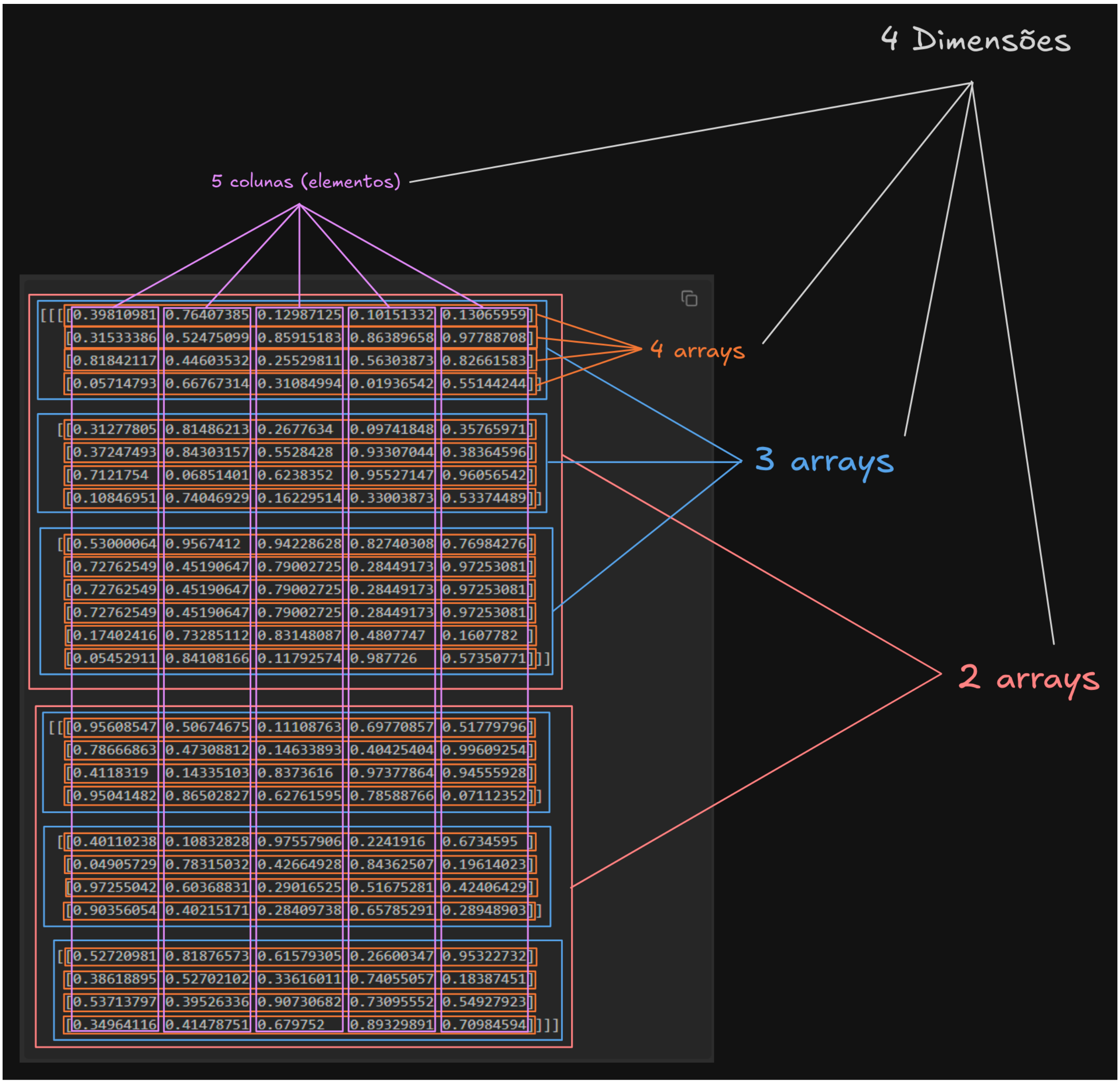


Para arrays com dimensões superiores a três, o shape continua seguindo a mesma lógica, com um valor para cada dimensão.

```
import numpy as np
arr = np.random.rand(2, 3, 4, 5) # Array 4D

print(arr.shape) # (2, 3, 4, 5)
```

```
[[[[0.39810981 0.76407385 0.12987125 0.10151332 0.13065959]
   [0.31533386 0.52475099 0.85915183 0.86389658 0.97788708]
   [0.81842117 0.44603532 0.25529811 0.56303873 0.82661583]
   [0.05714793 0.66767314 0.31084994 0.01936542 0.55144244]]

  [[0.31277805 0.81486213 0.2677634  0.09741848 0.35765971]
   [0.37247493 0.84303157 0.5528428  0.93307044 0.38364596]
   [0.7121754  0.06851401 0.6238352  0.95527147 0.96056542]
   [0.10846951 0.74046929 0.16229514 0.33003873 0.53374489]]
```

```
  [[0.53000064 0.9567412  0.94228628 0.82740308 0.76984276]
   [0.72762549 0.45190647 0.79002725 0.28449173 0.97253081]
   [0.72762549 0.45190647 0.79002725 0.28449173 0.97253081]
   [0.72762549 0.45190647 0.79002725 0.28449173 0.97253081]
   [0.17402416 0.73285112 0.83148087 0.4807747  0.1607782 ]
   [0.05452911 0.84108166 0.11792574 0.987726   0.57350771]]]


 [[[0.95608547 0.50674675 0.11108763 0.69770857 0.51779796]
   [0.78666863 0.47308812 0.14633893 0.40425404 0.99609254]
   [0.4118319  0.14335103 0.8373616  0.97377864 0.94555928]
   [0.95041482 0.86502827 0.62761595 0.78588766 0.07112352]]

  [[0.40110238 0.10832828 0.97557906 0.2241916  0.6734595 ]
   [0.04905729 0.78315032 0.42664928 0.84362507 0.19614023]
   [0.97255042 0.60368831 0.29016525 0.51675281 0.42406429]
   [0.90356054 0.40215171 0.28409738 0.65785291 0.28948903]]

  [[0.52720981 0.81876573 0.61579305 0.26600347 0.95322732]
   [0.38618895 0.52702102 0.33616011 0.74055057 0.18387451]
   [0.53713797 0.39526336 0.90730682 0.73095552 0.54927923]
   [0.34964116 0.41478751 0.679752   0.89329891 0.70984594]]]]
```



8. Produto escalar (dot product)

Dot Product

$$\begin{bmatrix} a & b \end{bmatrix} \bullet \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \end{bmatrix}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \bullet \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \bullet \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} aw + by & ax + bz \\ cw + dy & cx + dz \end{bmatrix}$$

O produto escalar de dois arrays nada mais é do que a soma do produto dos elementos que ocupam a mesma posição nos dois arrays.

Exemplo:

```
import numpy as np
arr_1 = np.array([1, 2, 3])
arr_2 = np.array([3, 2, 1])

print(np.dot(arr_1, arr_2)) # 10
```

$$\sum_{i=1}^{n} a_i * b_i$$

$$(1 * 3) + (2 * 2) + (3 * 1) = 3 + 4 + 3 = 10$$

Porém devemos tomar cuidado ao realizar o produto escalar de dois arrays com dimensões opostas, por exemplo, vamos pensar que eu tenha um `arr_1` como shape (4,) e outro array `arr_2` com o shape (3, 4):

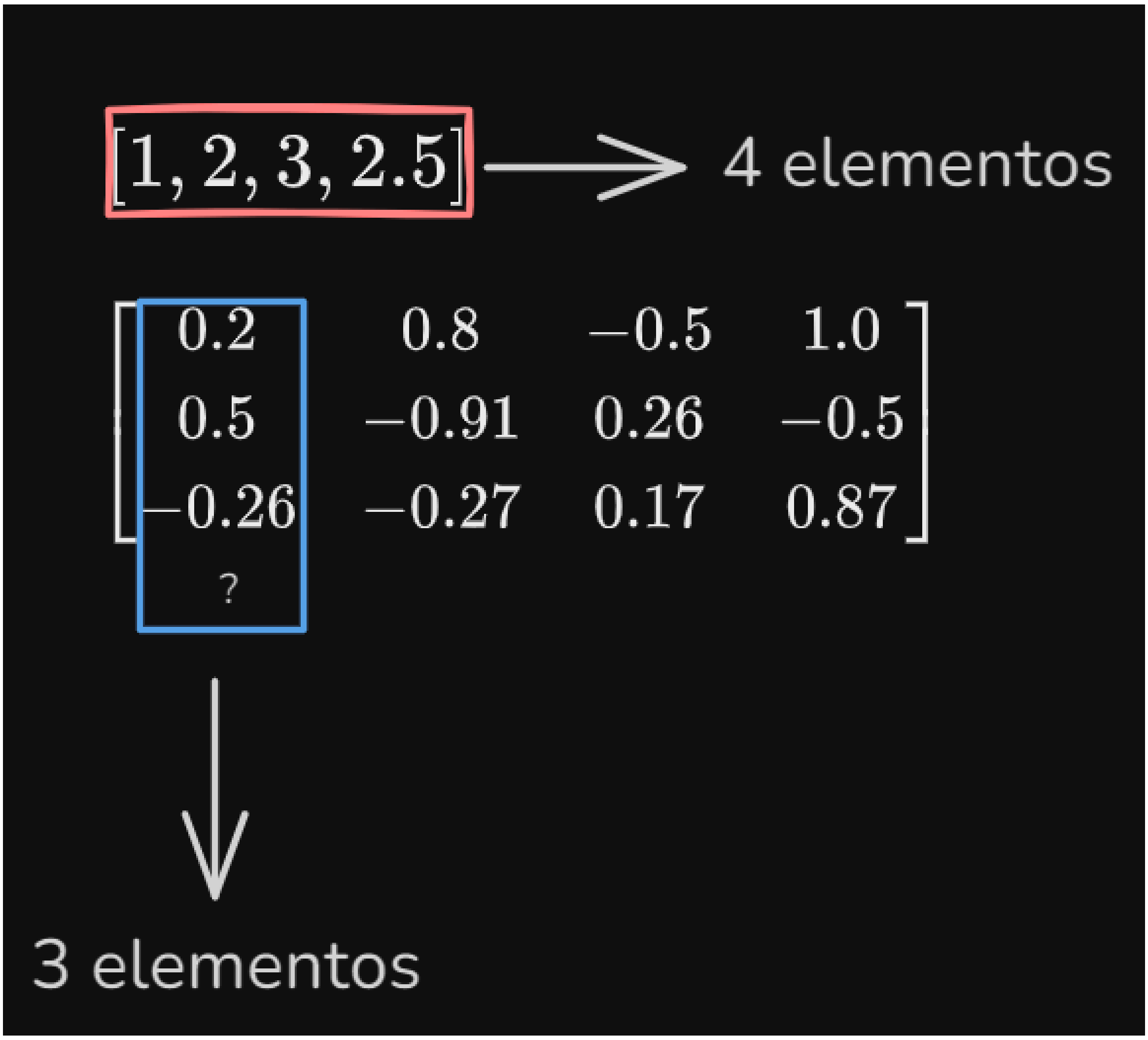
```
import numpy as np
```

```python
arr_1 = np.array([1, 2, 3, 2.5])

arr_2 = np.array([[0.2, 0.8, -0.5, 1.0],
                  [0.5, -0.91, 0.26, -0.5],
                  [-0.26, -0.27, 0.17, 0.87]])
```

