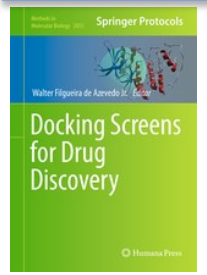
Aprendizado Profundo

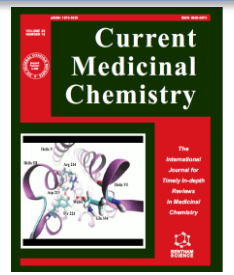
Prof. Dr. Walter F. de Azevedo, Jr.

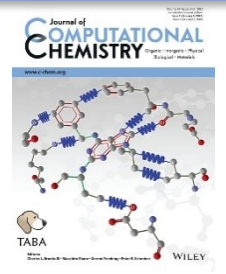
walter@azevedolab.net

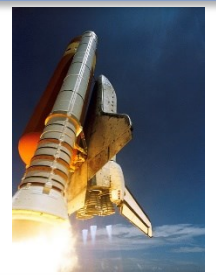
Biography 01♥
Biography 02♥
Biography 03♥
Biography 04♥

Frontiers Section Editor (Bioinformatics and Biophysics) for the Current Drug Targets ISSN: 1873-5592
Section Editor (Bioinformatics in Drug Design and Discovery) for the Current Medicinal Chemistry ISSN: 1875-533X

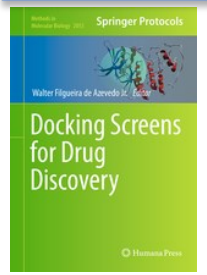


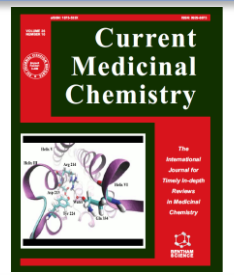


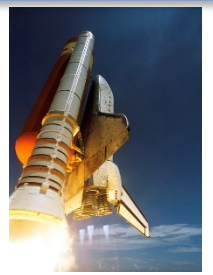




Conteúdo



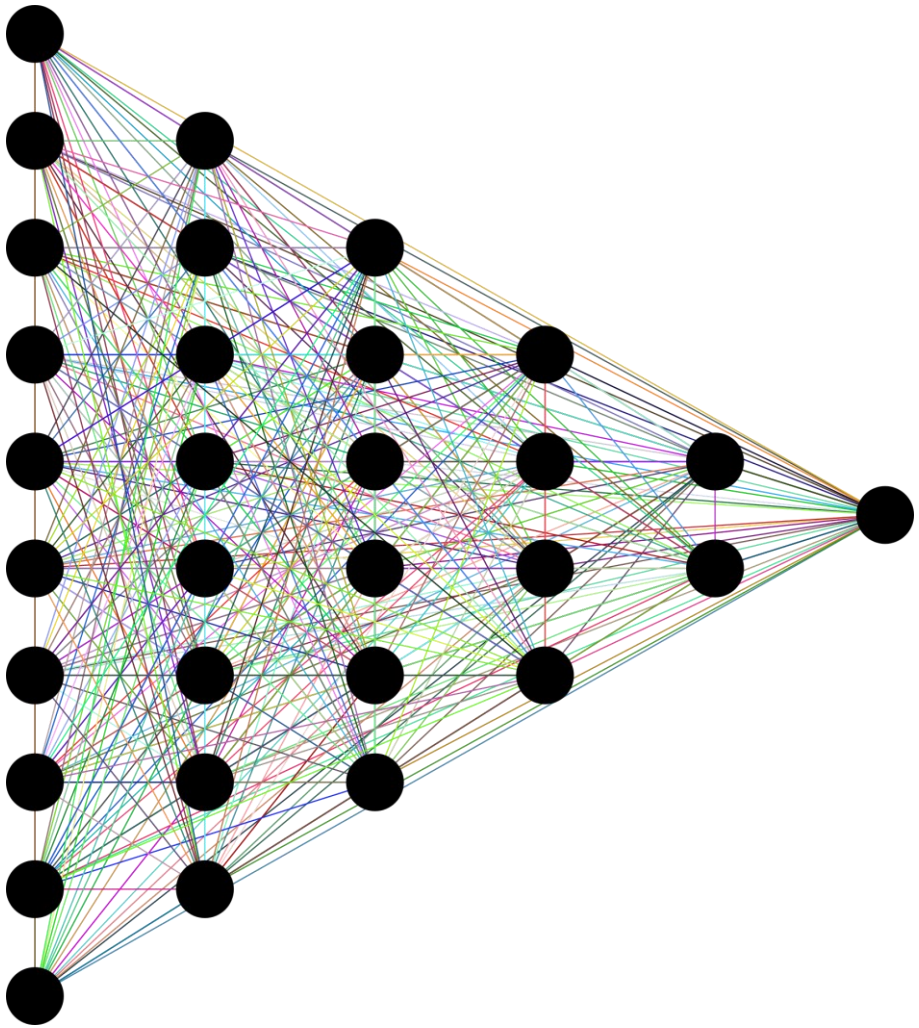




Resumo

Hoje nosso foco é no aprendizado profundo (*deep learning*). Adotamos uma abordagem intuitiva onde exploraremos a inspiração biológica das redes neurais para temos uma base pare entendermos o funcionamento das abordagens modernas de aprendizado profundo. Veremos o desenvolvimento de um programa de aprendizado profundo que usa as bibliotecas Keras e TensorFlow.

Palavras-chave: aprendizado de máquina, *machine learning*, modelo de aprendizado de máquina, aprendizado profundo, *deep learning*, redes neurais, perceptron, classificação, classificadores, Python, Scikit-Learn, Anaconda, NumPy, Matplotlib, Pandas, Jupyter, TensorFlow, Keras, curva ROC, função de perda, função de ativação, gradiente descendente, diabetes.



Fonte: https://pixabay.com/vectors/neural-network-thought-mind-mental-3816319/

Cérebro Humano

O cérebro humano é considerado por muitos como o mais capaz entre os animais do planeta Terra. Considerando-se que, um maior número de neurônios significa maior poder cognitivo, espera-se que o cérebro humano seja o campeão entre os animais em número de neurônios. Na verdade, apesar de muitos livros textos estabelecerem o número redondo de 100 bilhões de neurônios no cérebro (10^{11}) (Williams & Herrup, 1988), este número ainda é motivo de grande debate.

Referência: Williams RW, Herrup K. The control of neuron number. Annu Rev Neurosci. 1988; 11: 423–453. doi: 10.1146/annurev.ne.11.030188.002231. PMID: 3284447 PubMed

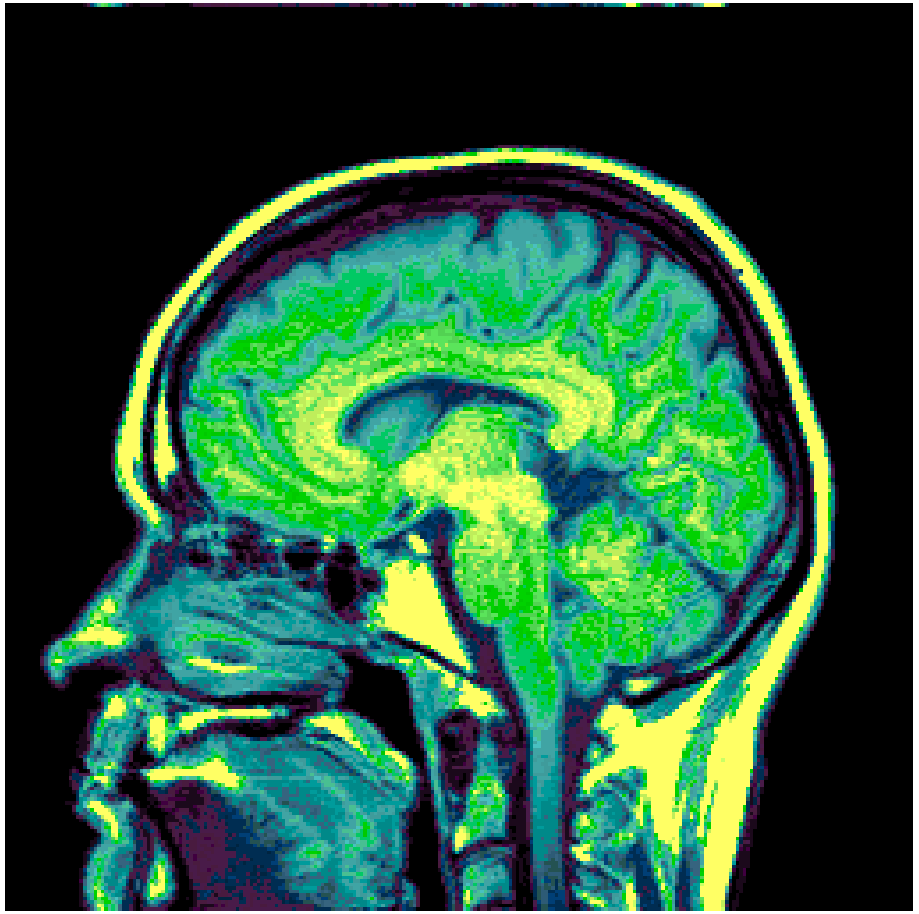
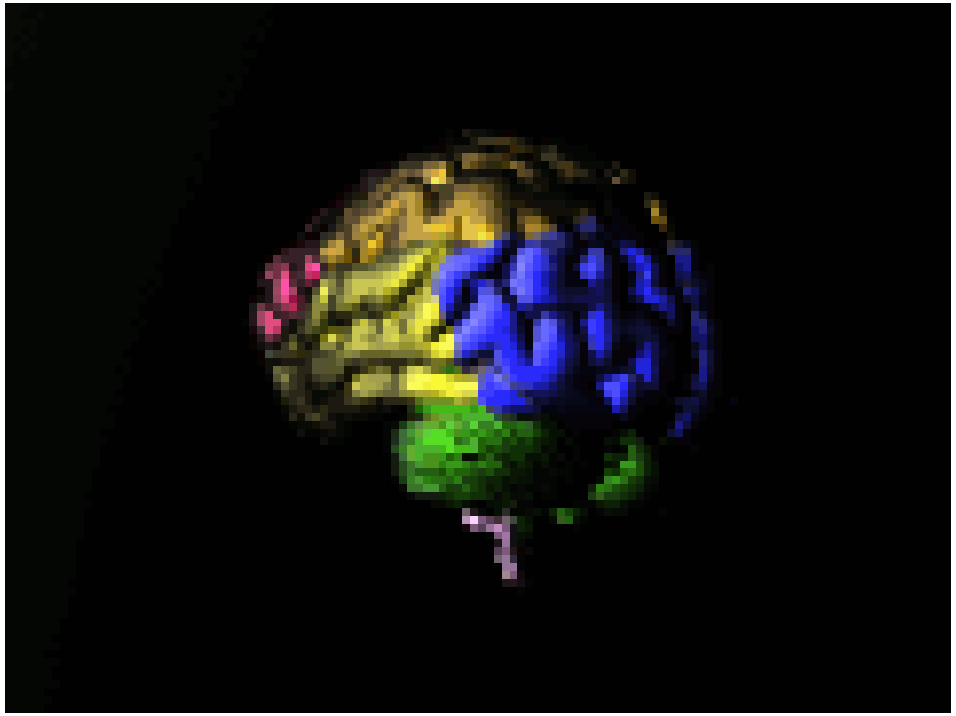


Imagem de CAT scan do cérebro.
Fonte:
http://netanimations.net/Moving_Animated_Heart_Beating_Lungs_Breathing_Organ_Animations.htm#.UXKY4rXvuSp

Cérebro Humano

Se considerarmos que o cérebro humano é formado por aproximadamente 10^{11} neurônios, sendo que cada uma destas células pode formar até 10.000 conexões, temos que o cérebro humano pode apresentar até 10^{14} sinapses. Tomamos um valor médio de 10^{3} sinapses por neurônio. Um estudo sobre o assunto (Azevedo *et al.*, 2009) estimou o número em 86,1 ± 8,1 bilhões de neurônios ($8{,}61.10^{10}$) num adulto.

Revisões posteriores, sobre o número de neurônios no cérebro humano, ficam entre 75 e 124 bilhões (Lent, 2012), assim, o número de 100 bilhões é um valor médio das estimativas.

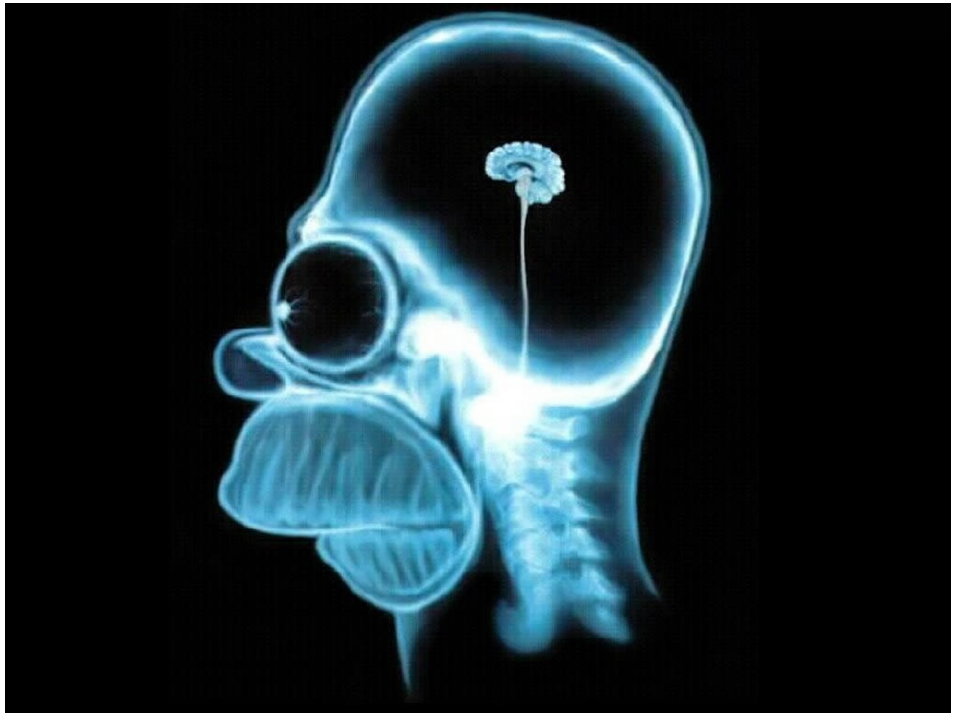


Concepção artística do cérebro humano.
Fonte : http://www.whydomath.org/

Referência:
Azevedo FA, Carvalho LR, Grinberg LT, Farfel JM, Ferretti RE, Leite RE, Jacob Filho W, Lent R, Herculano-Houzel S. Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. J Comp Neurol. 2009; 513(5): 532–541. doi: 10.1002/cne.21974. PMID: 19226510 PubMed

Cérebro Humano

A complexidade dos pensamentos e do trânsito de sinais no organismo humano são resultados da interação entre neurônios conectados. O impressionante número de conexões entre os neurônios cria um sistema altamente complexo envolvendo 10^{14} sinapses. Os resultados da ação dessas sinapses vemos a cada segundo de nossas vidas, pensando, criando e aprendendo... As interações, que geram padrões complexos são resultados das sinapses entre as células. Resumindo, tudo que pensamos e lembramos é resultado das interações dessa rede complexa de sinapses. Iremos ver as principais características das sinapses.



Segundo alguns autores, o cérebro humano tem aproximadamente 10^{14} sinapses, uns apresentam um número menor...
Fonte:
http://images.fanpop.com/images/image_uploads/Homer-Brain-X-Ray-the-simpsons-60337_1024_768.jpg

Estrutura Básica do Neurônio

No nosso estudo da eletricidade na célula, focaremos na membrana celular, boa parte destes fenômenos serão discutidos para entendermos o funcionamento elétrico dos neurônios. Os **neurônios** são **células nucleadas**, que apresentam um corpo central chamado de **soma**. Essas células apresentam grandes variações de forma, assim para os propósitos dos nossos estudos, vamos considerar que o neurônio apresenta a estrutura básica, mostrada no diagrama esquemático ao lado. No diagrama temos, além do corpo celular, **dendritos** e um terminal único, chamado **axônio.** O axônio é responsável pela transmissão do **impulso nervoso**. Podendo ser bem extenso, comparado com o resto do neurônio.

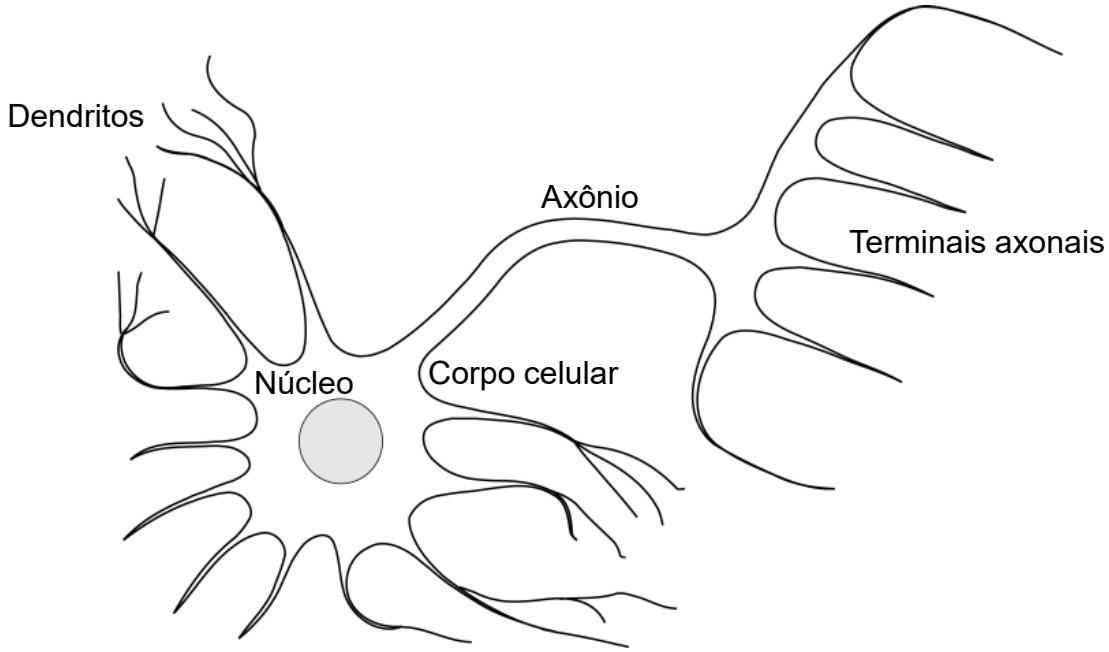


Diagrama esquemático simplificado de um neurônio.