`arr_1`: $[1, 2, 3, 2.5]$
`arr_2`:

$$\begin{bmatrix} 0.2 & 0.8 & -0.5 & 1.0 \\ 0.5 & -0.91 & 0.26 & -0.5 \\ -0.26 & -0.27 & 0.17 & 0.87 \end{bmatrix}$$

Aritmeticamente, esse cálculo não seria possível, mas por que? Bom, se você tentar calcular o produto escalar diretamente, teriamos que multiplicar cada delemento do `arr_1` com todos elementos da primeira coluna do `arr_2`, seria algo como:



$$1 * 0.2 \ + 2 * 0.5 \ + 3 * -0.26 \ + 2.5 * ?$$

e isso não seria possível, pelo menos não diretamente. Então como podemos resolver isso? Bom há duas maneiras, a primeira delas seria transpondo o segundo array, assim as colunas virariam linhas e as linhas se tornariam colunas ou vice-versa:

$$V = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \rightarrow V^T = \begin{bmatrix} a & b & c \end{bmatrix}$$

Nesse caso o `arr_2` ficaria assim:

$$\begin{bmatrix} 0.2 & 0.8 & -0.5 & 1.0 \\ 0.5 & -0.91 & 0.26 & -0.5 \\ -0.26 & -0.27 & 0.17 & 0.87 \end{bmatrix} \longrightarrow \begin{bmatrix} 0.2 & 0.5 & -0.26 \\ 0.8 & -0.91 & -0.27 \\ -0.5 & 0.26 & 0.17 \\ 1.0 & -0.5 & 0.87 \end{bmatrix}$$

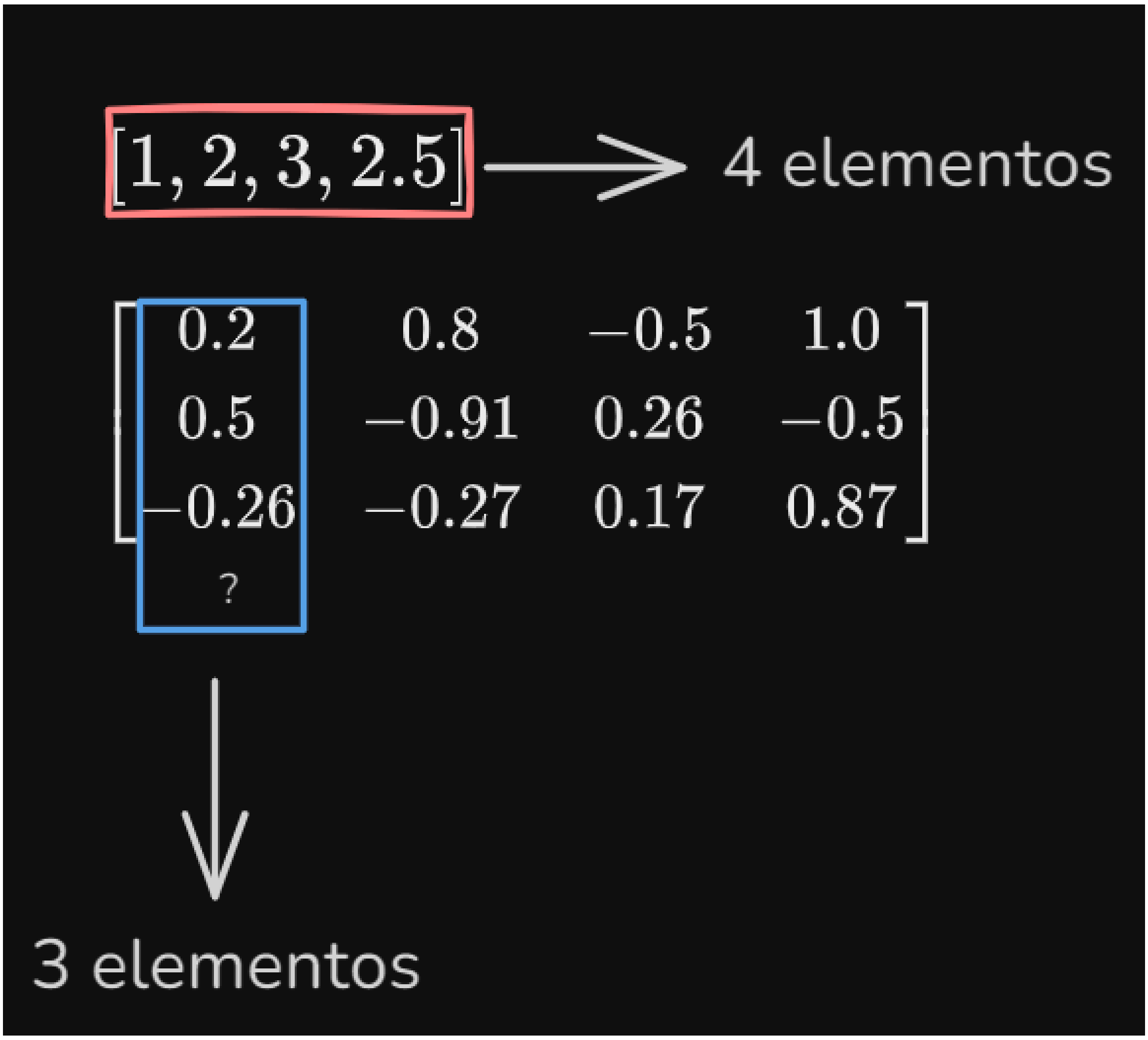
$$\begin{bmatrix} 0.2 & 0.5 & -0.26 \\ 0.8 & -0.91 & -0.27 \\ -0.5 & 0.26 & 0.17 \\ 1.0 & -0.5 & 0.87 \end{bmatrix}$$

```
import numpy as np

arr_2 = np.array([[0.2, 0.8, -0.5, 1.0],
                  [0.5, -0.91, 0.26, -0.5],
                  [-0.26, -0.27, 0.17, 0.87]])

arr_2 = arr_2.T # Transposição do array
```

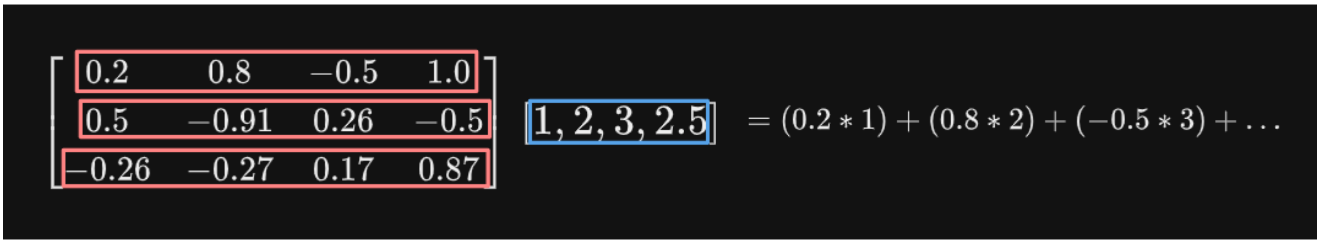
Com essa transposição, teríamos a quantidade de linhas igual a quantidade de elementos no `arr_1`, obedecendo as regras do produto escalar e realizando a operação diretamente.

$$[1, 2, 3, 2.5] \begin{bmatrix} 0.2 & 0.5 & -0.26 \\ 0.8 & -0.91 & -0.27 \\ -0.5 & 0.26 & 0.17 \\ 1.0 & -0.5 & 0.87 \end{bmatrix} = (1 * 0.2) + (2 * 0.8) + (3 * (-0.5)) + \ldots$$

Já a segunda maneira de fazermos isso seria invertendo a ordem dos arrays na operação, exemplo, vimos que se realizarmos a operação somando cada item do `arr_1` com cada linha da primeira coluna do `arr_2`:

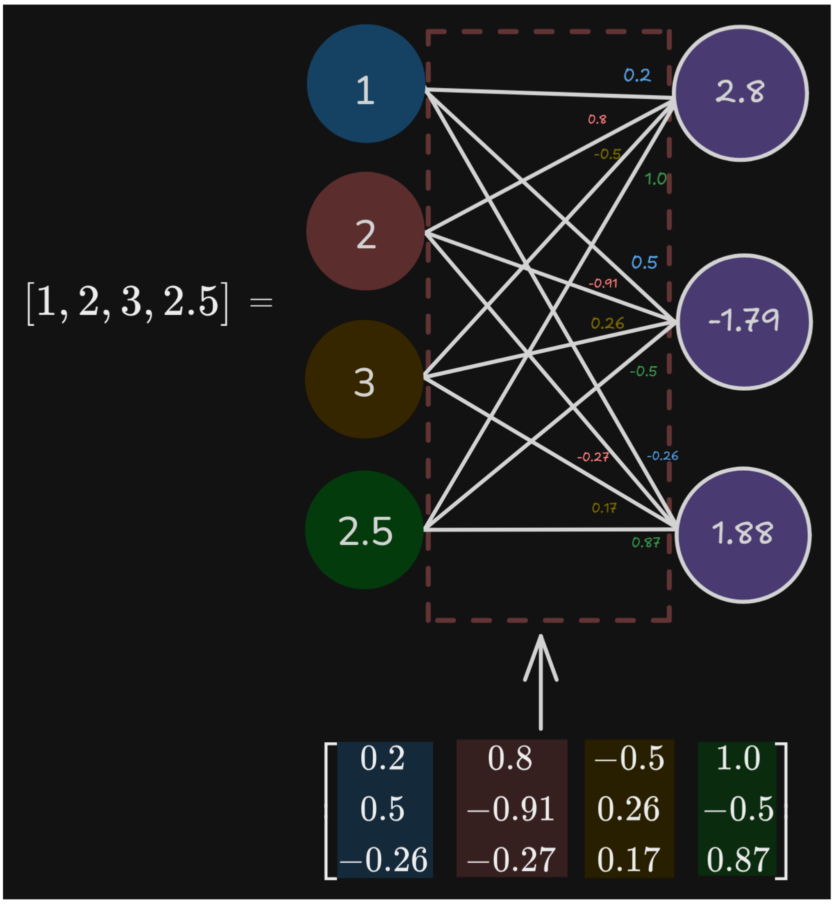


Isso não seria possível por conta da quantidade de linhas a primeira coluna do `arr_2` tem, porém e se realizassemos a operação apenas invertendo os arrays? Por exemplo:

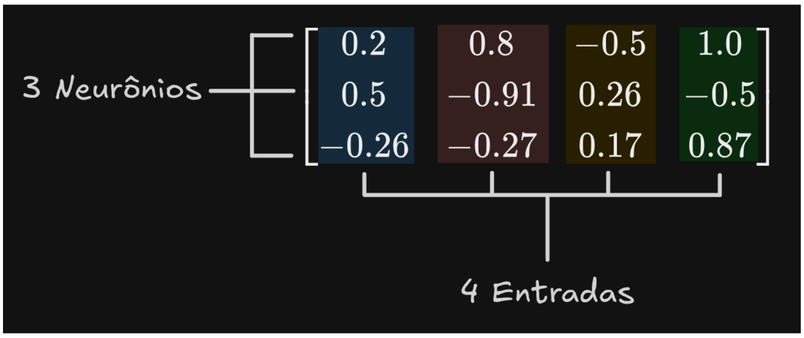


Outra coisa que pode nos ajudar a pensar em que maneira devemos organizar os arrays para fazer esse produto escalar seria o seguinte:

Sabemos que, cada valor de entrada na nossa rede neural, precisa passar por alguns processos antes de se tornar algum valor para a saída dessa rede. Mas o que eu quero dizer com isso? Bom, estamos criando um modelo de aprendizado de máquina, então iremos forneces alguns dados, valores, entradas para que a máquina encontre padrões e nos retorne outros valores, certo? Então, esses valores de saída, antes mesmo de se tornarem os valores finais dessa rede, eles passam pro alguns processos, equações, etc, mas onde esses processos e equações são feitos, dentro de outro neurônio:



Ou seja, cada linha dos pesos representam a quantidade dos próximos neurônios, para cada peso, um neurônio. Cada entrada em qualquer neurônio de qualquer camada irá ter a quantidade de pesos igual a quantidade de neurônios que irão recebê-la.



---

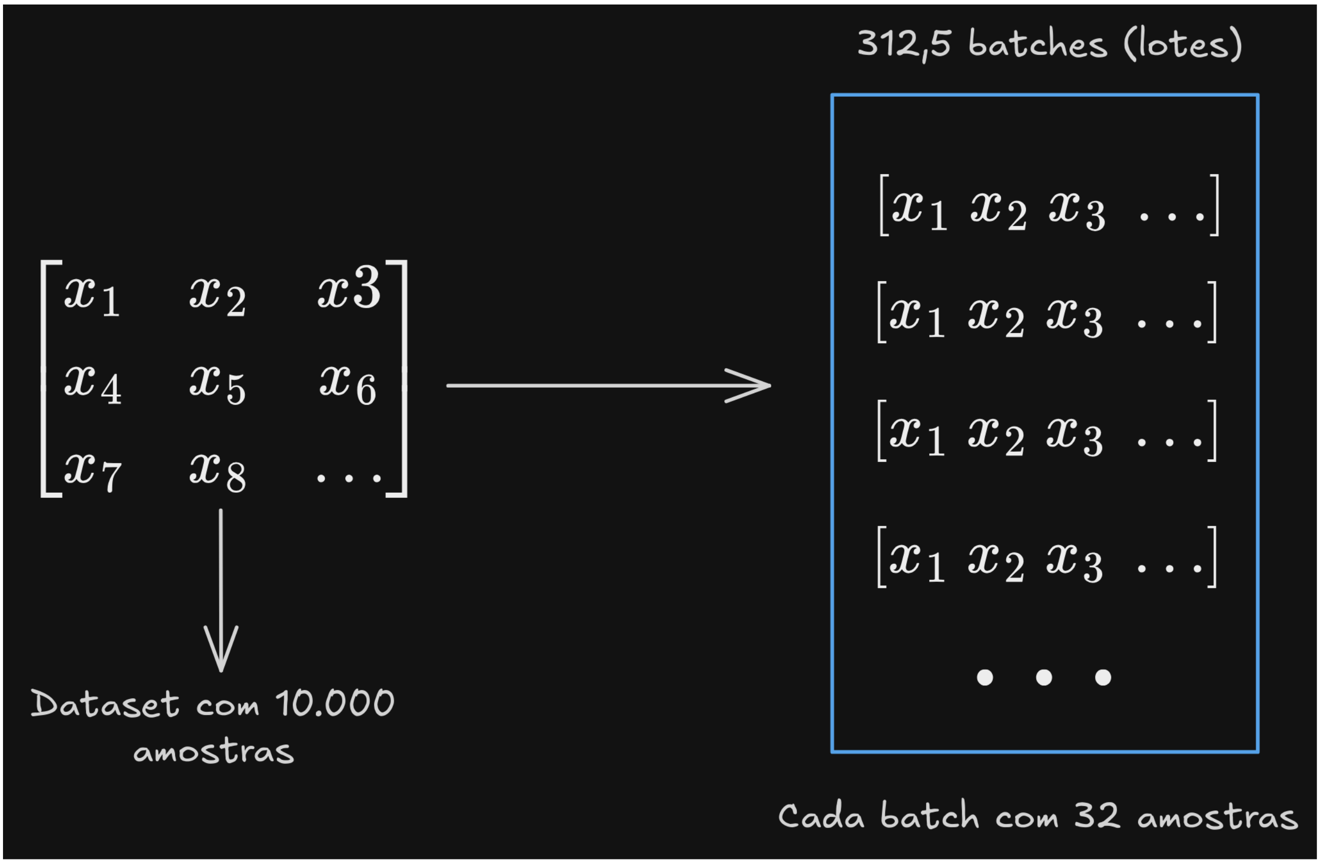
9. Batches

Batches (ou **lotes**, em português) são um conceito fundamental no treinamento de modelos de machine learning, especialmente em redes neurais. Eles se referem à divisão do conjunto de dados em pequenos grupos (lotes) que são processados separadamente durante o treinamento. Vamos explorar o que são batches, por que são usados e como funcionam.

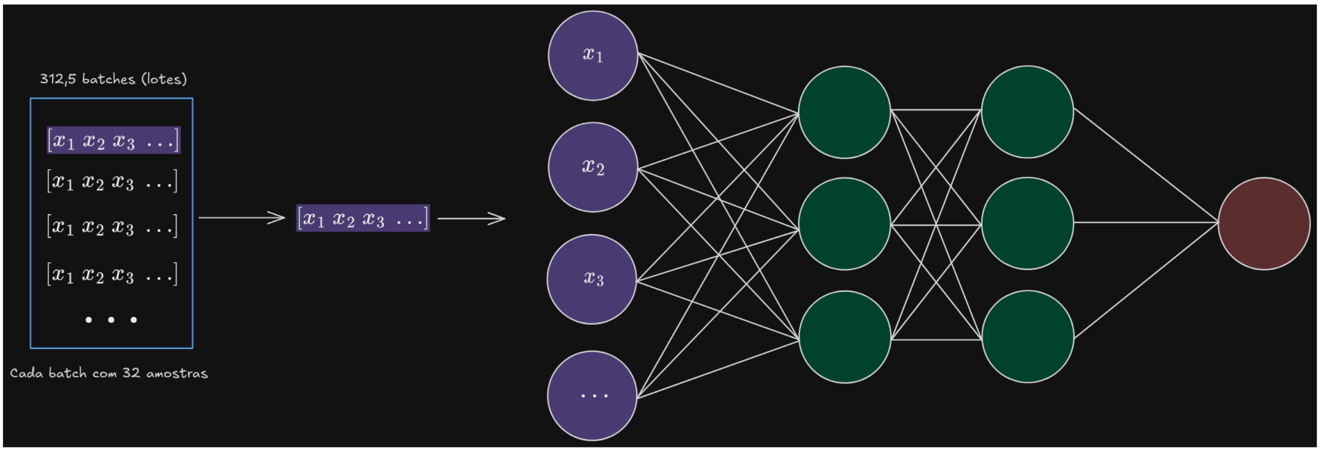
- Um **batch** é um subconjunto dos dados de treinamento.
- Em vez de processar todos os dados de uma vez (o que é chamado de **treinamento em lote completo** ou *batch training*), os dados são divididos em lotes menores.
- Cada batch é passado pelo modelo para calcular o erro (ou perda) e atualizar os parâmetros do modelo.