Modelo de Hodgkin-Huxley

O **modelo de Hodgkin-Huxley** foi proposto em 1952 (Hodgkin and Huxley, 1952) para modelar o **potencial de ação** do axônio de **sépia**. A sépia é um **cefalópode** que apresenta um axônio com diâmetro maior que o axônio de vertebrados, o que facilita inserção de eletrodos para medição do potencial de membrana.

Referência: Hodgkin AL, Huxley AF. A quantitative description of membrane current and its application to conduction and excitation in nerve. J Physiol. 1952; 117(4): 500–544. doi: 10.1113/jphysiol.1952.sp004764. PMID: 12991237 PubMed

Modelo de Hodgkin-Huxley

O **modelo de Hodgkin-Huxley** é um modelo computacional, sendo considerado o primeiro modelo da abordagem de **biologia de sistemas**. Esse modelo descreve a resposta do axônio de sépia a diferentes estímulos elétricos. Temos a implementação do modelo computacional de Hodgkin-Huxley (modelo HH) em diversos programas. Apresentaremos aqui um que foi implementado na linguagem MatLab, chamado HHsim que está disponível no site http://www.cs.cmu.edu/~dst/HHsim/ . A partir do simulador do potencial de ação (Hhsim), podemos testar diferentes tipos de estímulos elétricos aplicados ao axônio, bem como o efeito de moléculas que interagem com os canais iônicos.

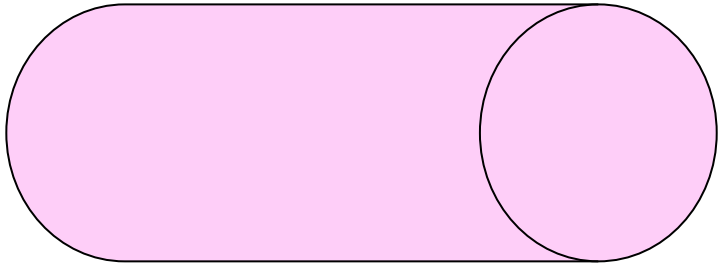
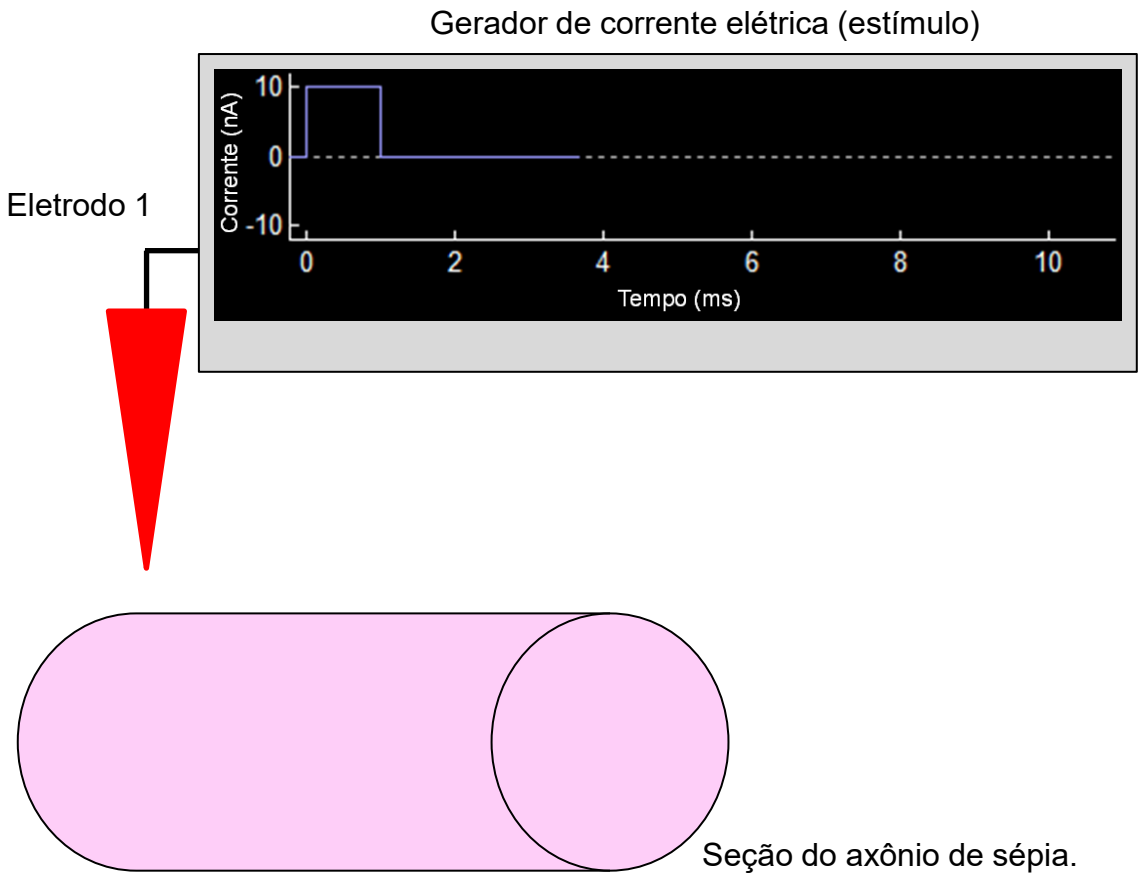


Diagrama esquemático de uma seção do axônio de sépia.

Axônio pré-sináptico da sépia, colorido em rosa para destaque.
Fonte: http://dels-old.nas.edu/USNC-IBRO-USCRC/resources_methods_squid.shtml

Modelo de Hodgkin-Huxley

O diagrama esquemático abaixo ilustra o arranjo experimental simulado no HHsim. Temos o cilindro representando uma seção do axônio da sépia, onde foram inseridos 2 eletrodos. Temos o eletrodo 1 responsável pelo estímulo, que será medido em unidades de corrente elétrica, nA (nanoAmpére, 10^{-9} A).

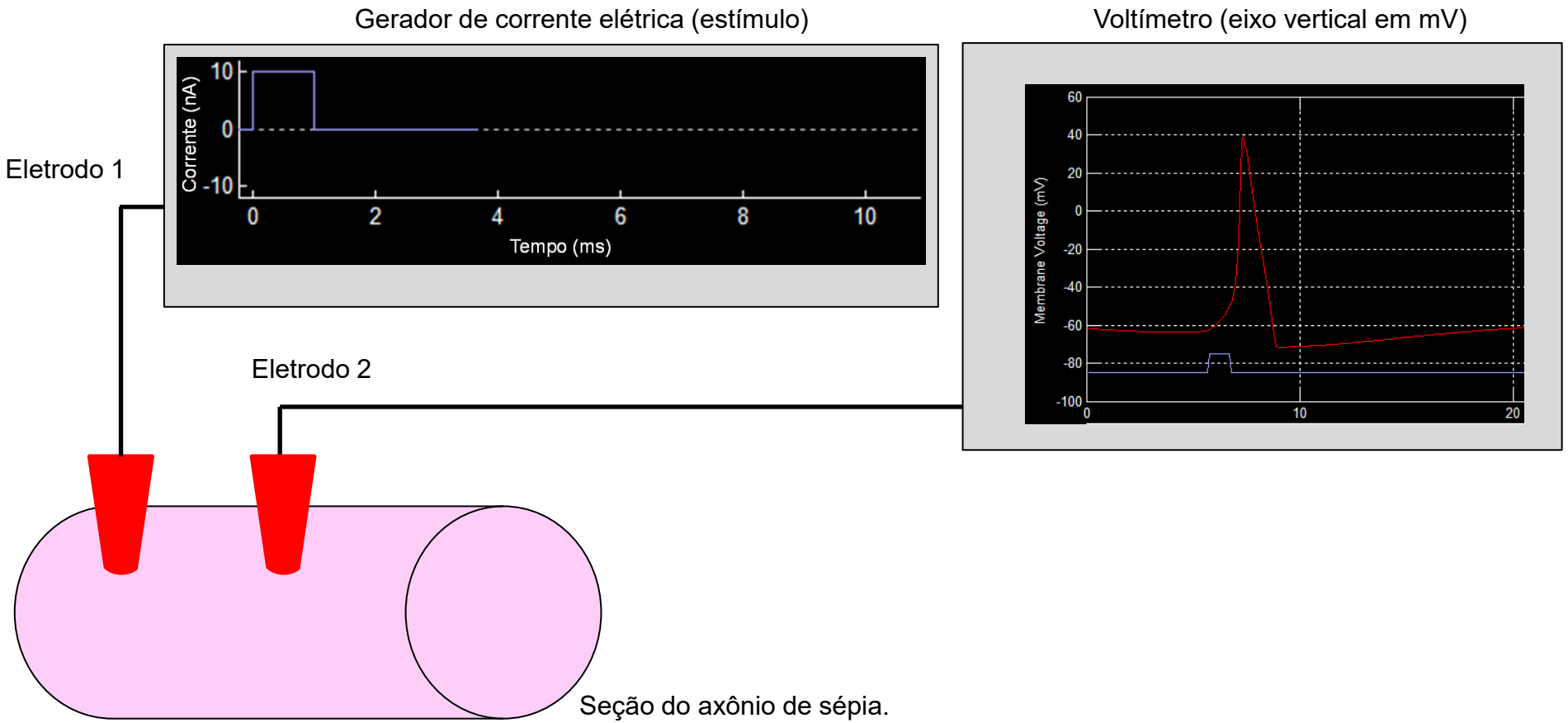


Seção do axônio de sépia.

Referência: Hodgkin AL, Huxley AF. A quantitative description of membrane current and its application to conduction and excitation in nerve. J Physiol. 1952; 117(4): 500–544. doi: 10.1113/jphysiol.1952.sp004764. PMID: 12991237 PubMed

Modelo de Hodgkin-Huxley

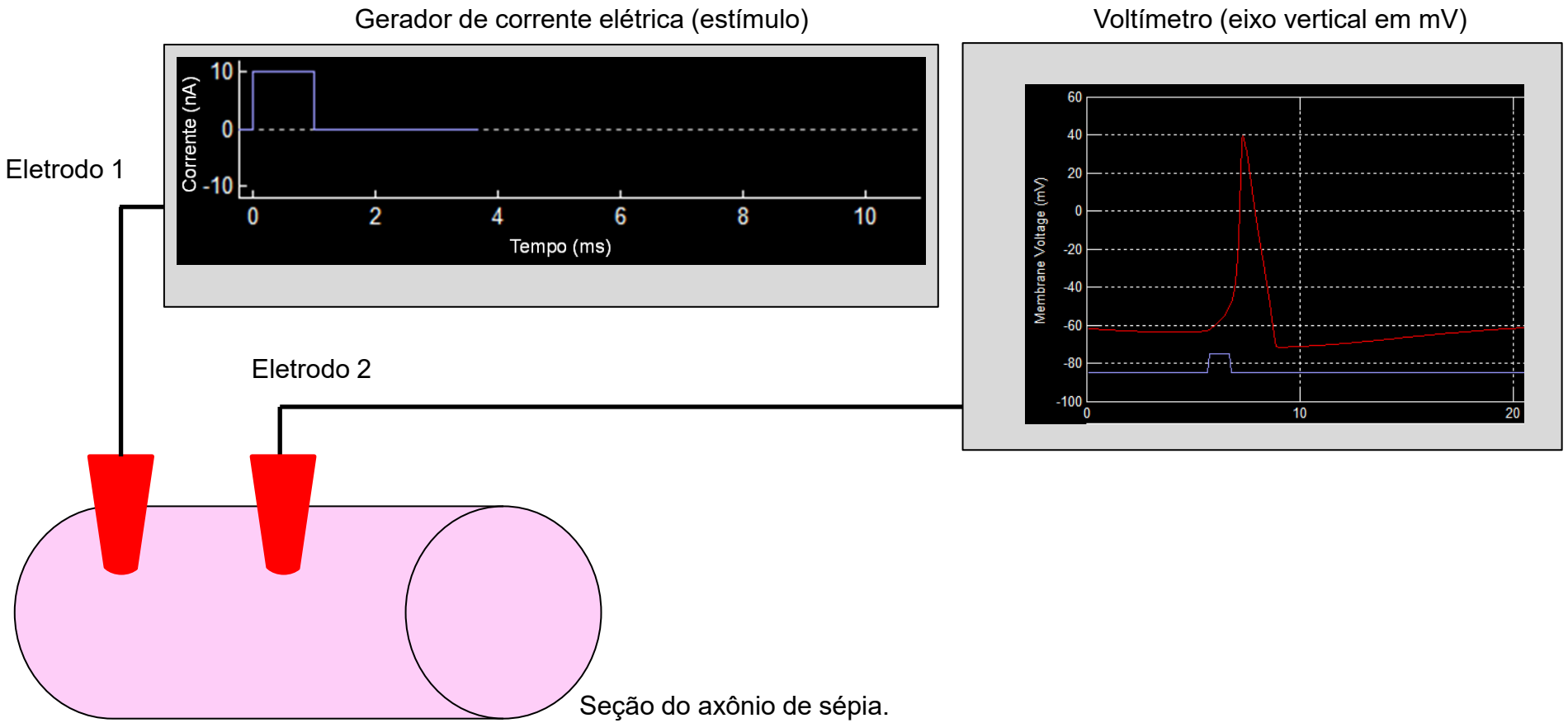
Temos um segundo eletrodo (eletrodo 2) inserido após o eletrodo 1. O posicionamento de eletrodo 2 indica que ele está mais próximo do terminal axonal que o eletrodo 1. Assim, o estímulo gerado no eletrodo 1 pode propagar-se ao longo do axônio e ser registrado no eletrodo 2. O eletrodo 2 está ligado a um voltímetro, que registra o potencial de membrana em mV em função do tempo (eixo horizontal).



Seção do axônio de sépia.

Modelo de Hodgkin-Huxley

Na situação ilustrada abaixo, temos que o voltímetro mostra a evolução temporal (eixo horizontal) do potencial de membrana, num período de 20 ms, suficiente para vermos todas as fases do potencial de ação (despolarização, repolarização e hiperpolarização).



Seção do axônio de sépia.

Modelo de Hodgkin-Huxley

Vamos usar o HHsim para destacar as características do potencial de ação do neurônio. Na figura temos a situação de potencial de repouso. A linha vermelha indica o potencial da membrana em repouso, a linha roxa indica o estímulo aplicado, a linha amarela a condutância do Na^+ e a verde a condutância do K^+. As condutâncias indicam a facilidade com que os íons atravessam a membrana. Quanto maior a condutância, mais facilmente o íons cruza a membrana celular. No repouso temos as condutâncias iguais à zero (linhas verde e amarela).

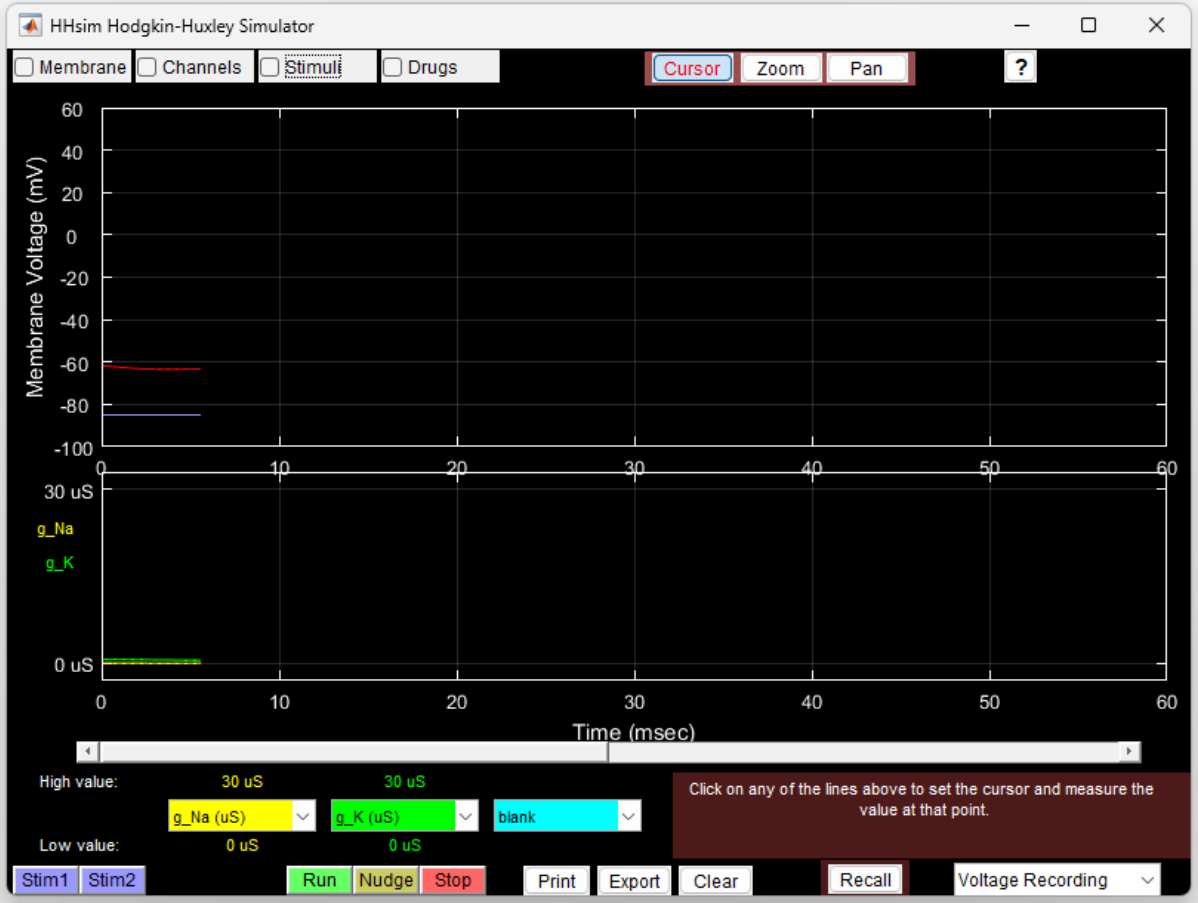


Gráfico do potencial contra o tempo (linha vermelha), gerado pelo HHsim.

Modelo de Hodgkin-Huxley

Aplicamos um estímulo, linha roxa, temos que o potencial de membrana atinge uma valor acima do potencial limiar (linha vermelha). Em tal situação, abrem-se os canais de Na^+ dependentes de voltagem. Confirmarmos a situação, verificando a condutância do Na^+ (linha amarela), que começa a subir, indicado o influxo de Na^+. O eixo horizontal é o do tempo. Todo evento está registrado em pouco mais de 20 ms.

Comparando-se as condutâncias, vemos que a condutância do Na^+ (linha amarela) atinge o valor máximo, antes da a condutância do K^+ (linha amarela). Isto deve-se ao fato do canal de Na^+ dependente de voltagem abrir-se antes do canal de K^+ dependente de voltagem

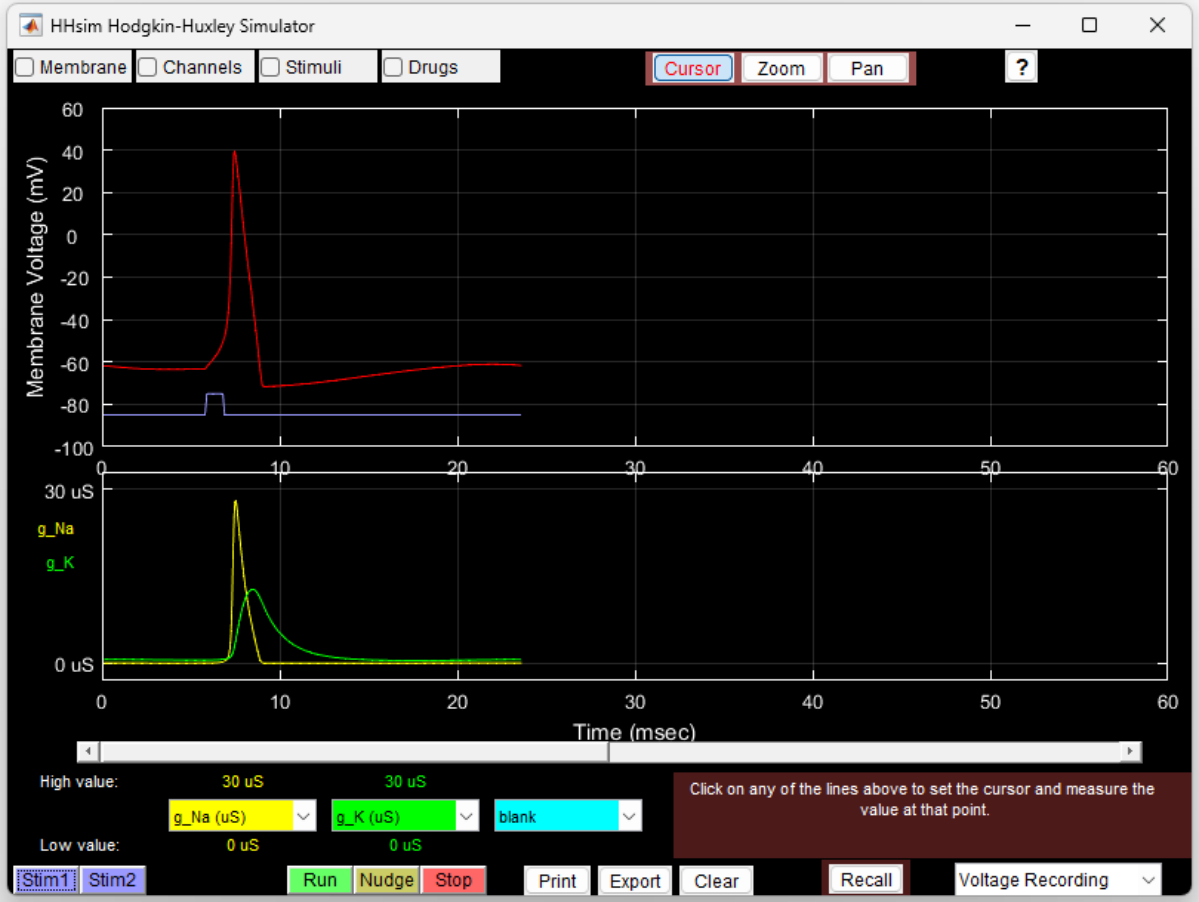


Gráfico do potencial contra o tempo (linha vermelha), gerado pelo HHsim.

Modelo de Hodgkin-Huxley

Depois de poucos milissegundos, ambos canais estão fechados, como vemos com as condutâncias retornando para o valor zero. Depois de mais alguns milissegundos, o potencial de membrana (linha vermelha) retorna ao valor de repouso. Podemos dizer que o potencial de ação é um evento de tudo-ou-nada, ou seja, só ocorre o disparo se o estímulo (ou soma de estímulos) for acima de um valor de referência chamado potencial limiar.

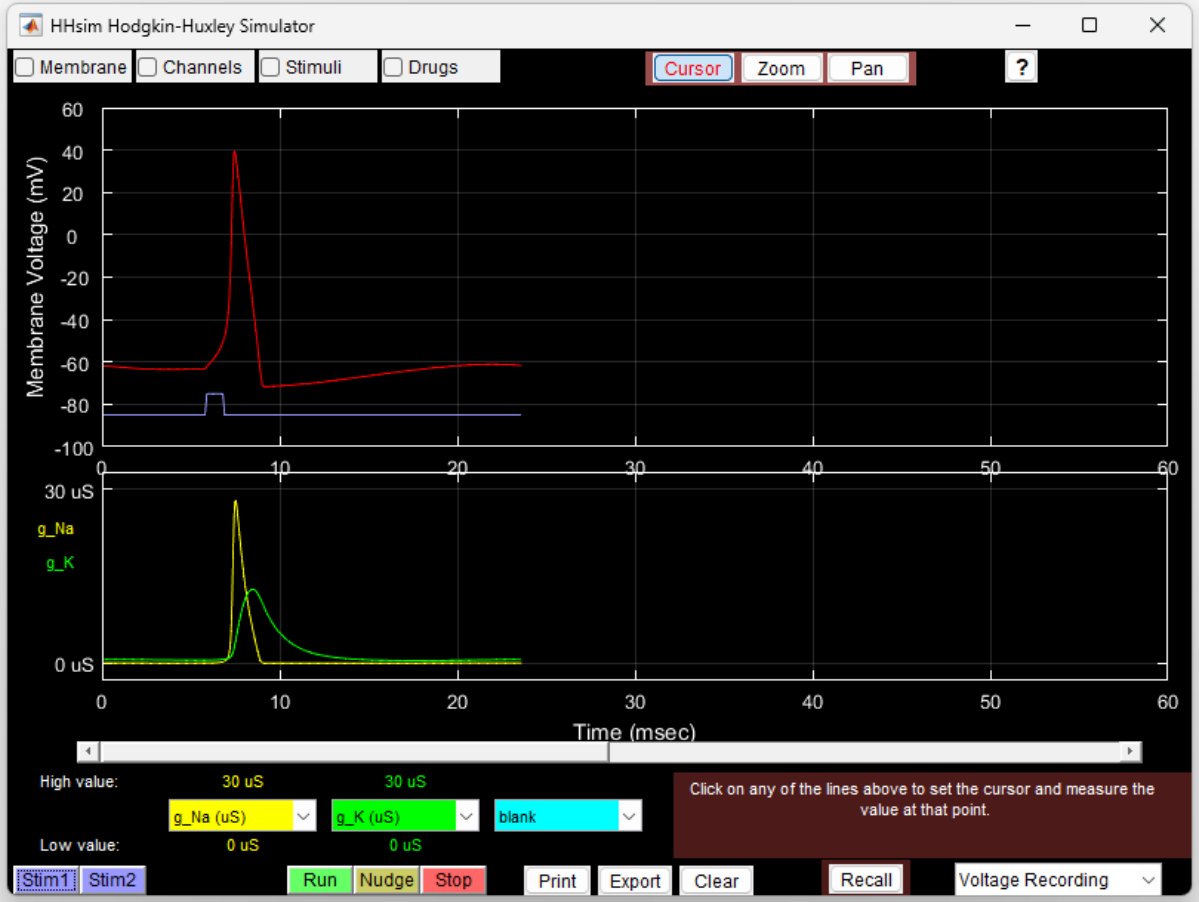


Gráfico do potencial contra o tempo (linha vermelha), gerado pelo HHsim.

Modelo de Hodgkin-Huxley

Na situação ilustrada no gráfico, temos dois estímulos (linha roxa). O primeiro estímulo é 30 % do segundo estímulo. Vemos que a resposta gerada pelo primeiro estímulo é quase nula. Observamos somente uma pequena oscilação da linha vermelha (potencial de membrana). O segundo estímulo gera o disparo do potencial de ação. O neurônio tem essa característica para evitar o seu disparo por sinais espúrios. Dizemos que o neurônio tem um potencial limiar, acima do qual estímulos causam o disparo do potencial. Estímulos abaixo do potencial limiar não geram o disparo do potencial de ação.

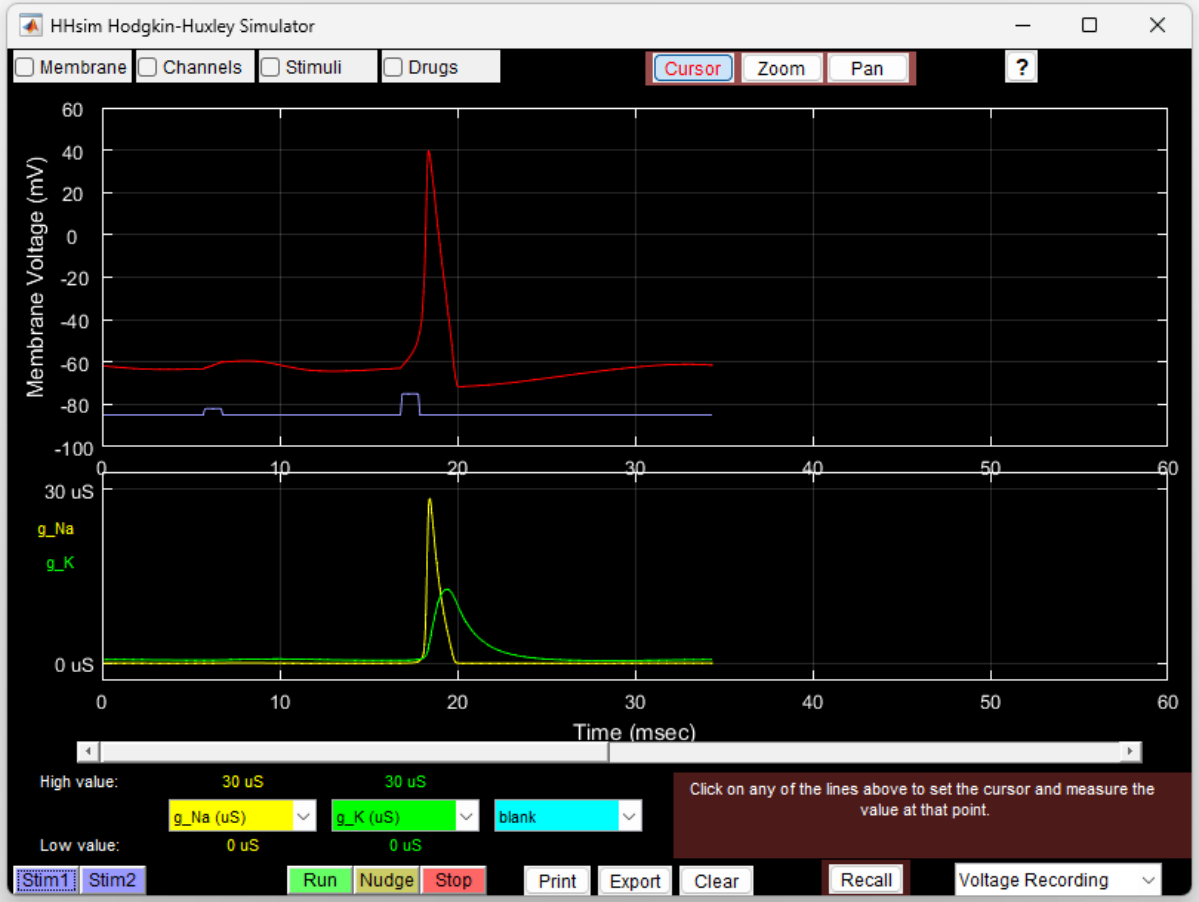
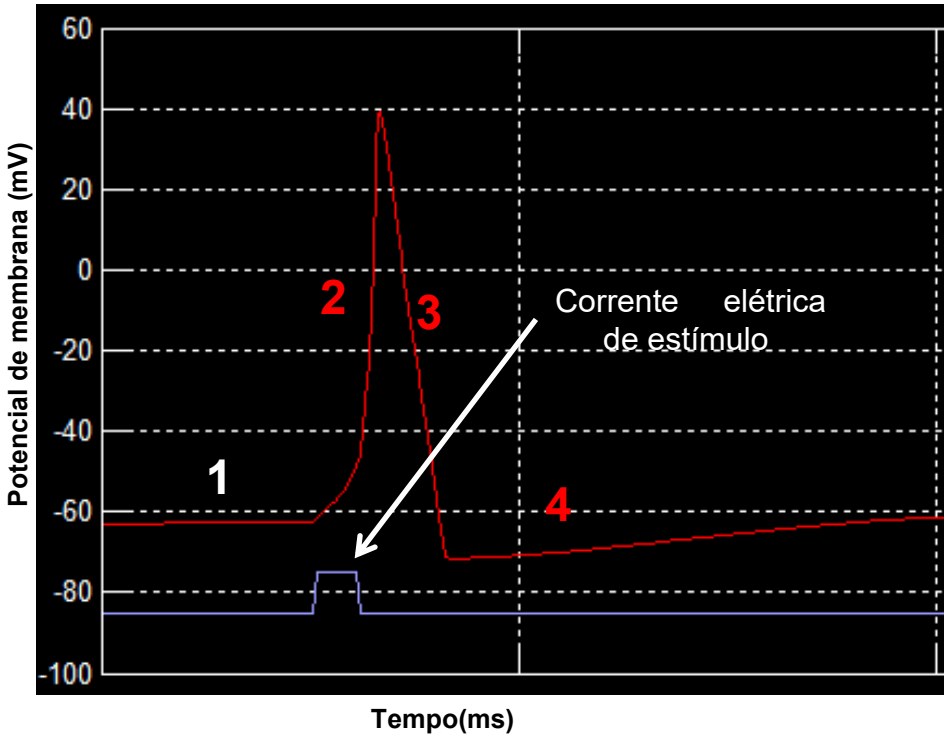


Gráfico do potencial contra o tempo (linha vermelha), gerado pelo HHsim.

Modelo de Hodgkin-Huxley

O neurônio processa os estímulos e responde ficando em repouso se a somatória dos estímulos ficar abaixo do valor limiar. Caso a somatória dos estímulos seja supere o potencial limiar, temos o disparo de um potencial de ação. Que se caracteriza por três fases denominadas de despolarização, repolarização e hiperpolarização. As fases estão em destaque no gráfico ao lado.