Exemplo:

Vamos supor que tenhamos um dataset com 10.000 features (dados de entrada, amostras, etc), para que nosso modelo não treine as 10.000 amostras de uma só vez o que poderia causar overfitting e alto custo de hardware, nós separamos essas amostras em lotes menores, ou, batches:



Depois de realizar o processo de separação de amostras, cada batch desse conjunto será utilizado para treinar o modelo e atualizar os valores, ou seja, cada batch é passado pela rede neural para calcular as predições, depois, compara-se as predições com os valores reais para computar uma função de perda (loss), então, o erro é propagado de volta pela rede para calcular os gradientes dos pesos, por fim, os parâmetros da rede (pesos e biases) são ajustados utilizando os gradientes, geralmente com algum algoritmo de otimização como SGD, Adam, etc.



Esse processo é repetido para cada batch dentro de uma época (um ciclo completo por todo o conjunto de dados). Após completar uma época, o conjunto de dados pode ser embaralhado e o processo recomeça para a próxima época até que o modelo atinja a performance desejada.

Portanto, os batches são utilizados para treinar o modelo em partes, passando cada um deles pela rede neural para calcular o erro e ajustar os pesos conforme necessário, repetindo esse ciclo durante o treinamento.

Utilizar batches durante o treinamento ajuda o modelo a identificar e aprender os padrões subjacentes nos dados, em vez de simplesmente memorizá-los. Essa abordagem favorece a generalização, permitindo que o modelo compreenda os dados de forma mais robusta e seja mais eficaz ao lidar com exemplos novos e variados.

Código

```python
import numpy as np

# Batch size: 3
# Entradas por batch: 4

inputs = [[1, 2, 3, 2.5], # Amostra 1
          [2.0, 5.0, -1.0, 2.0], # Amostra 2
          [-1.5, 2.7, 3.3, -0.8]] # Amostra 3

weights = [[0.2, 0.8, -0.5, 1.0],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]

biases = [2, 3, 0.5]

output = np.dot(inputs, np.array(weights).T) + biases
```

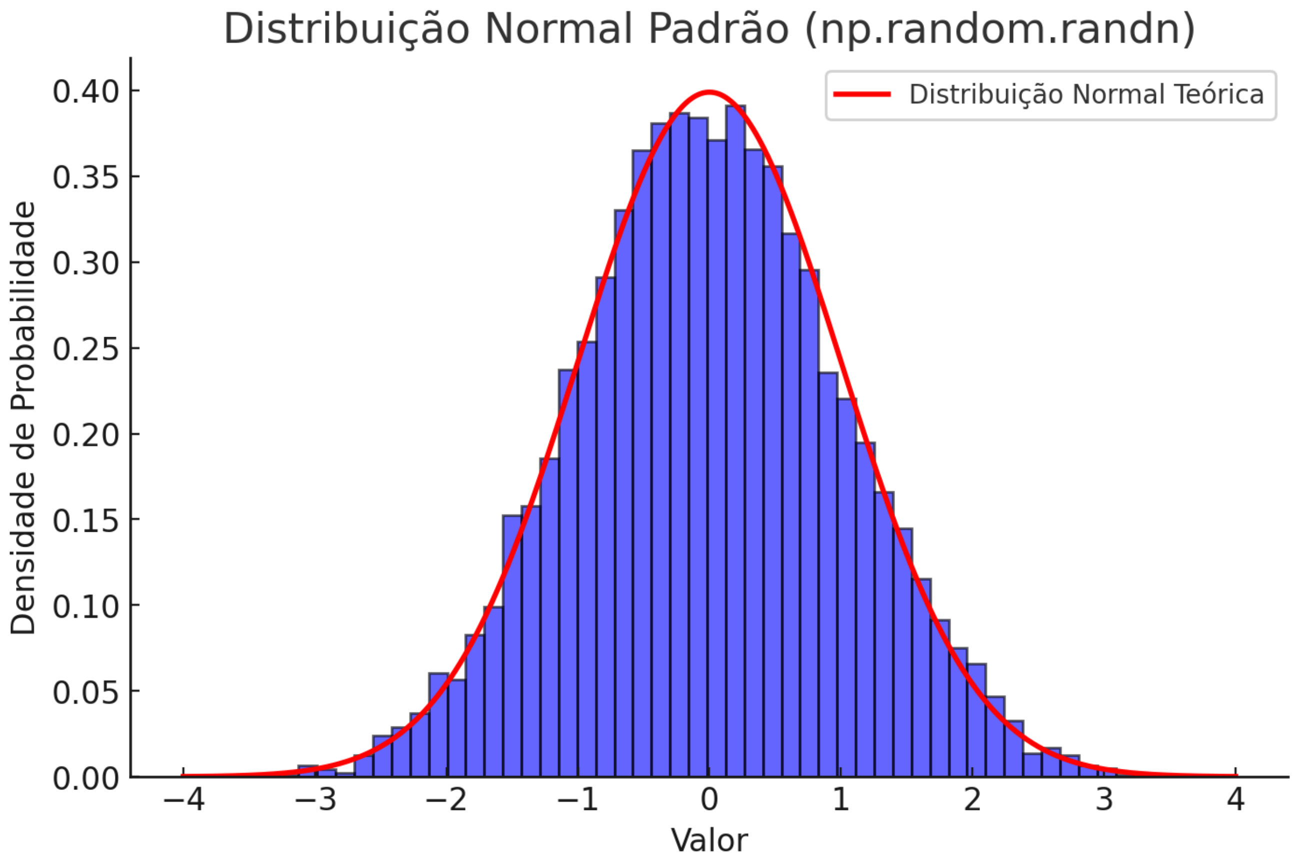
10. Camada densa (objeto)

```
import numpy as np

np.random.seed(0)

# Targets (X)
X = [[1, 2, 3, 2.5],
    [2.0, 5.0, -1.0, 2.0],
    [-1.5, 2.7, 3.3, -0.8]]

class Layer_Dense:
    # Método construtor
    def __init__(self, n_inputs, n_neurons):
        self.weight = 0.10 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))
```

Explicação

`np.random.seed(0)` -> Define a semente de geração dos números, isso permite que os número sejam reproduzidos posteriormente.

`self.weight = 0.10 * np.random.randn(n_inputs, n_neurons)` -> Basicamente o que essa linha faz é criar uma variável para armazenar os pesos que serão gerados a partir da função `np.random.randn` que basicamente gera números aleatórios seguindo a distribuição normal padrão com média de 0 e desvio padrão de 1.



O que isso significa?

- A maioria dos valores gerados estará **próxima de 0**.
- Mas **não há limites fixos**: os números podem ser **negativos ou positivos**, indo de −∞ a +∞.
- No entanto, **68% dos valores** estarão no intervalo -1, 1 [−1,1] (1 desvio padrão da média).
- **95% dos valores** estarão no intervalo −2,2 [−2,2].
- **99.7% dos valores** estarão no intervalo −3,3 [−3,3].

A multiplicação por `0.10` serve para **reduzir a escala** dos valores gerados por `np.random.randn(n_inputs, n_neurons)`.

`self.biases = np.zeros((1, n_neurons))` -> Aqui basicamente, estamos criando um vetor preenchido de zeros de acordo com a quantidade de neurônios, ou seja, inicialmente, cada neurônio irá receber um viés de 0.

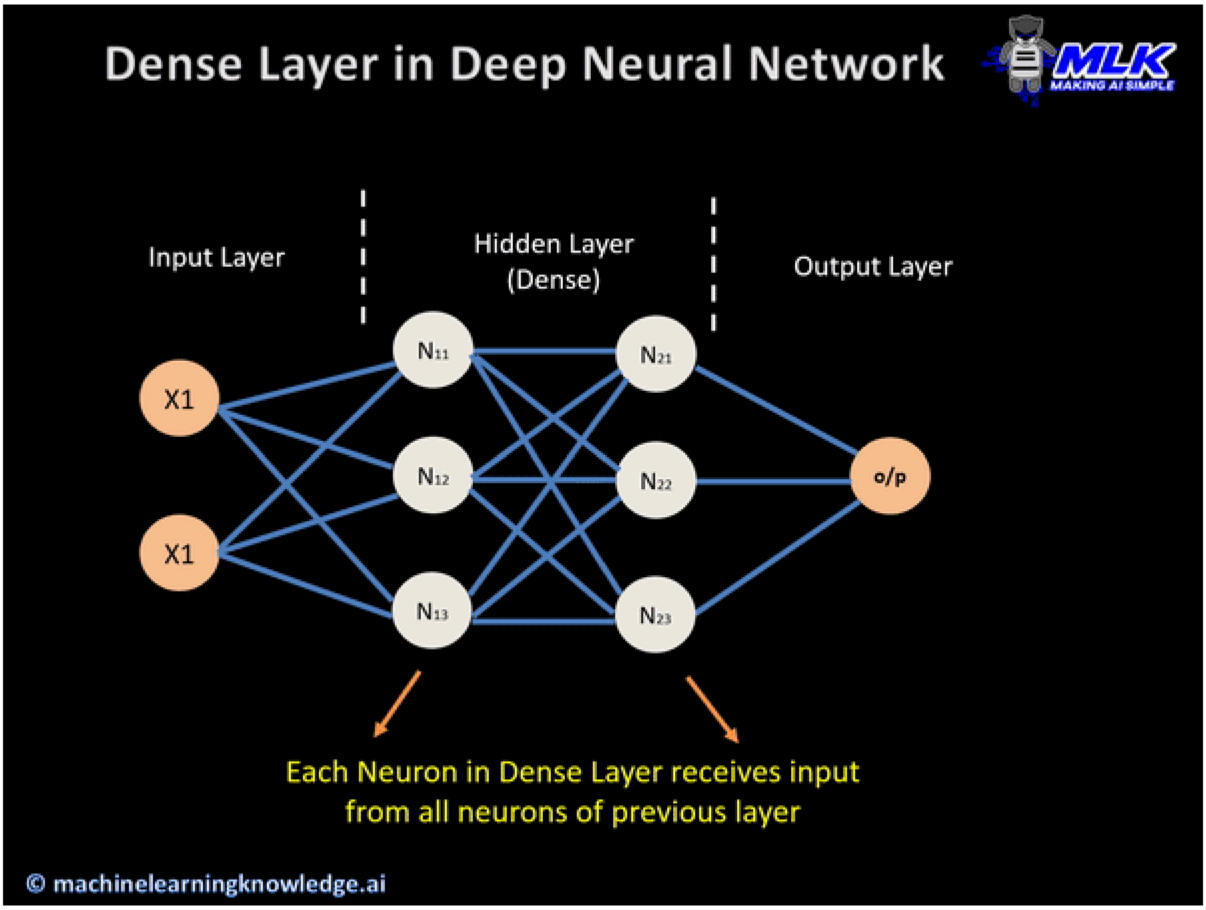
Agora iremos criar nossa `forward method`, basicamente o método do nosso objeto que irá inicializar as entradas, fazer a operação de um neurônio, etc:

```
def forward(self, inputs):
    self.output = np.dot(inputs, self.weights) + self.biases
```

Esse código pode ser interpretado aritmeticamente por essa fórmula:

$$\sum_{i=1}^{n}(x_i * w_i) + b$$

Ou seja, ela processa a saída de uma camada densa.