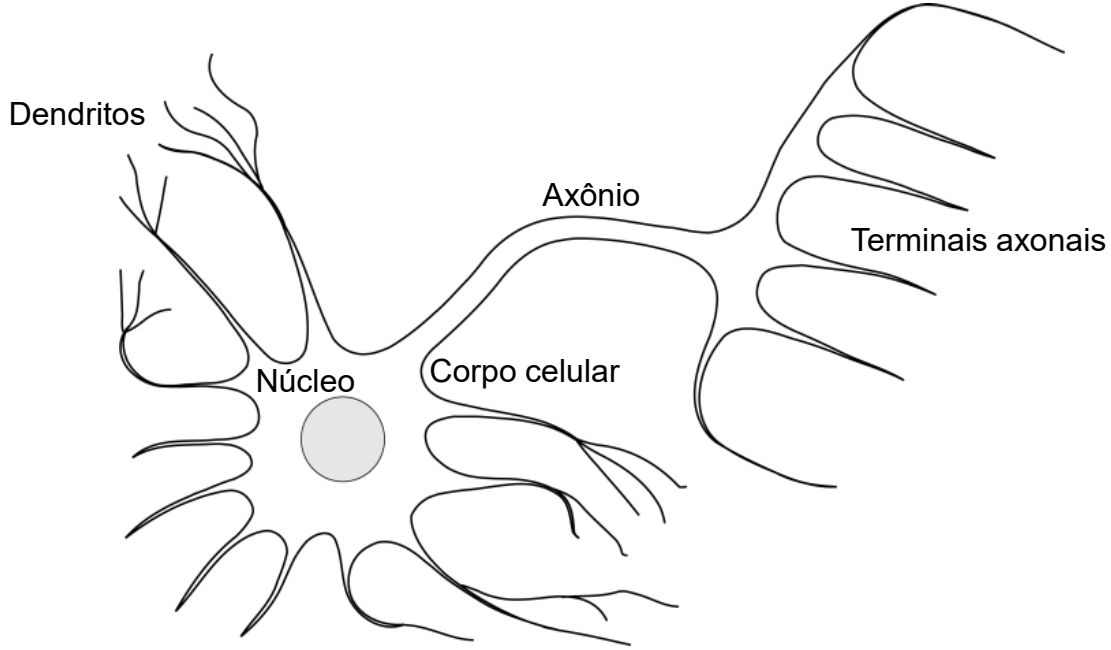
Fases indicadas no gráfico acima

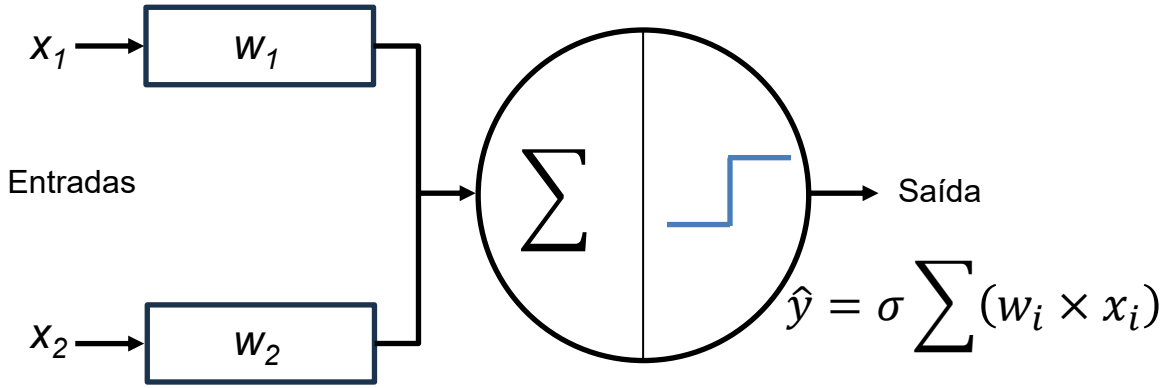
1 **Potencial de repouso**
2 **Despolarização**
3 **Repolarização**
4 **Hiperpolarização**

Potencial de ação

Redes Neurais

As **redes neurais** formam uma classe de algoritmos de aprendizado de máquina que assemelham-se aos neurônios do cérebro humano. Há uma analogia na somatória das entradas e o uso de uma **função de ativação** para emular o disparo de um potencial de ação. Acima temos a representação do nosso neurônio, o tijolo básico das redes neurais biológicas. Os dendritos podem ser pensados como as entradas. O corpo celular o processador e o axônio é a saída. Na figura abaixo à direita temos a representação de um neurônio artificial, que soma as entradas e submete o resultado a uma função de ativação (σ) (agora uma função matemática como a função degrau).

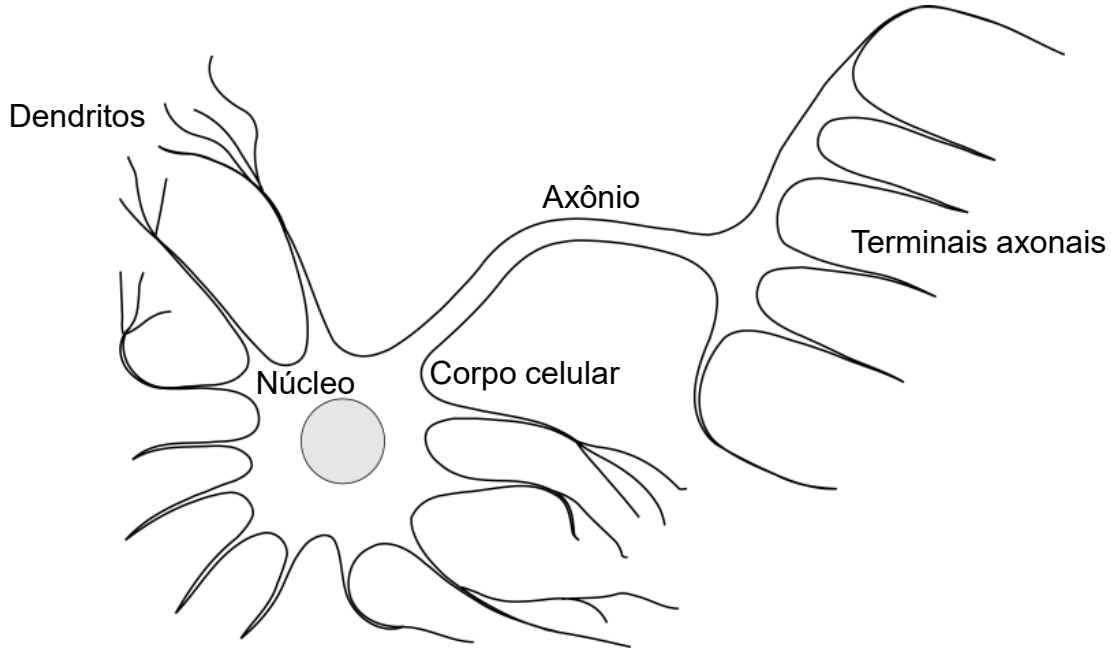


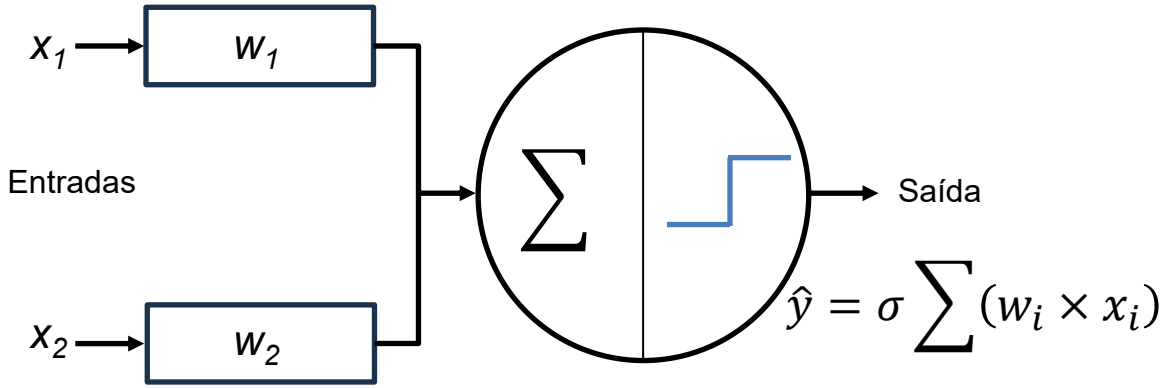


Na equação acima, w_i indica um peso que é multiplicado pela entrada x_i.

Redes Neurais

De uma forma geral é mais fácil descrevemos as redes neurais como uma função matemática que mapeia as entradas para gerarem saídas desejadas. Veremos intuitivamente um tipo básico de rede neural chamado **perceptron**. No diagrama esquemático do neurônio artificial (figura inferior à direita) temos uma rede neural de uma camada. Podemos dizer que o perceptron é simplesmente uma função matemática que toma um conjunto de entradas, faz uma computação (somatório seguida de aplicação da função de ativação (σ)) e gera como saída o resultado deste processo. A equação ao lado indica o resultado ($\hat{y}$).



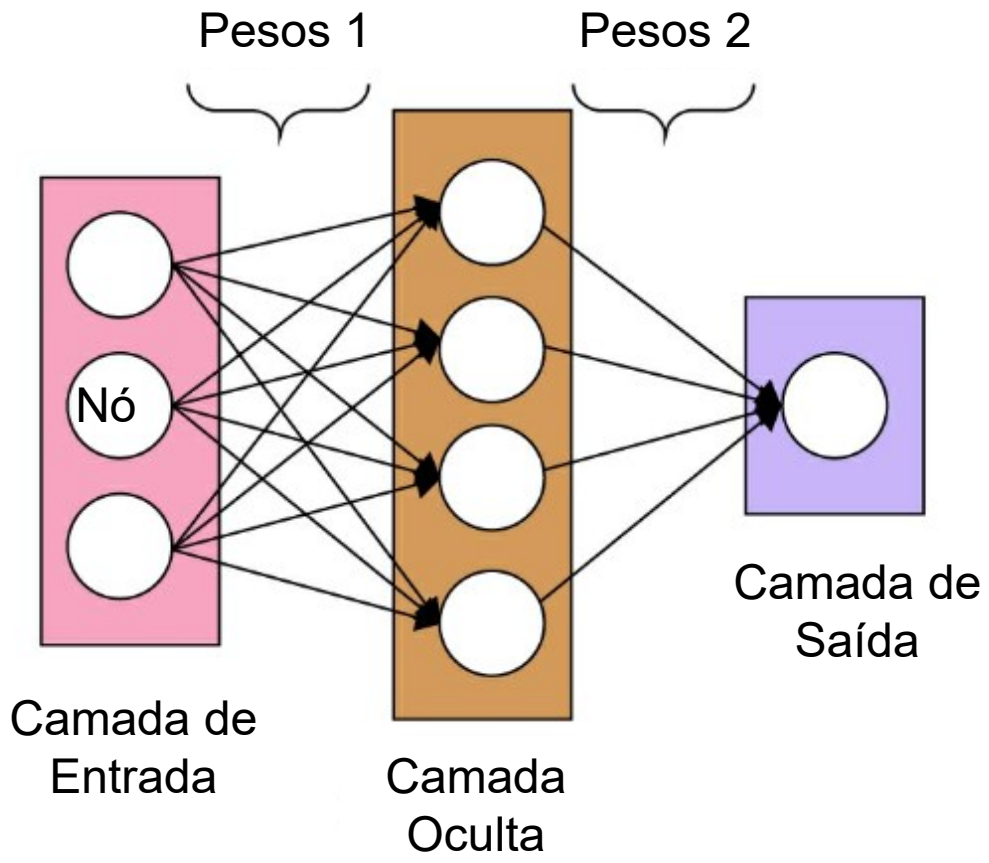


Na equação acima, w_i indica um peso que é multiplicado pela entrada x_i.

Redes Neurais

O perceptron usa uma das arquiteturas mais simples para a implementação de **redes neurais artificiais**. Sua proposta original foi descrita por Frank Rosenblatt (Rosenblatt, 1958). Generalizando, podemos dizer que redes neurais consistem dos seguintes componentes:
Uma camada de entrada, *x*
Um número arbitrário de camadas ocultas
Uma camada de saída, $\hat{y}$
Um conjunto de pesos (*w*) e vieses (bias) (*b*) entre cada camada
Uma função de ativação para cada camada oculta, σ

O diagrama esquemático ao lado ilustra a arquitetura de uma rede neural de duas camadas. **Note que a camada de entrada é normalmente excluída da contagem de camadas**.



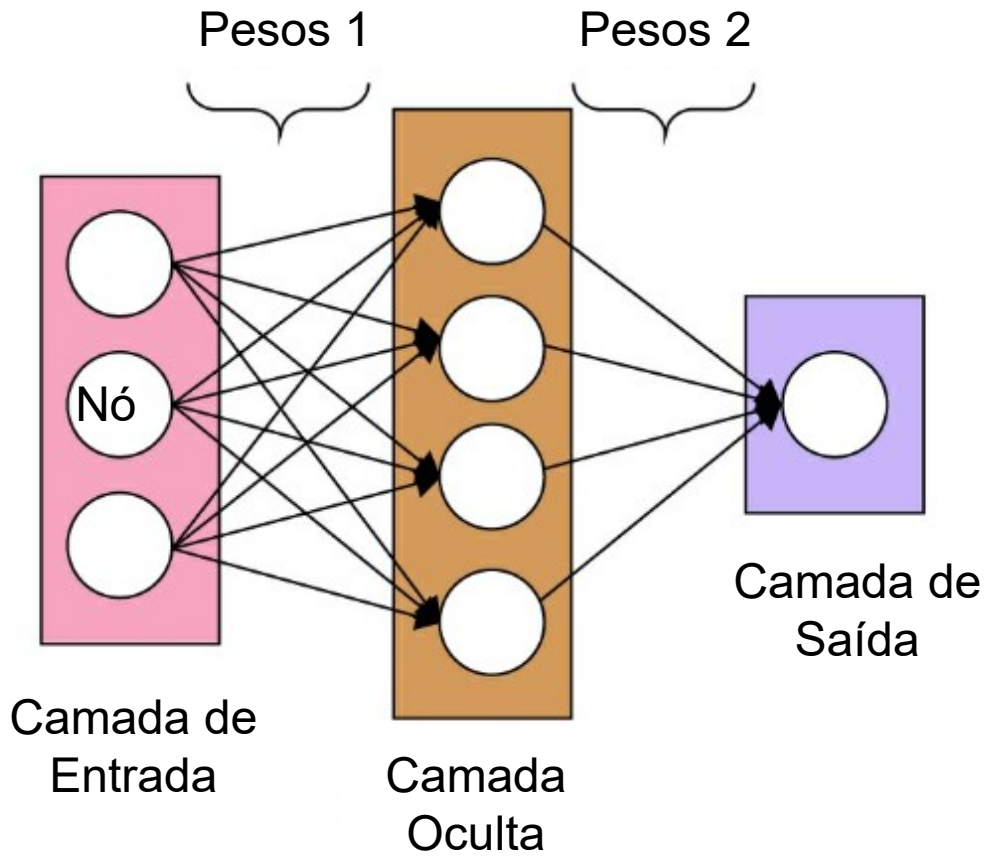
Redes Neurais

A saída ($\hat{y}$) para um perceptron de duas camada é definida pela seguinte expressão.

$$\hat{y} = \sigma(w_2\sigma(w_1x + b_1) + b_2)$$

Podemos destacar que na equação acima as únicas variáveis que afetam a saída ($\hat{y}$) são os pesos e vieses (bias) (*w*'s e *b*'s).

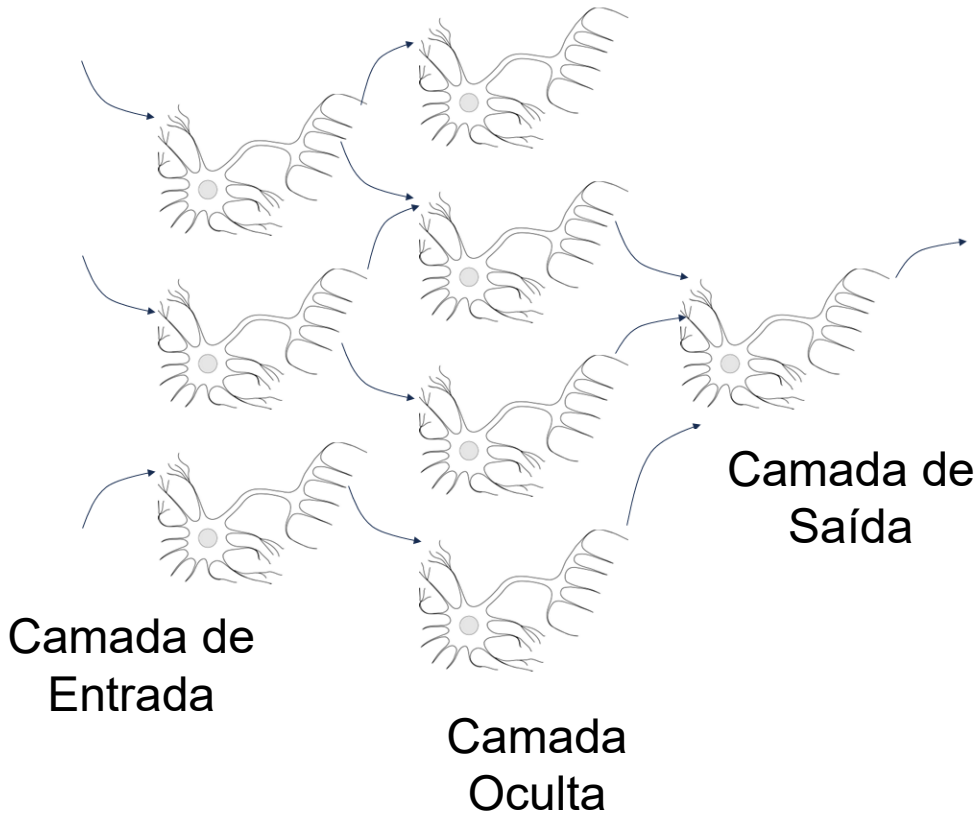
Os valores adequados de pesos e vieses (bias) determinam o poder de previsão da rede neural. Chamamos o processo de ajuste dos pesos e vieses (bias) a partir dos dados de entrada de **treinamento da rede neural**.

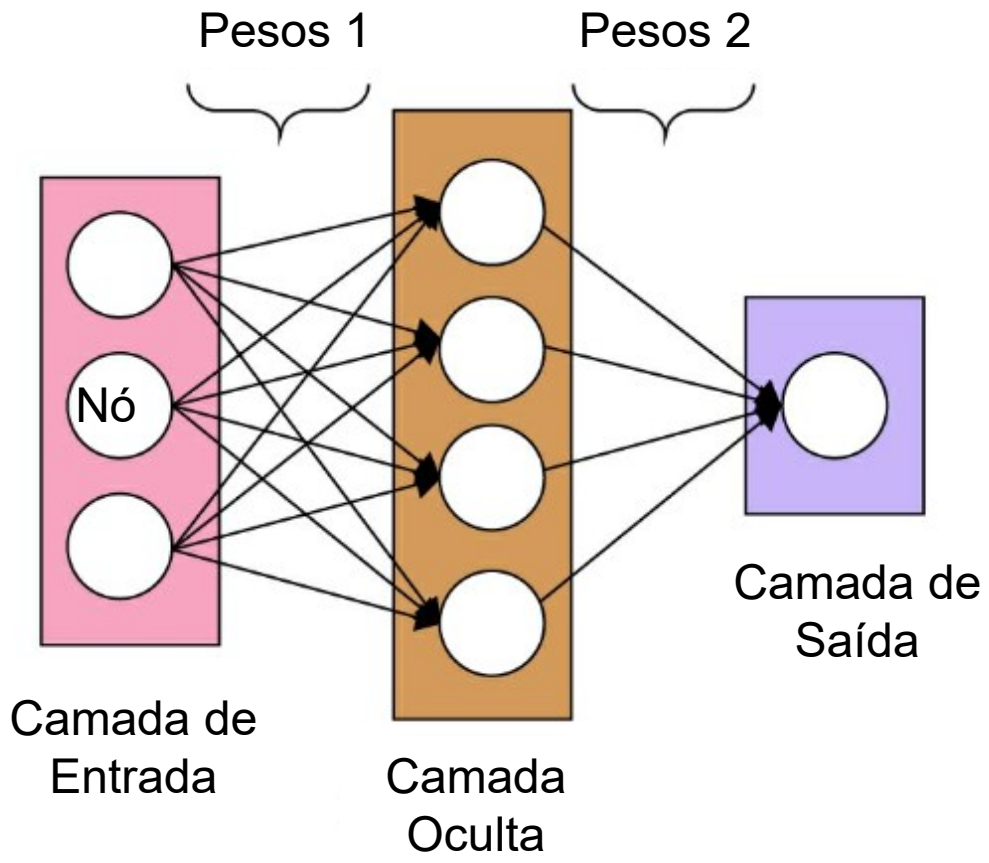


x: entradas
w: pesos
b: vieses (bias)
y: valores reais
$\hat{y}$: valores previstos pelo modelo
σ: função de ativação
loss(ŷ,y): função de perda
MSE: erro quadrático médio
$\nabla_w MSE(w)$: gradiente do *MSE*
η: taxa de aprendizado

Redes Neurais

Abaixo temos a comparação de uma rede neural biológica com uma rede neural computacional. As sinapses representam as conexões entre as camadas.



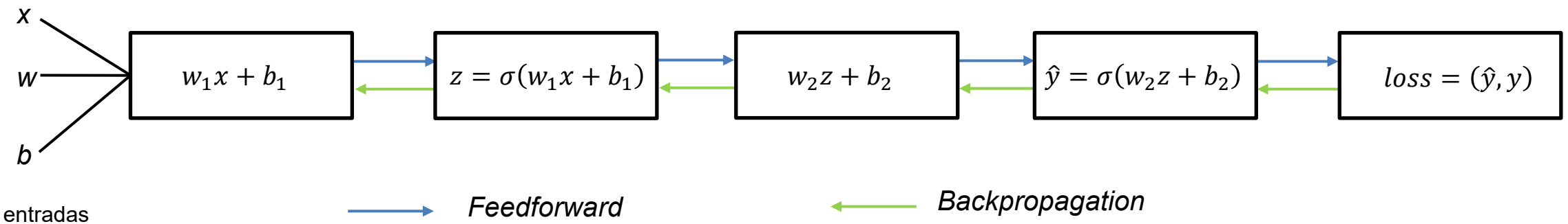


Redes Neurais

Cada iteração do processo de treinamento consiste dos seguintes passos

1. Cálculo da saída prevista ($\hat{y}$), conhecida como *Feedforward*
2. Atualização dos pesos e vieses (bias), conhecida como Retropropagação (*Backpropagation*).

Os passos estão ilustrados no diagrama esquemático abaixo.



x: entradas
w: pesos
b: vieses (bias)
y: valores reais
$\hat{y}$: valores previstos pelo modelo
σ: função de ativação
loss(ŷ,y): função de perda
MSE: erro quadrático médio
$\nabla_w MSE(w)$: gradiente do *MSE*
η: taxa de aprendizado

Redes Neurais

A função *loss(ŷ,y)* (traduzida como **função de perda**) visa avaliar o poder de previsão do modelo atual. No cálculo da função de perda temos como entrada os valores previstos ($\hat{y}$) e reais (ou experimentais) (y). Podemos usar como função de perda as métricas previamente estudadas durante o curso. No próximo código discutido nesta aula, usaremos o MSE (*mean squared error*) (erro quadrático médio). O objetivo da implementação da rede neural é chegar a um modelo que minimize a função de perda, no código a ser visto o objetivo é chegar a um modelo que minimize o MSE.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

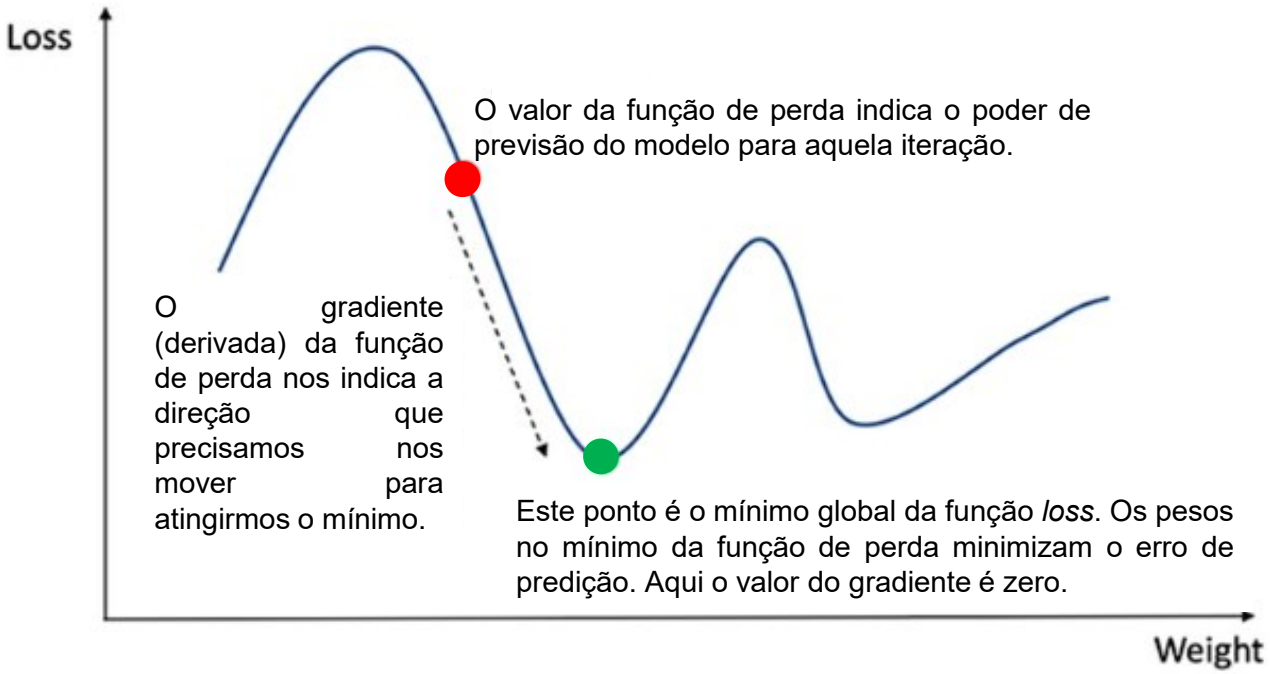
$$RMSE = \frac{1}{n}\sqrt{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}\left(y_i - \frac{\sum_{j=1}^{n} y_j}{n}\right)^2}$$

Nas equações acima, $\hat{y}_i$ indica os valores previstos por um modelo e y_i são os valores experimentais (ou reais) para n observações.

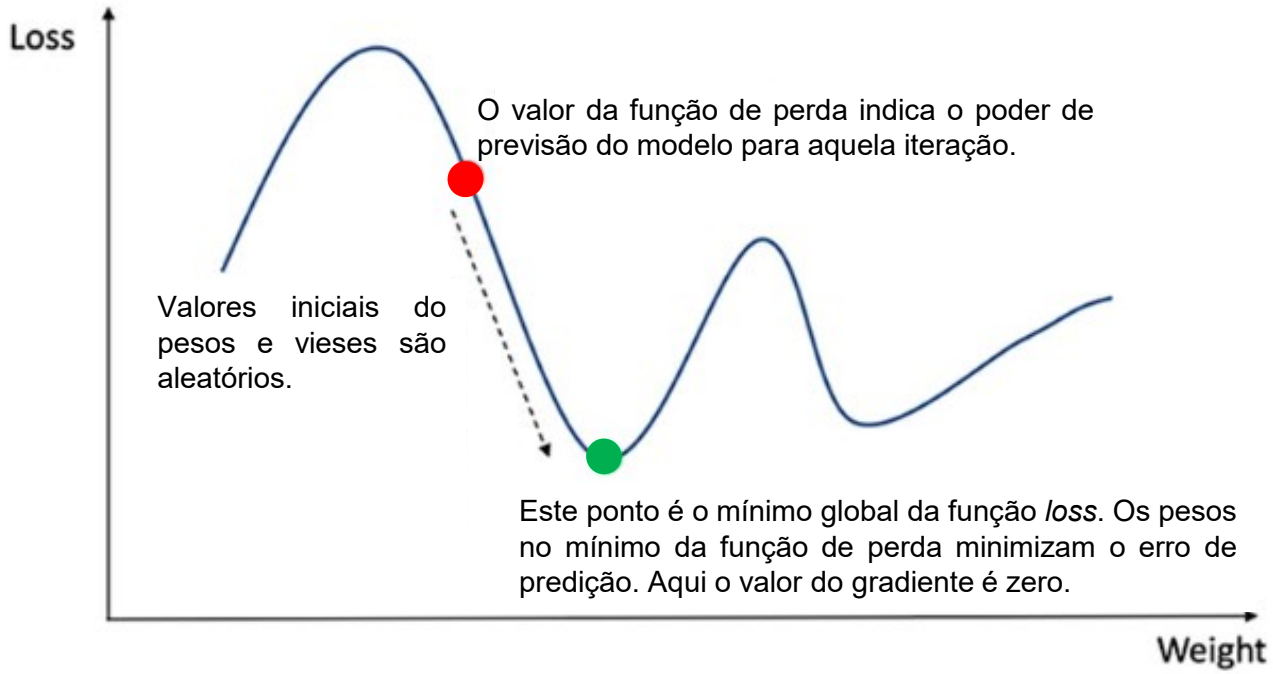
Redes Neurais

A partir da função de perda (*loss(ŷ,y)*), podemos atualizar os pesos e vieses (bias) usando o processo de retropropagação (*backpropagation*). Para determinarmos os ajustes necessários, fazemos uso da derivada parcial da função de perda com relação aos pesos e vieses. A derivada indica a inclinação da curva da função de perda. A partir da derivada, atualizamos os pesos e vieses por adição ou subtração dos valores dos pesos da iteração anterior. Esse método é chamado de **gradiente descendente**. A principal ideia do método de gradiente descendente é que a partir da derivada (gradiente) da função de perda com relação aos pesos, podemos ajustá-los de forma conveniente.



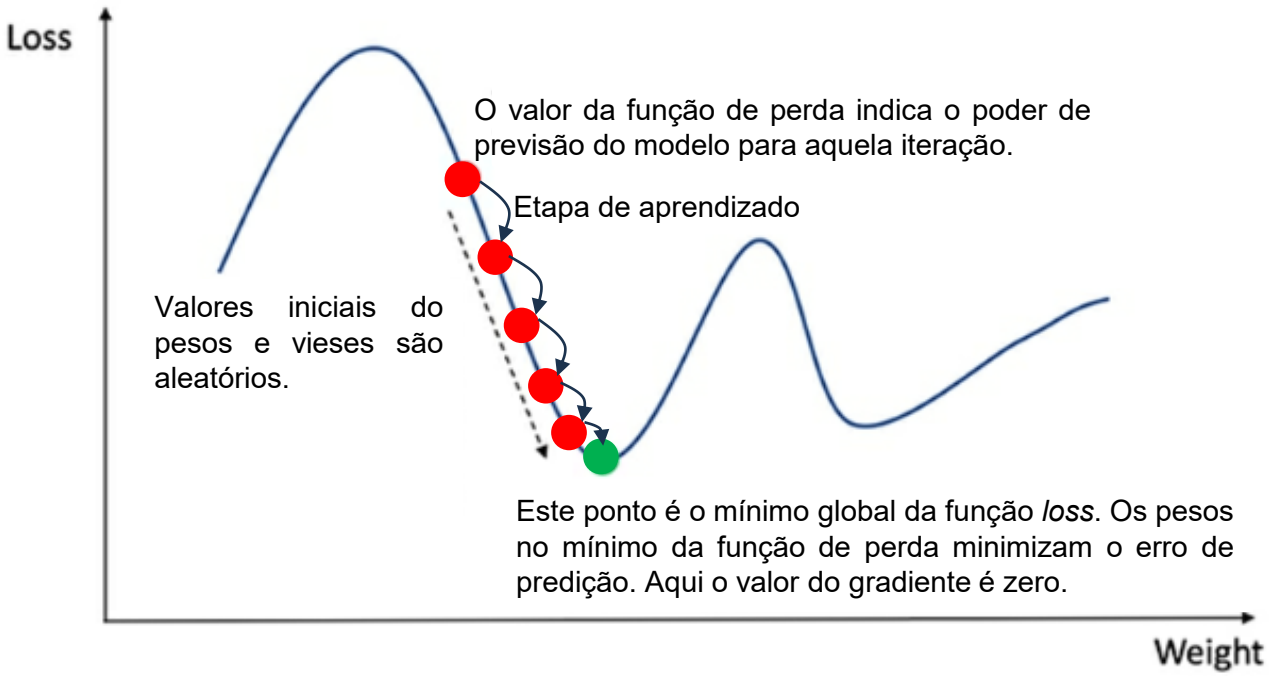
Redes Neurais

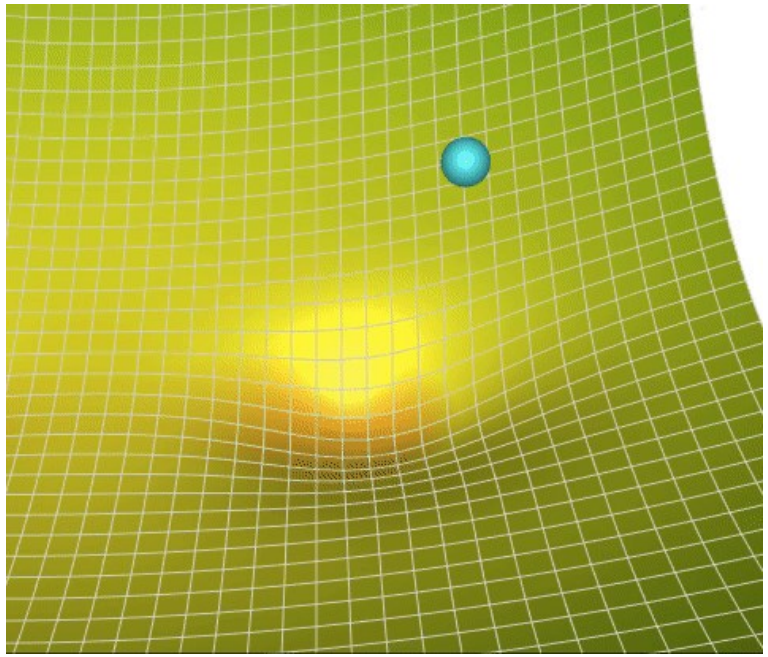
A ideia geral do gradiente descendente é ajustar iterativamente os parâmetros com o intuito de minimizar uma função de perda (também chamada de função de custo). Considere uma situação hipotética onde você está numa montanha sem abismos, onde seu objetivo é chegar à parte mais baixa. Você só consegue sentir a inclinação do terreno com os pés, pois a montanha está envolta num nevoeiro denso. A forma mais rápida de chegar à parte mais baixa é movendo-se na direção onde a inclinação é máxima. É justamente isso que o gradiente descendente realiza: calcula o gradiente local da função de perda (MSE) em relação ao vetor de parâmetro *w* e segue em direção ao gradiente descendente. Quando o gradiente é zero, você está na parte mais baixa da montanha.



Redes Neurais

Na implementação do método do gradiente descendente, iniciamos atribuindo pesos aleatórios. Podemos representar os pesos numa notação condensada na forma de um vetor (**w**). O negrito indica que temos um vetor. Inicialmente, o vetor tem valores aleatórios (isso se chama **inicialização aleatória**). Na sequência, aprimoramos gradualmente o poder de previsão do modelo (quantificado pelo o MSE), dando um passo de cada vez. Vamos no processo diminuindo a função de perda chegando ao mínimo com a convergência do algoritmo. Na figura abaixo à esquerda, cada iteração representa um ponto sobre a curva da função *loss*.