Em uma rede neural simples, pode-se ter uma **camada de entrada** (onde os dados entram), uma ou mais **camadas densas ocultas** (onde o aprendizado ocorre) e uma **camada de saída** (onde a previsão é gerada). A camada densa é frequentemente usada nas camadas ocultas e de saída, pois sua estrutura totalmente conectada permite que a rede aprenda uma ampla variedade de padrões e complexidades.

Por exemplo, em uma rede neural para classificação, a camada densa pode ser responsável por aprender representações complexas dos dados de entrada, como uma imagem ou uma sequência de texto, para que a rede possa fazer previsões precisas sobre essas entradas.

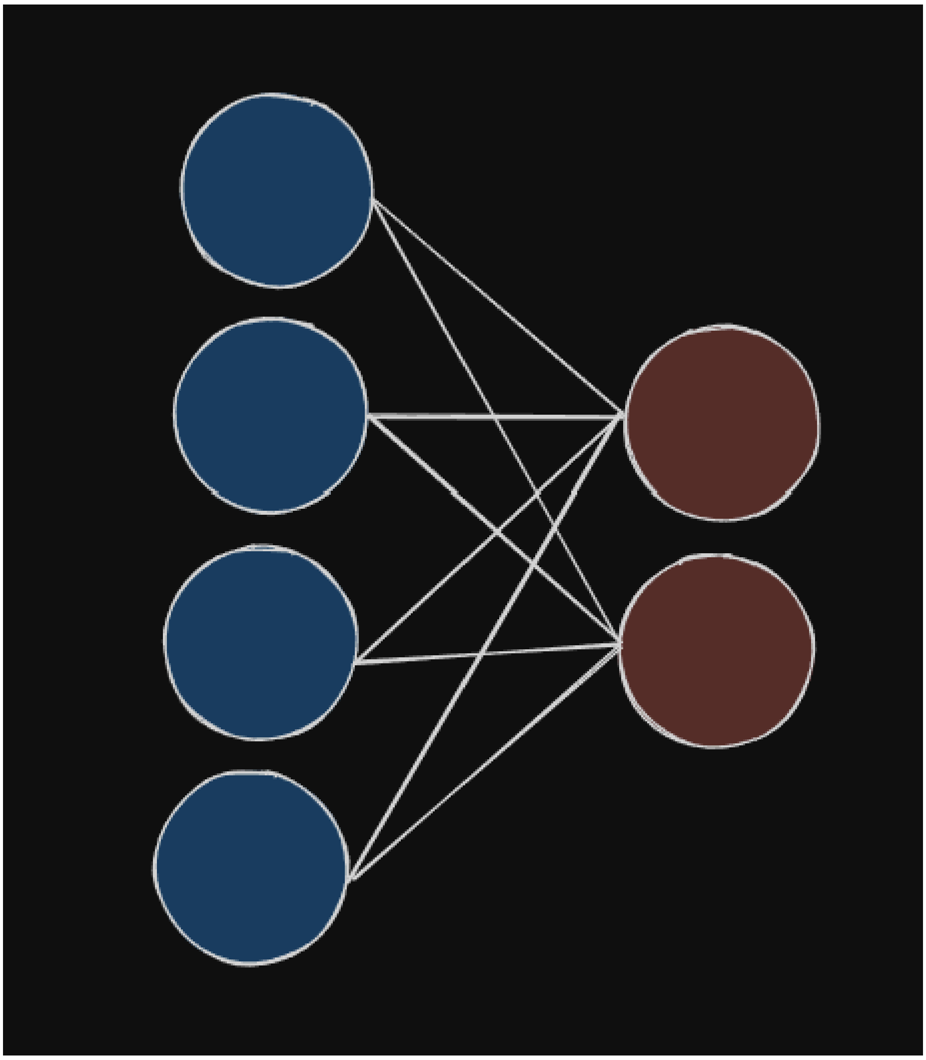
Código final:

```
import numpy as np

np.random.seed(0)
```

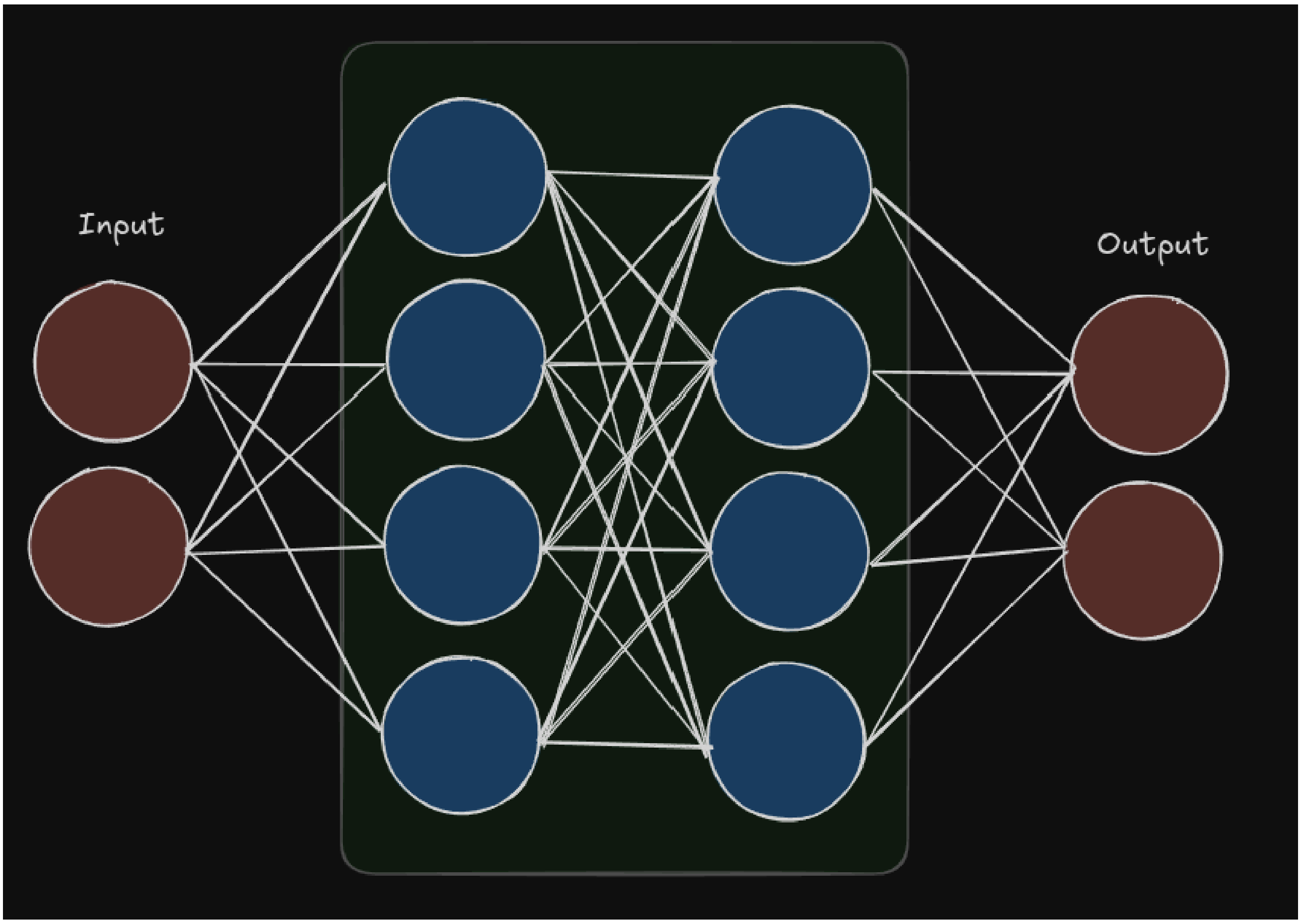
```
# Dados de entrada X
X = [[1, 2, 3, 2.5],
    [2.0, 5.0, -1.0, 2.0],
    [-1.5, 2.7, 3.3, -0.8]]


class Layer_Dense:
    def __init__(self, n_inputs, n_neurons):
        self.weight = 0.10 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))

    # Calcula a saída da camada
    def forward(self, inputs):
        self.output = np.dot(inputs, self.weight) + self.biases


# Definindo a estrutura da rede
Layer1 = Layer_Dense(4, 5) # 4 entradas para 5 neurônios
Layer2 = Layer_Dense(5, 2) # 5 entradas para 2 neurônios


Layer1.forward(X) # inicializa a camada 1 com os valores de X
Layer2.forward(Layer1.output) # inicializa a camada 2 com os valores
de saída da camada 1

print(Layer2.output) # Imprime no terminal os valores de saída da
camada 2
```

11. Camada densa; Camada oculta e Camadas esparsas

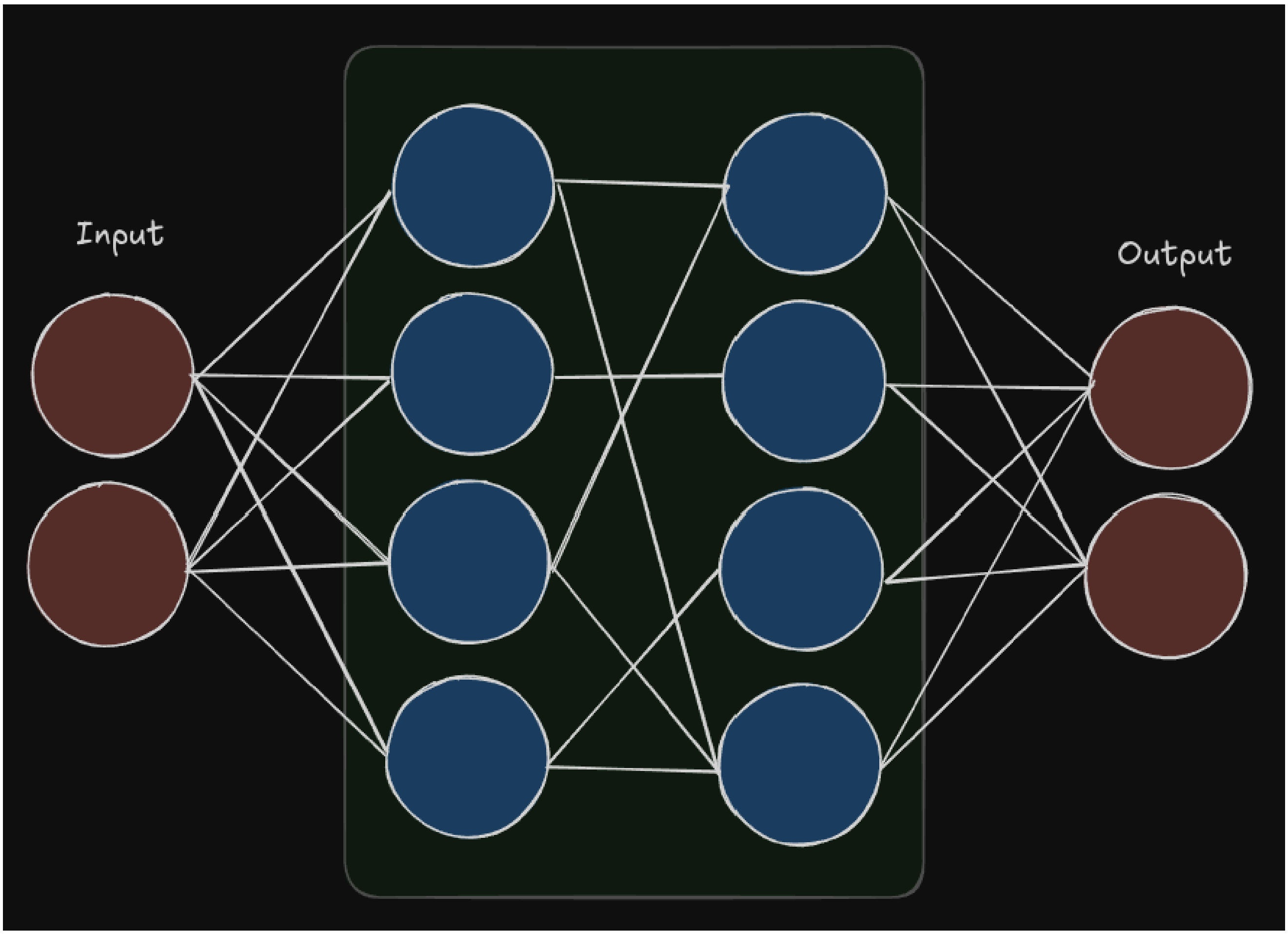
Uma camada densa é um tipo de camada onde cada neurônio está conectado à todos os outros neurônios da camada anterior.



Já a camada oculta refere-se à todos os neurônios que estão entre a camada de entrada e saída da rede neural.



Um **neurônio** não precisa, necessariamente, estar conectado a todos os neurônios da camada anterior ou da camada posterior. Em uma **camada esparsa** (ou **sparse layer**), a maioria das conexões entre os neurônios é removida ou restringida, de modo que apenas algumas conexões específicas são mantidas. Ou seja, uma camada esparsa, é justamente o contrário de uma camada densa (dense layer).



---

12. Função de ativação

Uma função de ativação é uma função matemática usada em redes neurais para introduzir não linearidade às saídas de cada neurônio. Ela determina se o neurônio deve ou não ser ativado em base a soma ponderada das entradas. Cada neurônio da camada oculta irá ter, individualmente sua função de ativação acoplada.

Principais tipos de função de ativação:

- Step function
- Sigmoide
- Tangente hiperbólica
- ReLU
- Softmax

12.1. Implementação ReLU

A função **ReLU (Rectified Linear Unit)** é uma função de ativação amplamente utilizada em redes neurais artificiais, especialmente em **redes neurais profundas (DNNs)** e **redes convolucionais (CNNs)**. Ela é definida matematicamente como:

$$f(x) = max(0, x)$$

Ou seja, se a entrada x for positiva, a saída será x; caso contrário, a saída será 0.

Vantagens do ReLU

1. **Evita o problema do gradiente desaparecendo**
   - Diferente de funções como **sigmoide** e **tangente hiperbólica (tanh)**, que podem ter gradientes muito pequenos para valores extremos, o ReLU mantém gradientes fortes para entradas positivas, acelerando o treinamento.
2. **Computacionalmente eficiente**
   - A função é simples e rápida de calcular, pois apenas compara o valor de entrada com zero.
3. **Introduz esparsidade**
   - Como os valores negativos são transformados em zero, a rede pode aprender de forma mais eficiente ao ativar apenas algumas unidades.

Implementação na prática

```
"""
Funções de ativação da camada oculta (ReLU)
"""

import numpy as np

# nnfs (Neural Network From Scratch)
import nnfs
from nnfs.datasets import spiral_data

"""
Inicializando a biblioteca nnfs
-Define uma semente fixa para o gerador de números aleatórios
(numpy.random.seed)
- Configura o dtype padrão do numpy para float32
- etc...
"""
nnfs.init()

np.random.seed(0)

# Dados de entrada X
X = [[1, 2, 3, 2.5],
    [2.0, 5.0, -1.0, 2.0],
    [-1.5, 2.7, 3.3, -0.8]]
```

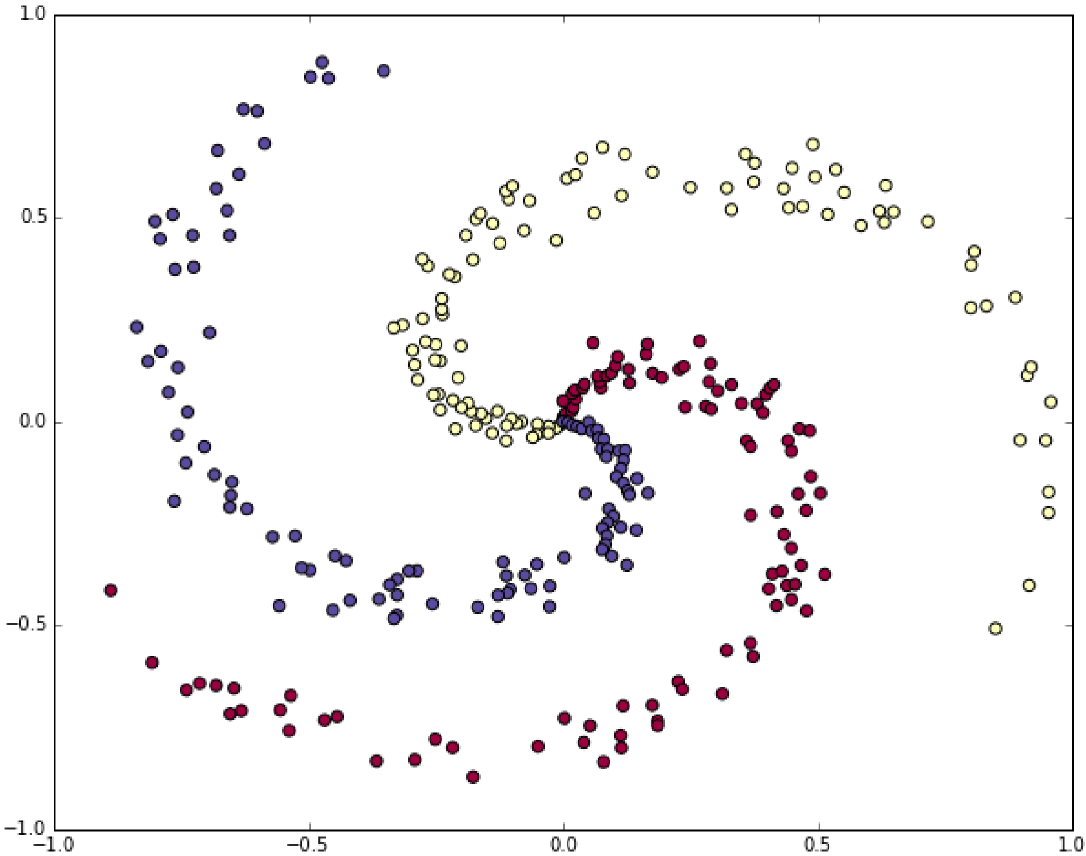
```
# Gera dados em espiral. 100 (número de amostras), 3 (Números de
classes)
X, y = spiral_data(100, 3)

class Layer_Dense:
    def __init__(self, n_inputs, n_neurons):
        self.weight = 0.10 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))

    # Calcúla a saída da camada
    def forward(self, inputs):
        self.output = np.dot(np.array(inputs), self.weight) +
self.biases

class Activation_ReLU:
    def forward(self, inputs):
        """
        Retorna o maior valor entre 0 e cada iterável de "inputs"
(entradas)
        """
        self.output = np.maximum(0, inputs)

# Configurando a estrutura da rede
Layer1 = Layer_Dense(2, 5) # Inicializa a camada 1 com 2 entradas
para 5 neurônios
activation1 = Activation_ReLU()
Layer1.forward(X) # Usando dados de entrada X para obter a saída
activation1.forward(Layer1.output) # Usando a função de ativação
```

Explicação:

`nnfs` -> nnfs (Neural Network From Scratch) é um pequeno módulo em python criado pelo youtuber e desenvolvedor sentdex para acompanhar o seu livro *Neural Networks from Scratch* e facilitar a implementação das redes neurais do zero.