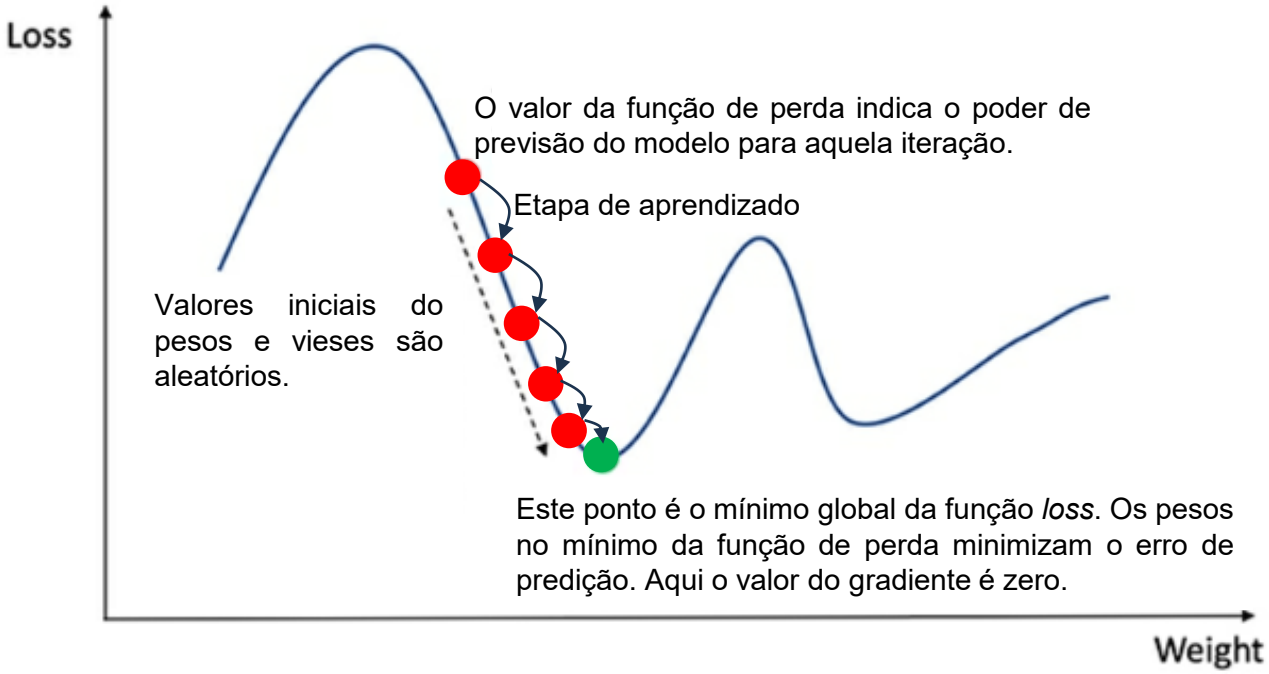


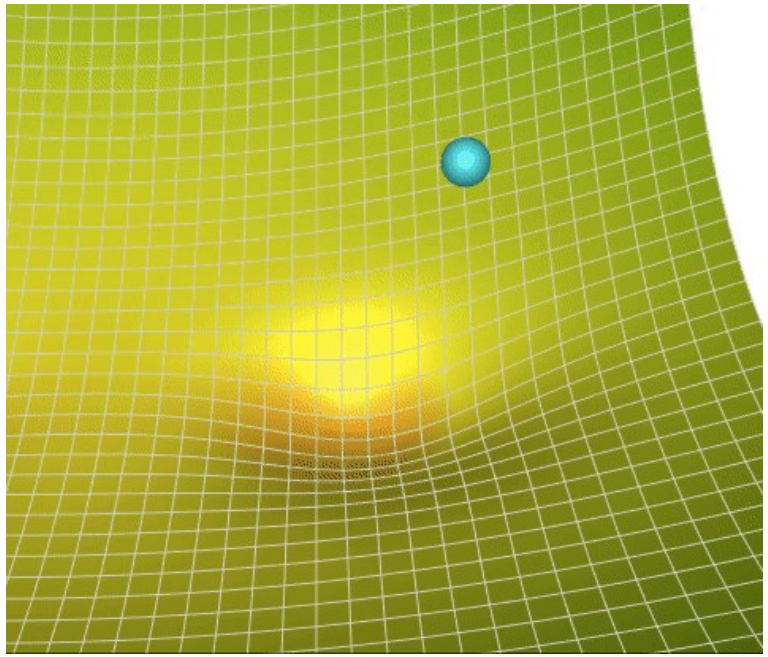


Animação ilustrando o método de gradiente descendente.

Redes Neurais

A animação abaixo à direita capta a ideia central do método de gradiente descendente. Na animação vemos um ponto com três caminhos possíveis indicados por vetores. O vetor preto é o gradiente, que indica a direção a ser seguida para atingirmos o mínimo global.

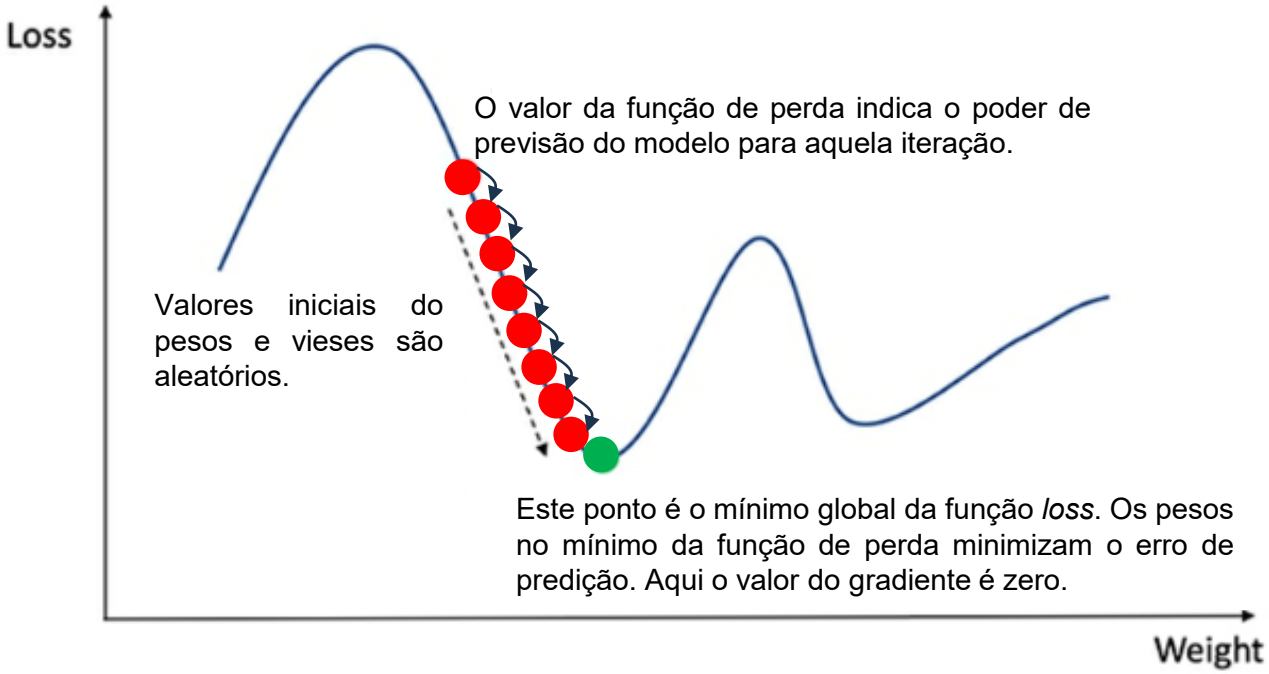


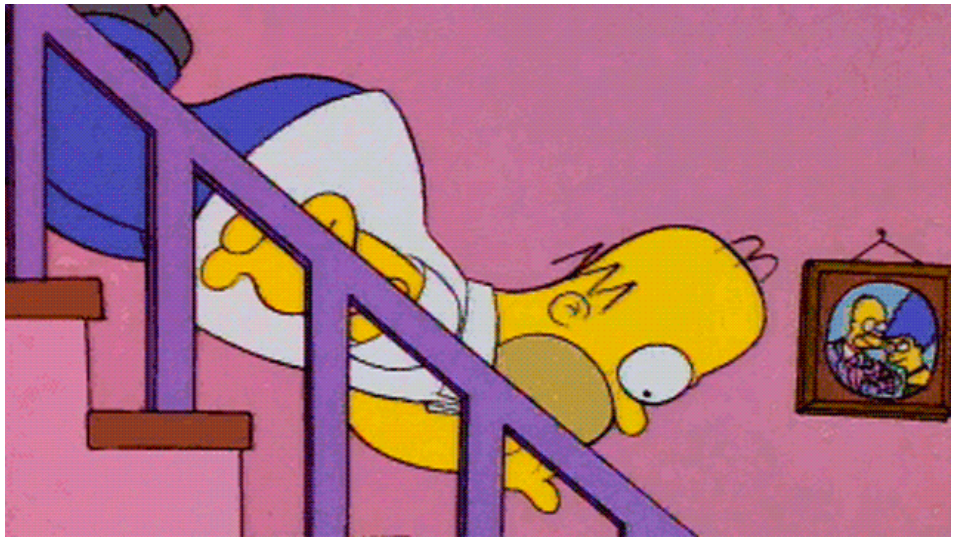


Animação ilustrando o método de gradiente descendente.

Redes Neurais

Um parâmetro importante na aplicação do método de gradiente descendente é o tamanho das etapas, determinado pelo hiperparâmetro da **taxa de aprendizado** (*learning rate*). Se a taxa de aprendizado for muito pequena, o algoritmo precisará passar por muitas iterações (**épocas**) para convergir, o que levará muito tempo (figura abaixo).

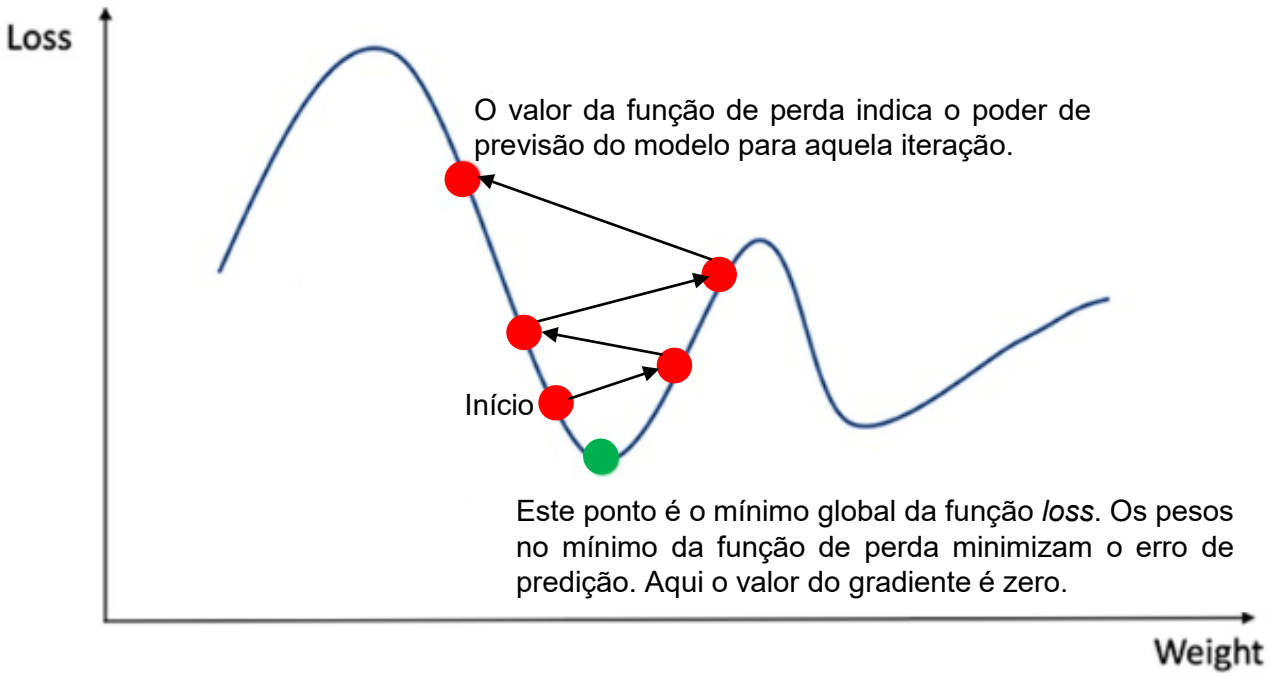




Homer Simpson implementa sua versão do gradiente descendente.

Redes Neurais

Por outro lado, numa situação onde a taxa de aprendizado é muito alta, podemos atravessar o vale e acabar do outro lado, possivelmente em um lugar mais alto do que estava antes. Nesse caso, o algoritmo pode divergir, com valores cada vez maiores, e não encontrar o ponto de mínimo global, como ilustrado na figura abaixo.

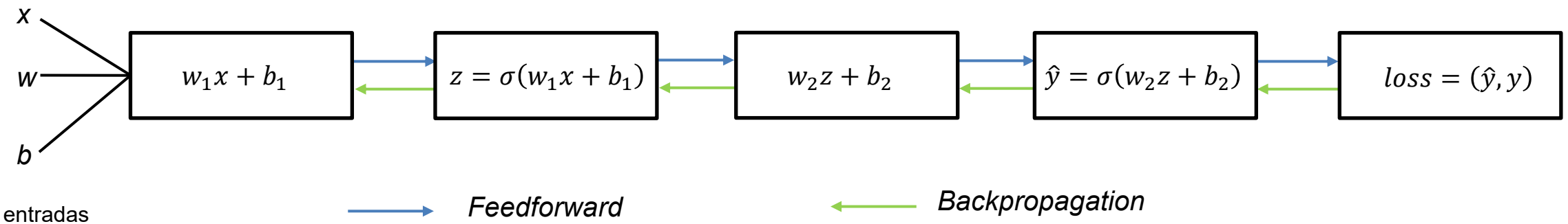


Redes Neurais

Uma vez que você tenha o vetor gradiente, que aponta para cima, basta ir na direção oposta para descer. Isso significa subtrair $\nabla_w MSE(w)$ de w. É aqui que a taxa de aprendizado η entra em cena: multiplicar o vetor gradiente por η para determinar o tamanho da etapa de declive:

$$w(próxima\ iteração) = w - \eta \nabla_w MSE(w)$$

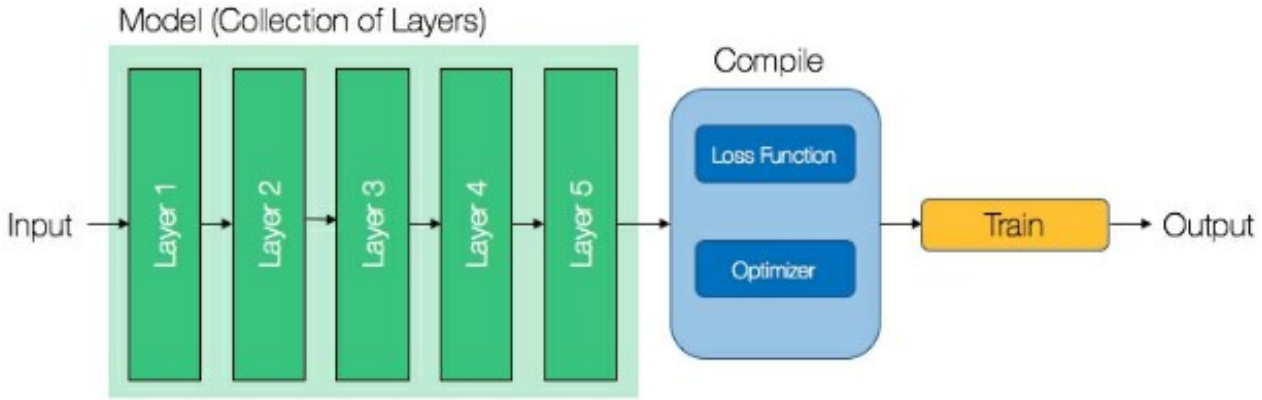
O algoritmo da rede neural prevê a repetição do processo abaixo, cada ciclo é chamado de iteração ou época.



x: entradas
w: pesos
b: vieses (bias)
y: valores reais
$\hat{y}$: valores previstos pelo modelo
σ: função de ativação
loss(ŷ,y): função de perda
MSE: erro quadrático médio
$\nabla_w MSE(w)$: gradiente do *MSE*
η: taxa de aprendizado

Redes Neurais com Keras

Usaremos as bibliotecas TensorFlow e Keras para implementar nossas redes neurais. Especificamente usaremos a biblioteca Keras que é uma API *Application Programming Interface* (Interface de Programação de Aplicação) que roda o TensorFlow. Com Keras implementamos redes neurais de forma intuitiva e simples. Os blocos fundamentais da biblioteca Keras são as camadas de neurônios. Nós podemos empilhar essas camadas linearmente e criarmos nossos modelos. A função de perda nos fornece a métrica que usaremos para treinar nosso modelo usando o *optimizer* (**otimizador**). O diagrama esquemático abaixo ilustra a relação entre os blocos constituintes da biblioteca Keras.



Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 93). Packt Publishing. Edição do Kindle.

Redes Neurais com Keras

Podemos pensar nas camadas em Keras como átomos, porque estas são as menores unidades de uma rede neural. Cada camada tem uma entrada e executa uma função matemática, em seguida gera a saída para a camada seguinte. As camadas possíveis em Keras incluem camadas densa, de ativação e *dropout*. Nosso foco aqui é na camada densa. Uma camada densa também é conhecida como camada totalmente conectada, visto que ela tem todas as entradas da função matemática que ela implementa. Uma camada densa implementa a seguinte função já vista.

$$\hat{y} = \sigma(wx + b)$$

x: entradas
w: pesos
b: vieses (bias)
y: valores reais
$\hat{y}$: valores previstos pelo modelo
σ: função de ativação
loss(ŷ,y): função de perda
MSE: erro quadrático médio
$\nabla_w MSE(w)$: gradiente do *MSE*
η: taxa de aprendizado

Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 95). Packt Publishing. Edição do Kindle.