`nnfs.init()` -> Inicializa o módulo e configura o numpy para o melhor uso com redes neurais. Essa função define uma semente fixa para o gerador de números aleatórios, define o valor padrão do numpy dtype para float32, e mais alguns ajustes.

`spiral_data(100, 3)` -> Gera dados em espiral com 100 amostras e 3 classes e os armazena nas variáveis X e y (features e labels)



No caso desta imagem há mais do que 100 amostras e 4 classes

Mas, por que gerar dados em espiral? Gerar **dados em espiral** (`spiral_data()`) é útil porque cria um **problema não linear** que não pode ser resolvido apenas com um **perceptron simples** ou modelos lineares. Isso força a rede neural a aprender representações mais complexas.

`np.maximum(0, inputs)` :

```
class Activation_ReLU():
    def forward(self, inputs):
        self.output = np.maximum(0, inputs)
```

Criando classe da função de ativação (ReLU) com o método forward que servirá para extrair o maior valor entre 0 e cada elemento iterável do array de `inputs`, ou seja, se o valor for maior que 0, esse mesmo valor será retornado e armazenado em `self.output`, caso o valor for menor que 0, 0 será o maior valor, ou seja, irá armazenar 0 em `self.output`. Basicamente representa a definição matemática:

$$f(x) = \begin{cases} 0, & \text{se } x \leq 0 \\ x, & \text{se } x > 0 \end{cases}$$

Por fim, a camada 1 é inicializada com 2 entradas para 5 neurônios, o objeto da classe da função de ativação é iniciada, logo após, a saída da camada 1 é calculada com os valores de X, e então, a função de ativação é acionada a partir da saída da camada 1:

```
Layer1 = Layer_Dense(2, 5)
activation1 = Activation_ReLU()
Layer1.forward(X)
activation1.forward(Layer1.output)
```

12.2. Implementação Softmax

A função **Softmax** é uma função de ativação frequentemente utilizada em problemas de classificação multiclasse, particularmente em redes neurais. Ela transforma um vetor de valores **reais** em um vetor de **probabilidades** no intervalo de `0` a `1`, de modo que a soma dessas probabilidades seja igual a `1`.

Formula

Seja $z = [z_1, z_2, \ldots, z_n]$ o vetor de entrada, onde cada z_i é um valor real, a função Softmax é definida como:

$$Softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

- Onde e^{z_i} é a exponencial de z_i
- $\sum_{j=1}^{n} e^{z_j}$ é a soma das exponenciais de todos os elementos do vetor z

O resultado da aplicação do Softmax é um vetor $p = [p_1, p_2, \ldots, p_n]$ onde cada p_i é uma probabilidade que representa a chance de cada classe para o modelo, com a condição de que:

$$\sum_{i=1}^{n} p_i = 1$$

Suponha que a rede neural forneça os seguintes valores de saída para 3 classes: $[3, 1, 0.2]$. Aplicando o Softmax, você obteria a probabilidade de cada classe.

Implementação na prática

Implementação básica:

```python
import numpy as np
import nnfs

nnfs.init()

layer_outputs = [4.8, 1.21, 2.385]

exp_values = np.exp(layer_outputs)
norm_values = exp_values / np.sum(exp_values)

print(norm_values) # [0.89528266 0.02470831 0.08000903]
print(sum(norm_values)) # 0.9999999999999999
```

Basicamente o que fizemos foi pegar os dados de entrada, exponenciar esses dados, normalizá-los e por fim obter os dados de saída:

Vamos supor que temos os dados de entrada sendo "cachorro", "gato" e "humano", sendo $[1, 2, 3]$ respectivamente, iremos exponenciar cada dado de entrada: $[e^1, e^2, e^3]$, então normalizaremos os dados dessa saída:

$$\begin{bmatrix} \frac{e^1}{e^1+e^2+e^3} \\ \frac{e^2}{e^1+e^2+e^3} \\ \frac{e3}{e^1+e^2+e^3} \end{bmatrix}$$

E, por fim, obter os dados de saída: $[0.09, 0.24, 0.67]$

O processo de exponenciação e normalização juntos é o que a função de ativação softmax faz, ele engloba essas duas operações em uma só:

$$S_{i,j} = \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,j}}}$$

$S_{i,j}$ - Saída da função softmax

$e^{z_{i,j}}$ - É a exponencial do valor $z_{i,j}$

$\sum_{l=1}^{L} e^{z_{i,j}}$ - É a soma das exponenciais de todas as pontuações $z_{i,j}$ para a amostra i, considerando todas as classes l do conjunto de dados

$z_{i,j}$ - Representa o valor bruto (pontuação não normalizada) associado à entrada j para a amostra i

L - Representa o número total de classes ou categorias possíveis no modelo

Implementação avançada:

```
import numpy as np
import nnfs

nnfs.init()

layer_outputs = [[4.8, 1.21, 2.385],
                [8.9, -1.81, 0.2],
                [1.41, 1.051, 0.026]]

exp_values = np.exp(layer_outputs)
norm_values = exp_values / np.sum(exp_values, axis=1, keepdims=True)

print(norm_values)

"""
[[8.95282664e-01 2.47083068e-02 8.00090293e-02]
 [9.99811129e-01 2.23163963e-05 1.66554348e-04]
 [5.13097164e-01 3.58333899e-01 1.28568936e-01]]
"""
```

`np.exp()` - Essa função retorna a exponencial ($f(x) = e^x$) de algum valor

`axis` - define **ao longo de qual eixo** a soma será realizada em um array multidimensional

Axis	Explicação
None	A soma é feita sobre todos os elementos do array, retornando um único número.
0	Soma os valores das colunas
1	Soma os valores das linhas

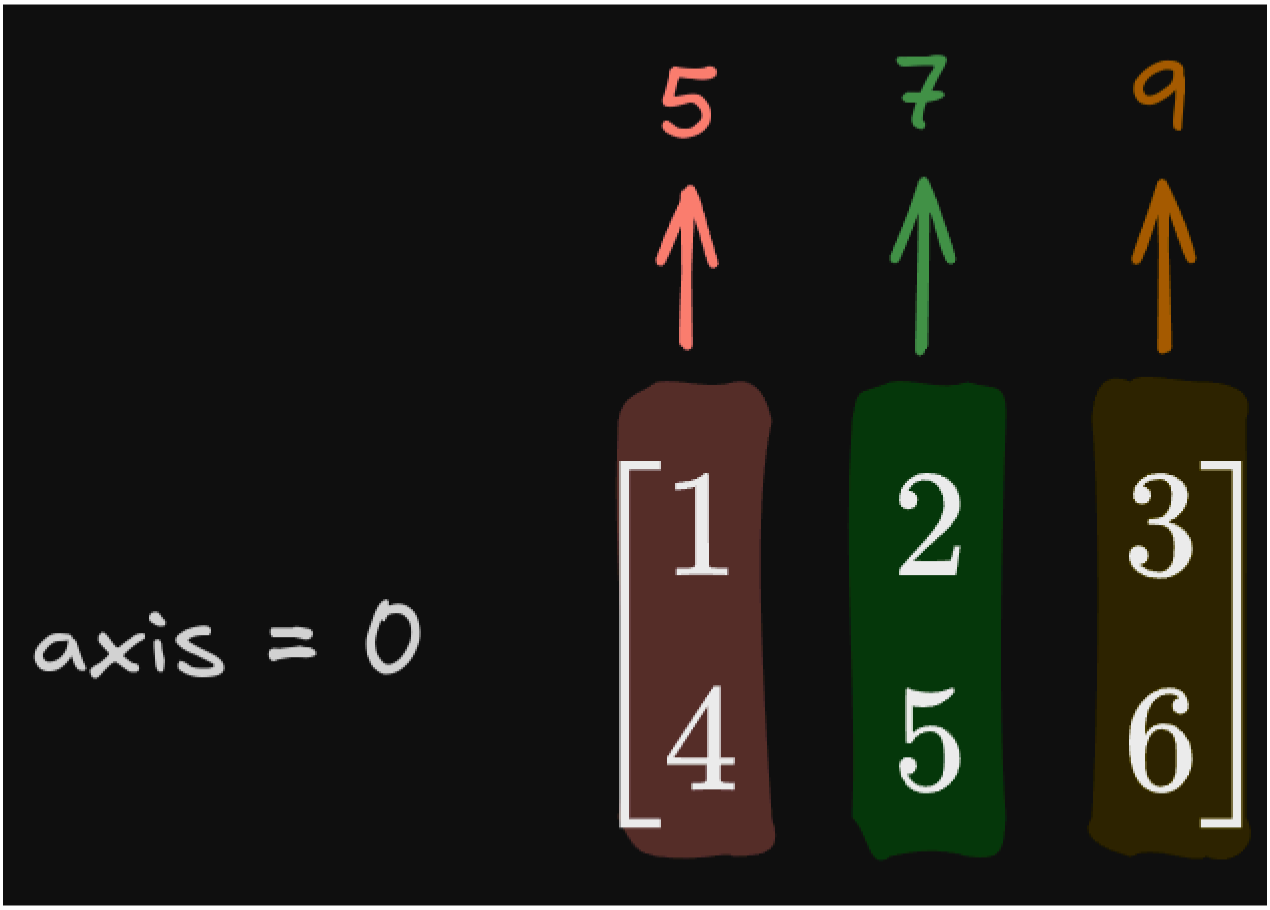
Exemplos:

`axis = 0`

```
import numpy as np

arr = np.array([[1, 2, 3],
                [4, 5, 6]])
```

```
soma_total = np.sum(arr, axis=0)
print(soma_total)
```

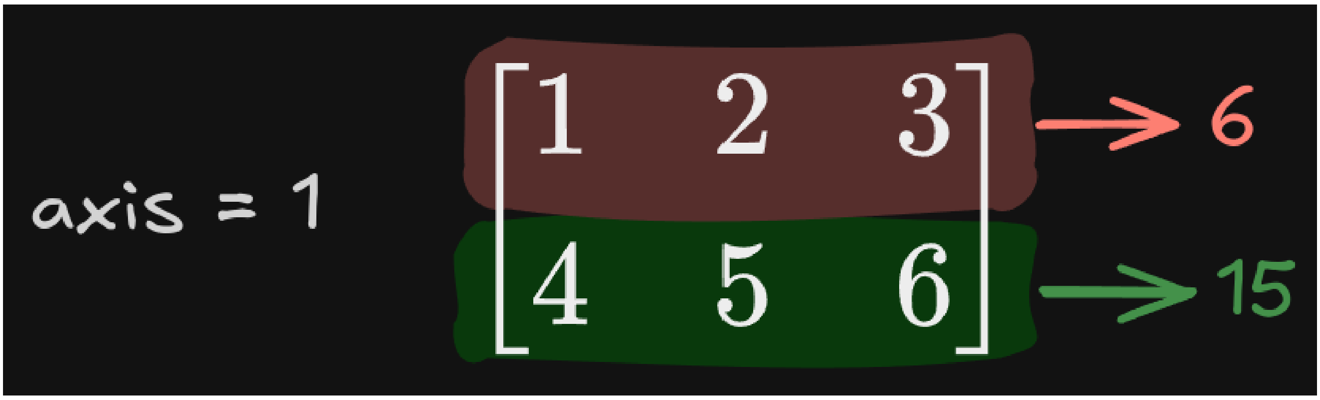
```
[5 7 9]
```



`axis = 1`

```
import numpy as np

arr = np.array([[1, 2, 3],
                [4, 5, 6]])

soma_total = np.sum(arr, axis=1)
print(soma_total)
```

```
[6 15]
```



`keepdims` - define se o resultado da soma **mantém ou reduz** a dimensionalidade original do array:

keepdims	Explicação
False (padrão)	Reduz a dimensão do array após a soma.
True	Mantém a dimensão original, preenchendo com 1 onde ocorreu a soma.

`keepdims = False`

```
import numpy as np

arr = np.array([[1, 2, 3],
                [4, 5, 6]])

soma_total = np.sum(arr)
print(soma_total)
```

```
21
```

`keepdims = True`

```
import numpy as np

arr = np.array([[1, 2, 3],
                [4, 5, 6]])

soma_total = np.sum(arr)
print(soma_total)
```

```
[[21]]
```

Exemplo com os dois parâmetros:

`axis = 1` & `keepdims = True`

```
import numpy as np

arr = np.array([[1, 2, 3],
                [4, 5, 6]])

soma_total = np.sum(arr, axis=1, keepdims=True)
print(soma_total)
```

```
[[ 6]
 [15]]
```

Classe da softmax:

```
class Activation_softmax:
    def forward(self, inputs):
        exp_values = np.exp(inputs - np.max(inputs,
                                            axis=1,
                                            keepdims=True))

        nom_values = exp_values / np.sum(exp_values,
                                        axis=1,
                                        keepdims=True)# Normaliza a
saída exponencial
```

12.2.1. Estabilidade numérica

É importante entender também que, no processo de exponenciação, é recomendável aplicar uma técnica chamada **estabilidade numérica** para evitar problemas com valores extremamente grandes.

A estabilidade numérica ($v_i = u_i - max(u)$) é um calculo que o softmax realiza para poder evitar o overflow dos dados de saída. **Overflow** é um problema numérico que ocorre quando um número fica **maior do que o sistema consegue representar**, resultando em erros ou valores infinitos, isso pode ocorrer quando as entradas são representadas por números muito grandes.

Esse conceito aplicado à exponencial resulta nesta fórmula:

$$e^{v_i} = e^{u_i - max(u)}$$

A função de ativação **Softmax** atua na camada de saída da rede neural. Como as redes neurais geralmente processam múltiplas amostras ao mesmo tempo para melhorar a eficiência do treinamento, o que recebemos na entrada dessa camada é um **batch** de dados, ou seja, um conjunto de amostras processadas simultaneamente. No código abaixo:

```
exp_values = np.exp(inputs - np.max(inputs, axis=1, keepdims=True))
```

estamos lidando com esse batch de dados. Se aplicarmos `np.max(inputs)` sem especificar o eixo, a função retornará o maior valor de todo o batch inteiro, em vez do maior valor de cada amostra individualmente, o que comprometeria a estabilidade numérica da Softmax. Para resolver esse problema, utilizamos os parâmetros `axis=1` e `keepdims=True`, garantindo que o máximo seja calculado para cada amostra separadamente e que o formato do array seja preservado, permitindo a subtração correta. Dessa forma, asseguramos que a normalização da Softmax seja realizada adequadamente para cada conjunto de dados dentro do batch.

Implementação final:

```
"""
Funções de ativação da camada oculta (softmax)
"""

import numpy as np
# nnfs (Neural Network From Scratch)
import nnfs
from nnfs.datasets import spiral_data

"""
Inicializando a biblioteca nnfs
-Define uma semente fixa para o gerador de números aleatórios
(numpy.random.seed)
- Configura o dtype padrão do numpy para float32
- etc...
"""
nnfs.init()

class Layer_Dense:
    def __init__(self, n_inputs, n_neurons):
        self.weight = 0.10 * np.random.randn(n_inputs, n_neurons)
```

```
        self.biases = np.zeros((1, n_neurons))

    # Calcula a camada de saída
    def forward(self, inputs):
        self.output = np.dot(np.array(inputs), self.weight) +
self.biases


class Activation_ReLU:
    def forward(self, inputs):
        self.output = np.maximum(0, inputs) # Retorna o maior valor
entre 0 e cada iterável de "inputs" (entradas)

class Activation_softmax:
    def forward(self, inputs):
        exp_values = np.exp(inputs - np.max(inputs, axis=1,
keepdims=True))
        probabilities = exp_values / np.sum(exp_values, axis=1,
keepdims=True)# Normaliza a saída exponencial
        self.output = probabilities

# Criando os dados em espiral com 100 amostras para 3 classes (33
amostras para cada classe)
X, y = spiral_data(samples=100, classes=3)

"""Configurando as camadas"""

# Criando a primeira camada densa com 2 entradas e 3 neurônios
dense1 = Layer_Dense(2, 3)
# Incializando a função de ativação #ReLU
activation1 = Activation_ReLU()

# Criando a segunda camada densa com 3 entradas para 3 neurônios
dense2 = Layer_Dense(3, 3)
# Inicializando a função de ativação softmax
activation2 = Activation_softmax()

"""Fornecendo as entradas"""

# Fornecendo os dados do spiral_data para a primeira camada densa
dense1.forward(X)
# Ativando a função ReLU com a saída da primeira camada
activation1.forward(dense1.output)

# Obtendo a saída da função ReLU e usando como a entrada da camada
densa 2
dense2.forward(activation1.output)
```

```
# Ativando a função softmax com a saída da camada densa 2
activation2.forward(dense2.output)

# Imprimindo as primeiras 5 amostras do batch de saída da função
softmax
print(activation2.output[:5])
```

Depois de criarmos a classe da função de ativação softmax, iremos usá-la. Para isso, iremos configurar as camadas e após isso, entrar com os dados, assim como no código acima.

12.3. Principais diferenças entre ReLU e Softmax

- **ReLU (Rectified Linear Unit)**:
  A **ReLU** (Seção 12.1) é uma função de ativação **intermediária** usada para ativar neurônios dentro de uma rede neural. Sua principal função é **introduzir não linearidade** na rede, ajudando o modelo a aprender representações mais complexas. Ela é normalmente usada nas camadas ocultas (hidden layers) da rede.
- **Softmax**:
  O **Softmax** (Seção 12.2) é uma função de ativação usada principalmente na **camada de saída** de redes neurais para **classificação multiclasse**. Ela converte as saídas da rede em **probabilidades**, representando a probabilidade de uma amostra pertencer a cada classe. A soma das probabilidades será sempre 1, o que é útil para tarefas de **decisão**.

12.4. Recapitulação

A função ReLU (Seção 12.1) apenas "liga" e "desliga" os neurônios dependendo da entrada, permitindo que valores positivos passem e bloqueando valores negativos. Por exemplo:

- Para um valor de entrada de $x = -5$, a saída será $f(x)=0.
- Para $x = 2$, a saída será $f(x) = 2$

Já a função Softmax (Seção 12.2), transforma um vetor de valores reais em um conjunto de **probabilidades**. Essas probabilidades indicam a chance de cada classe ser a correta, e a soma dessas probabilidades deve ser igual a 1. Por exemplo:

Um cenário de classificação multiclasse com 3 classes, se o modelo gerar as saídas $[3, 1, 0.2]$, o softmax converterá essas saídas em probabilidades:

Após a aplicação da função de ativação, você terá algo como $[0.8, 0.1, 0.1]$, que são as probabilidades de cada classe:

- Classe 1 tem 80% de chance de ser a correta.
- Classe 2 tem 10% de chance de ser a correta.
- Classe 3 tem 10% de chance de ser a correta.

13. Calculando perda

O cálculo de perda (ou função de custo) em redes neurais é uma medida que quantifica o quão boa ou quão ruim é a previsão feita pela rede em relação aos valores reais. Esse valor de perda é fundamental no treinamento de redes neurais, pois o seu valor guia o processo de ajuste dos pesos durante a otimização.