Redes Neurais com Keras

Se as camadas são os átomos, os modelos são as moléculas em Keras. Um modelo é simplesmente uma coleção de camadas e a mais fundamental em Keras é o modelo sequencial. Um modelo sequencial nos permite empacotar camadas, onde uma camada isolada está conectada a somente uma outra camada. Essa flexibilidade nos permite desenhar arquiteturas de redes sem nos preocuparmos com a matemática atrás de seu funcionamento. Escolhida a arquitetura da rede neural, precisamos definir o processo de treinamento, o que é realizado pelo método *compile* em Keras. Esse método tem vários argumentos, mas os mais importantes são o otimizador e a função de perda.

$$\hat{y} = \sigma(wx + b)$$

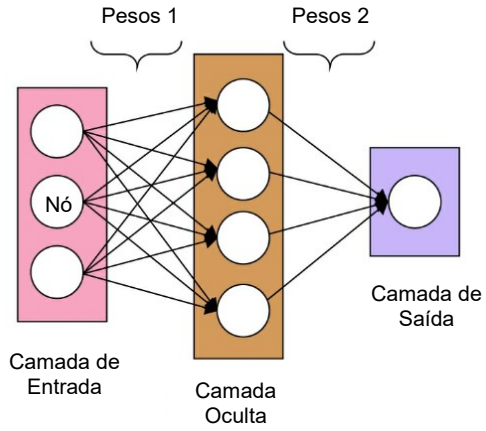
x: entradas
w: pesos
b: vieses (bias)
y: valores reais
$\hat{y}$: valores previstos pelo modelo
σ: função de ativação
loss(ŷ,y): função de perda
MSE: erro quadrático médio
$\nabla_w MSE(w)$: gradiente do *MSE*
η: taxa de aprendizado

Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 96). Packt Publishing. Edição do Kindle.

Redes Neurais com Keras

Veremos um código em Python que usa a biblioteca Keras para implementar uma rede neural para que lê três entradas x_1, x_2 e x_3 e prevê a saída (y), como indicada na tabela abaixo. Implementaremos uma rede neural com duas camadas (arquitetura ilustrada à direita) que usa a função de perda MSE e o otimizador é o método de gradiente descendente.

x_1	x_2	x_3	y
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	0



Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 65). Packt Publishing. Edição do Kindle.

Redes Neurais com Keras

Inicie o Jupyter e carregue o código *nn_keras.ipynb* disponível na pasta da aula de hoje. O código está mostrado a seguir. Iremos discutir os principais comandos da biblioteca Keras que possibilitam a implementação da nossa rede neural. Para facilitar a visualização, deixaremos o diagrama esquemático da arquitetura da rede implementada junto com o código.

Home | Home | nn_keras

http://localhost:8888/notebooks/nn_keras.ipynb

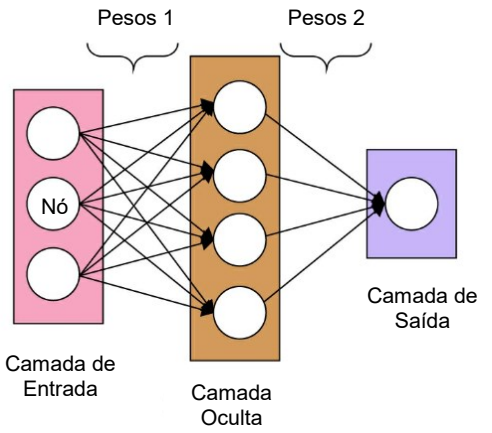
jupyter nn_keras Last Checkpoint: 12 seconds ago

File Edit View Run Kernel Settings Help

Trusted

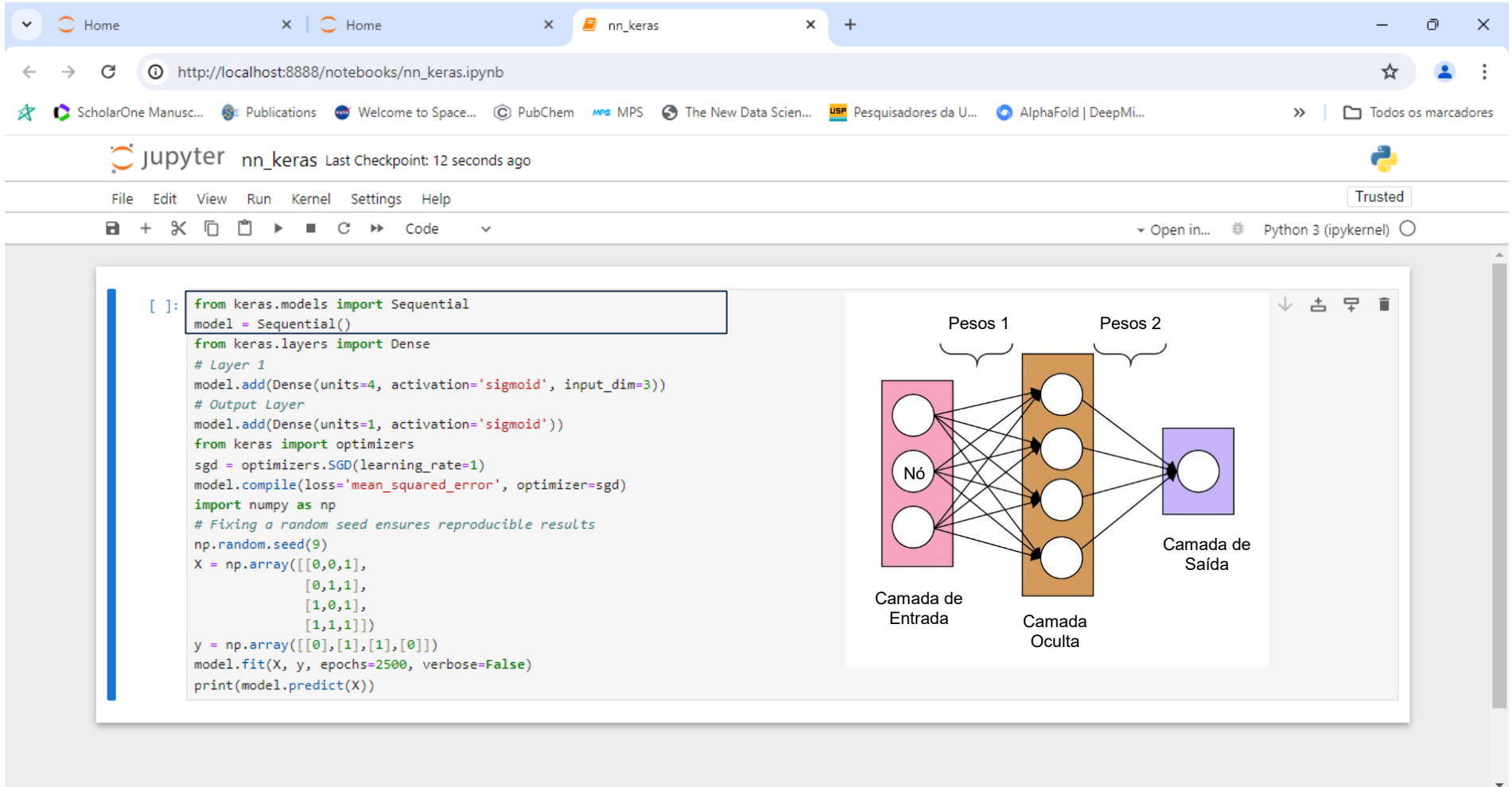
Code | Open in... | Python 3 (ipykernel)

```
[ ]: from keras.models import Sequential
model = Sequential()
from keras.layers import Dense
# Layer 1
model.add(Dense(units=4, activation='sigmoid', input_dim=3))
# Output Layer
model.add(Dense(units=1, activation='sigmoid'))
from keras import optimizers
sgd = optimizers.SGD(learning_rate=1)
model.compile(loss='mean_squared_error', optimizer=sgd)
import numpy as np
# Fixing a random seed ensures reproducible results
np.random.seed(9)
X = np.array([[0,0,1],
              [0,1,1],
              [1,0,1],
              [1,1,1]])
y = np.array([[0],[1],[1],[0]])
model.fit(X, y, epochs=2500, verbose=False)
print(model.predict(X))
```



Redes Neurais com Keras

A primeira linha importa o *Sequential* da biblioteca Keras. Para construir uma coleção de camadas, inicialmente declaramos um modelo sequencial. Criamos um modelo vazio com a linha de comando *model = Sequential().*



Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 100). Packt Publishing. Edição do Kindle.

Redes Neurais com Keras

Agora a adição de camadas é tão simples como encaixar blocos de Lego. Na linha 3 importamos o *Dense*. Tomando a arquitetura desenhada abaixo, nós adicionamos as camadas da esquerda para direita. Para inserir camadas, usamos o comando *model.add()*. Precisamos informar o número de features (*input_dim*). Outro parâmetro é o número de unidades (*units*), que é o número de neurônios da camada (nos).

Home | Home | nn_keras

http://localhost:8888/notebooks/nn_keras.ipynb

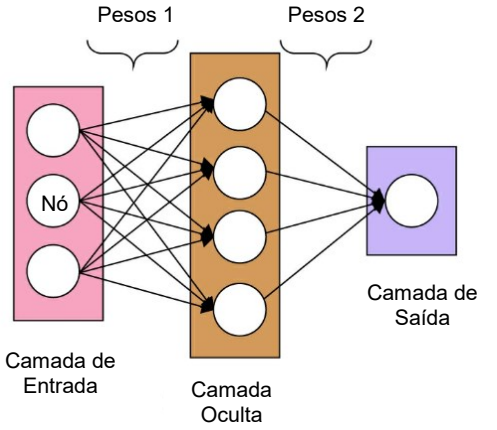
jupyter nn_keras Last Checkpoint: 12 seconds ago

File Edit View Run Kernel Settings Help

Trusted

Code | Open in... | Python 3 (ipykernel)

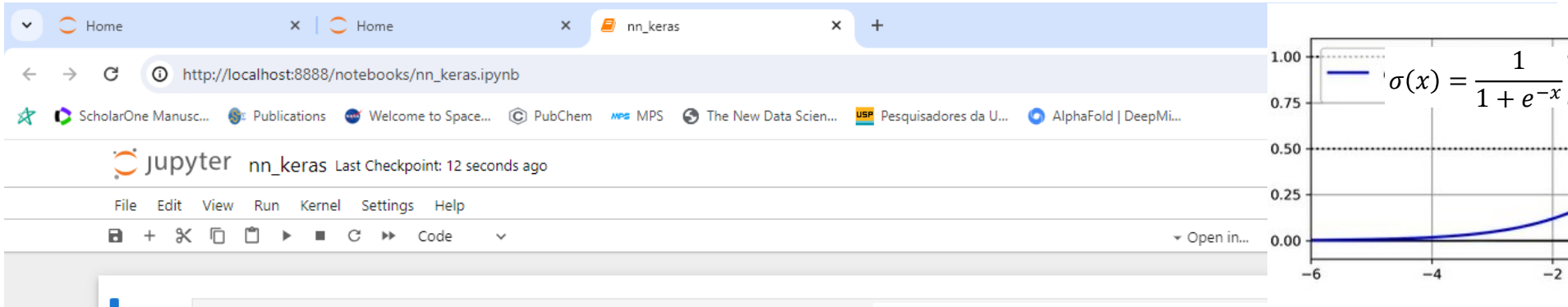
```
[ ]: from keras.models import Sequential
model = Sequential()
from keras.layers import Dense
# Layer 1
model.add(Dense(units=4, activation='sigmoid', input_dim=3))
# Output Layer
model.add(Dense(units=1, activation='sigmoid'))
from keras import optimizers
sgd = optimizers.SGD(learning_rate=1)
model.compile(loss='mean_squared_error', optimizer=sgd)
import numpy as np
# Fixing a random seed ensures reproducible results
np.random.seed(9)
X = np.array([[0,0,1],
              [0,1,1],
              [1,0,1],
              [1,1,1]])
y = np.array([[0],[1],[1],[0]])
model.fit(X, y, epochs=2500, verbose=False)
print(model.predict(X))
```



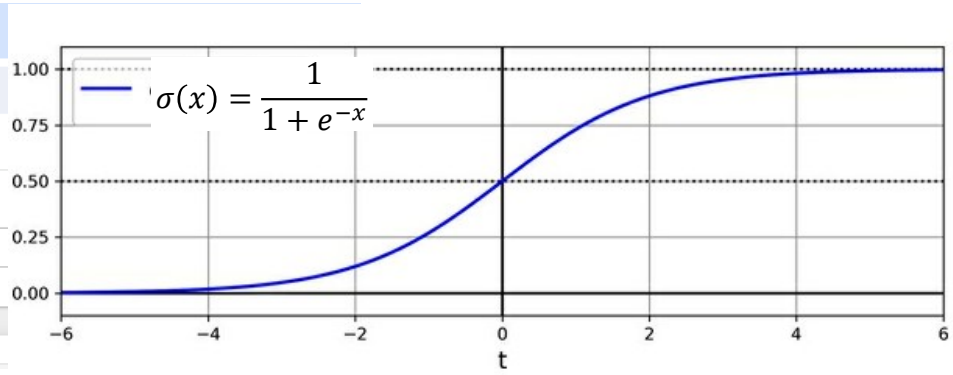
Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 100). Packt Publishing. Edição do Kindle.

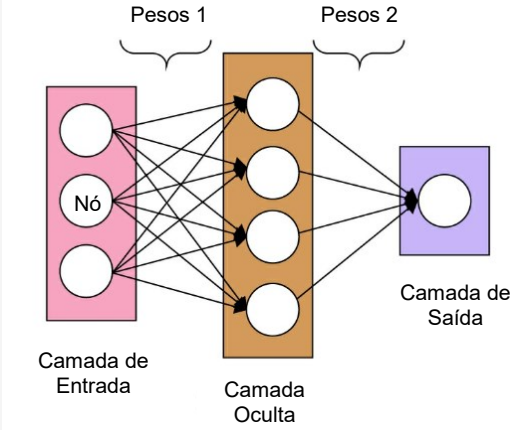
Redes Neurais com Keras

Além do número de nos (*units=4*) e de features (*input_dim=3*), precisamos definir a função de ativação. Usaremos a função sigmoide (*activation='sigmoid'*). A função sigmoide $\sigma(x)$ está definida abaixo. Destacamos que aumentar o número de unidades amplia a complexidade do modelo, o que significa mais pesos a serem treinados.




```
[ ]: from keras.models import Sequential
model = Sequential()
from keras.layers import Dense
# Layer 1
model.add(Dense(units=4, activation='sigmoid', input_dim=3))
# Output Layer
model.add(Dense(units=1, activation='sigmoid'))
from keras import optimizers
sgd = optimizers.SGD(learning_rate=1)
model.compile(loss='mean_squared_error', optimizer=sgd)
import numpy as np
# Fixing a random seed ensures reproducible results
np.random.seed(9)
X = np.array([[0,0,1],
              [0,1,1],
              [1,0,1],
              [1,1,1]])
y = np.array([[0],[1],[1],[0]])
model.fit(X, y, epochs=2500, verbose=False)
print(model.predict(X))
```

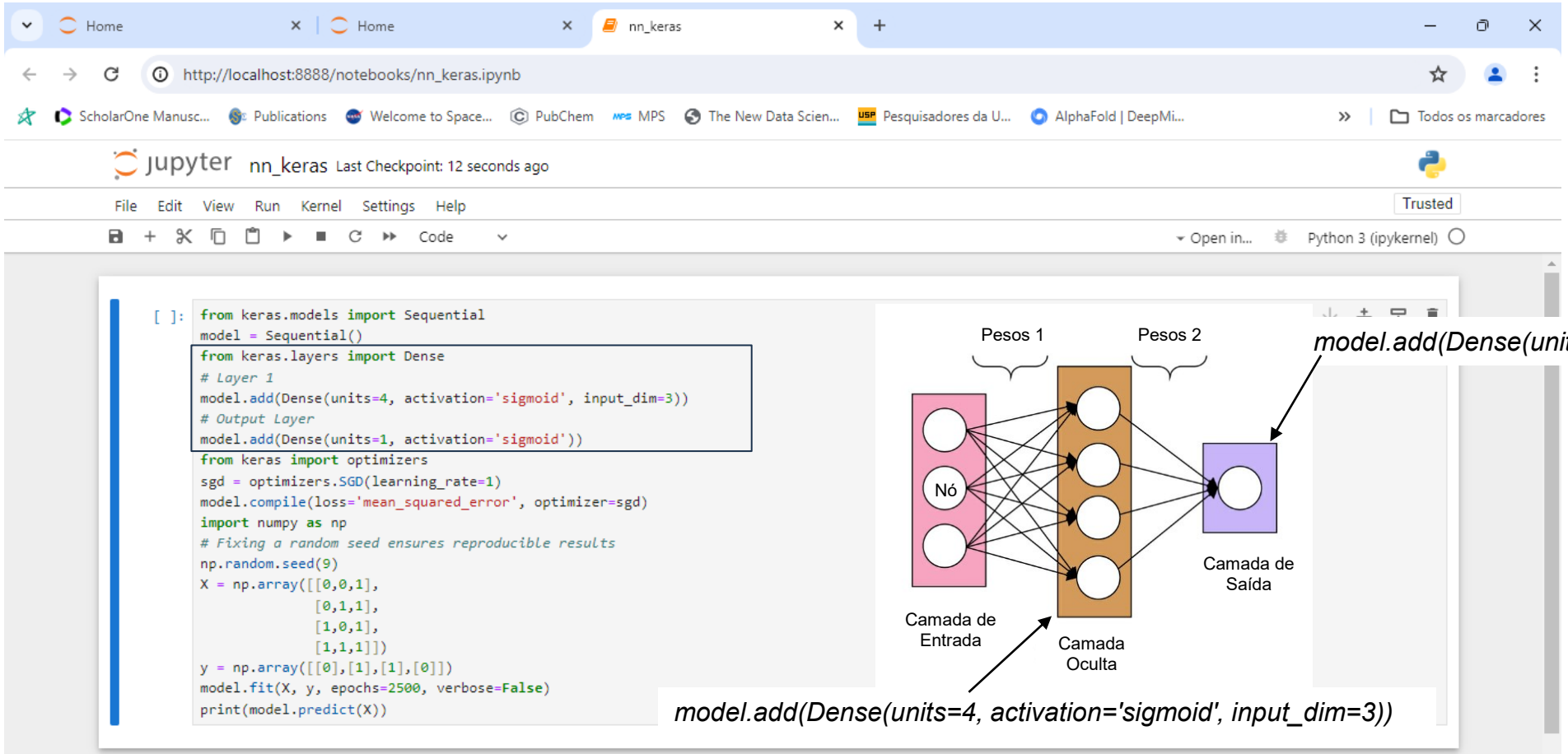




Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 100). Packt Publishing. Edição do Kindle.

Redes Neurais com Keras

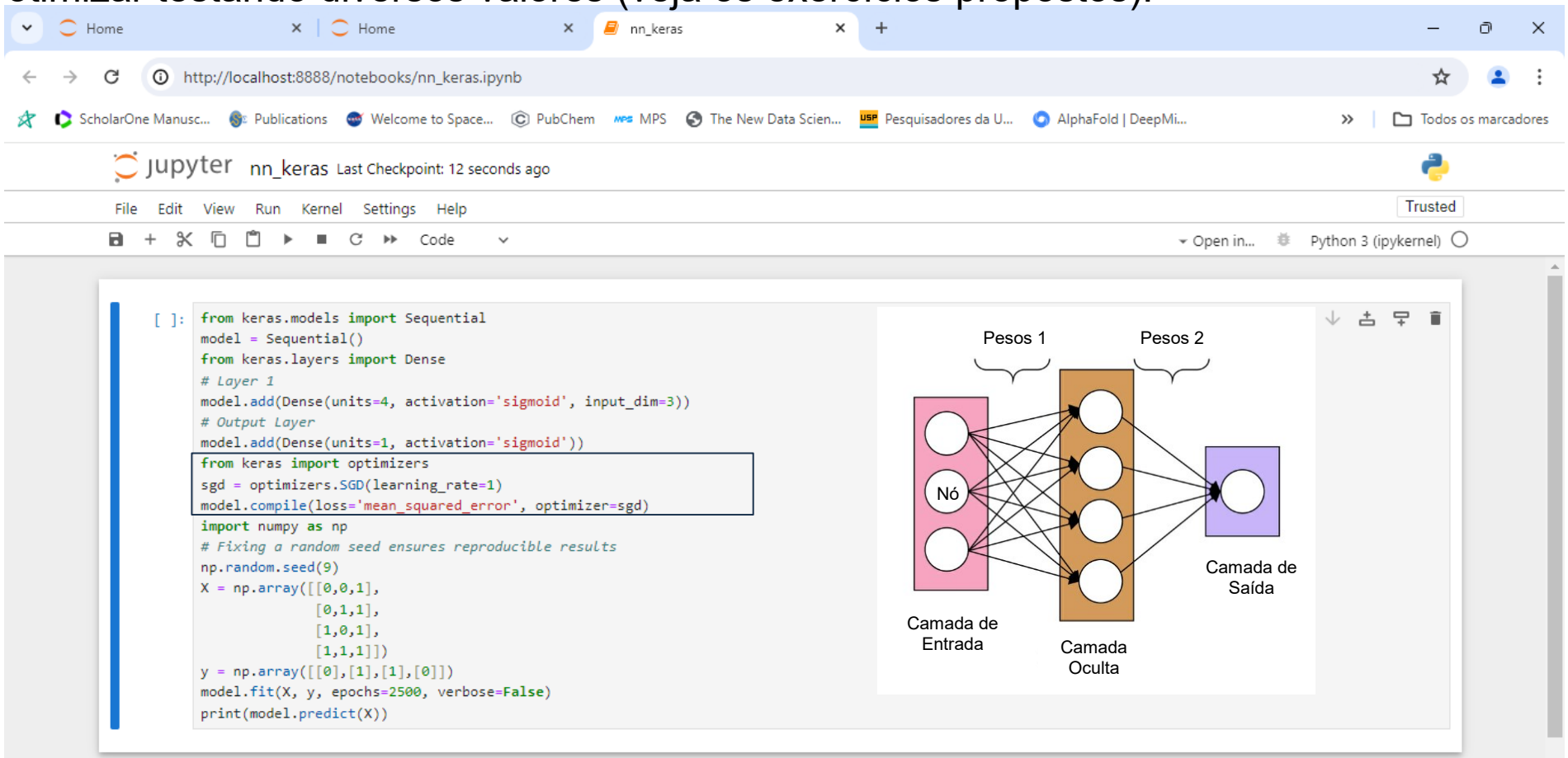
Veja que podemos traçar uma equivalência entre o diagrama esquemático da arquitetura da rede com o código desenvolvido com a biblioteca Keras. A camada entrada não é implementada, a nossa primeira camada é a camada oculta e em seguida temos a camada de saída. Quando dizemos que a rede neural tem duas camadas, estamos nos referindo a camada oculta e a camada de saída.



Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 100). Packt Publishing. Edição do Kindle.

Redes Neurais com Keras

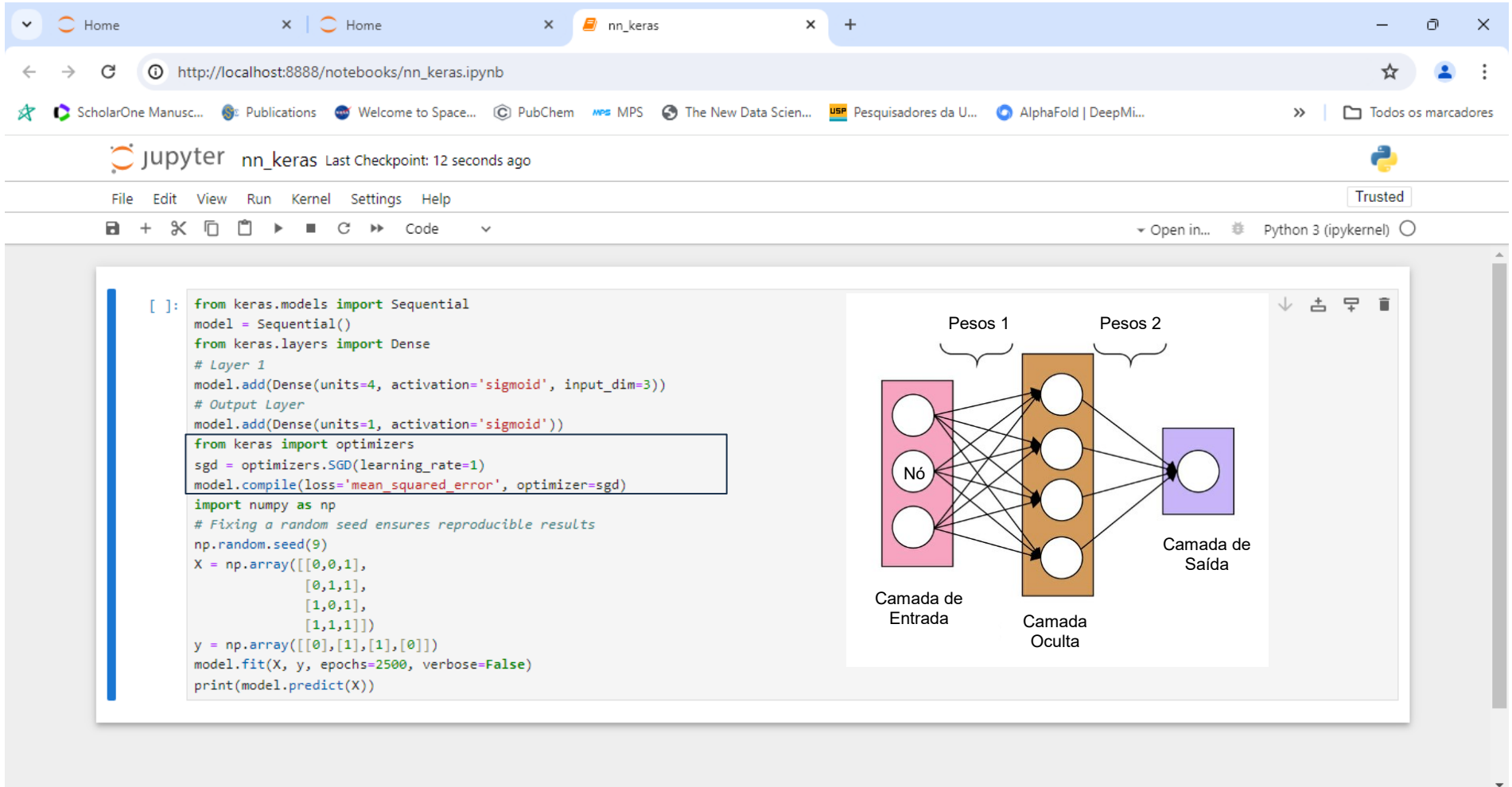
Uma vez satisfeitos com a arquitetura do modelo, realizaremos a compilação e treinamento da rede. Inicialmente importamos os otimizadores da biblioteca Keras. Nós usaremos o otimizador gradiente descendente com uma taxa de aprendizado de 1 (*learning_rate=1*). Como destacado, essa taxa é um hiperparâmetro que podemos otimizar testando diversos valores (veja os exercícios propostos).



Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 100). Packt Publishing. Edição do Kindle.

Redes Neurais com Keras

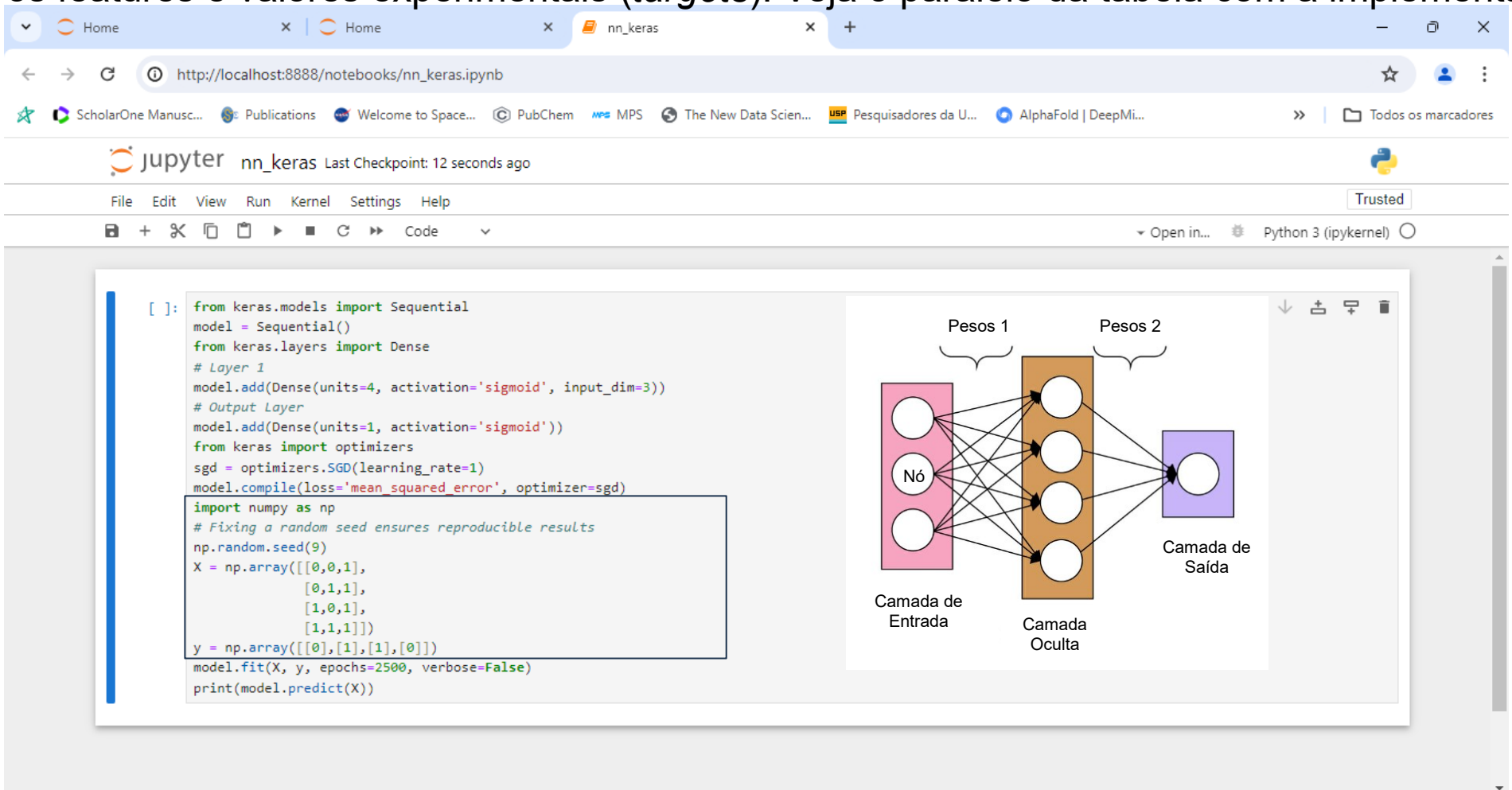
Depois definimos a função de perda com o comando *compile*, mostrado na última linha em destaque. A linha de comando *model.compile(loss='mean_squared_error', optimizer=sgd)* indica que a função de perda é MSE (*mean_squared_error*) e o otimizador é o gradiente descendente (*sgd*) (*stochastic gradient descent*).



Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 100). Packt Publishing. Edição do Kindle.

Redes Neurais com Keras

Na sequência iremos definir as entradas e saídas. Usaremos os recursos da biblioteca NumPy. Inicialmente importamos a biblioteca NumPy. Depois usamos a linha de comando *np.random.seed(9)* que define a semente aleatória. Esse recurso garante a reprodutibilidade dos resultados. Depois definimos os arrays *X* e *y*, que trazem os features e valores experimentais (*targets*). Veja o paralelo da tabela com a implementação no NumPy.

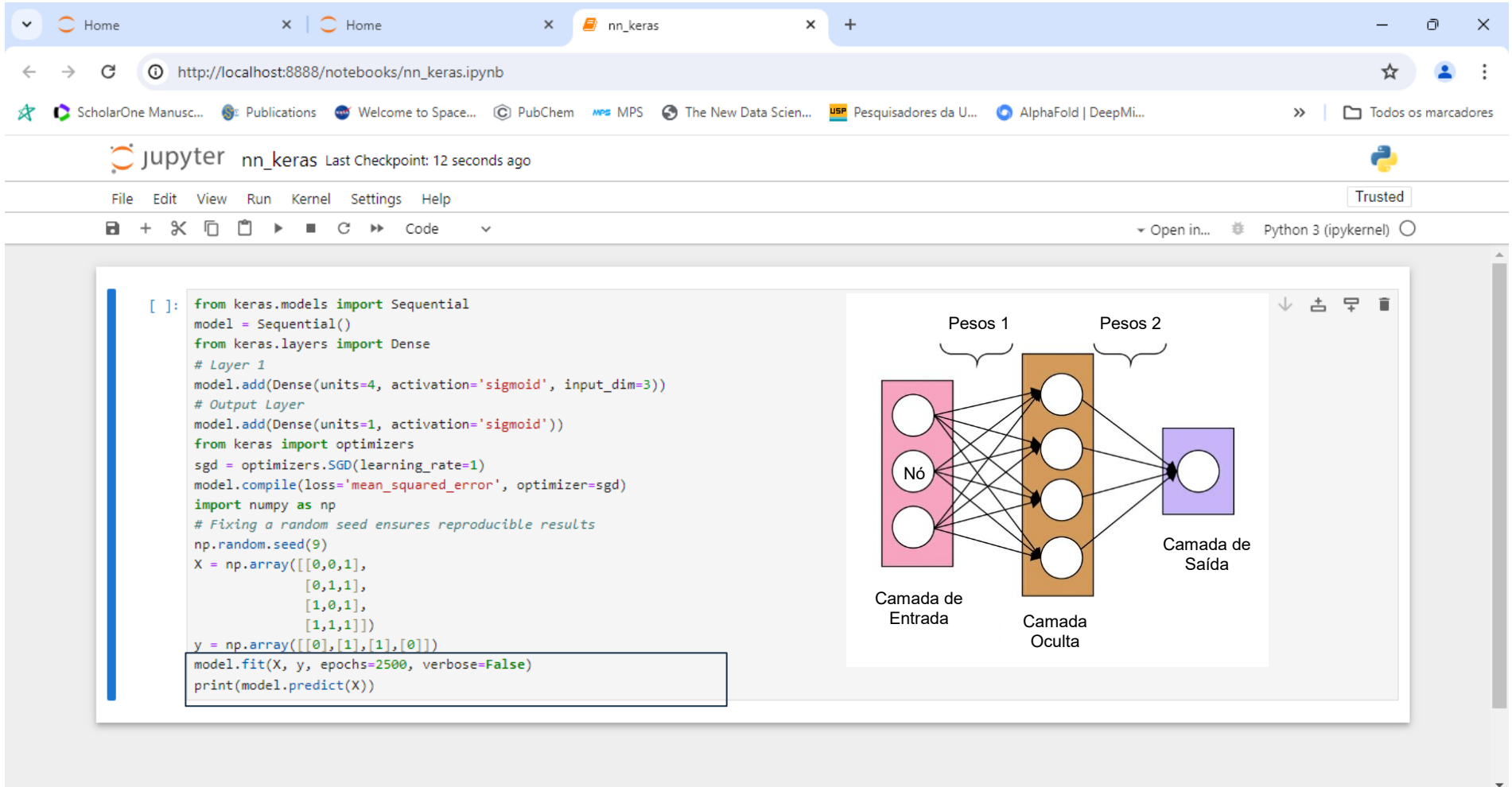


x_1	x_2	x_3	y
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	0

Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 100). Packt Publishing. Edição do Kindle.

Redes Neurais com Keras

As duas últimas linhas geram o modelo e mostram o resultados. A linha de comando *model.fit(X, y, epochs=2500, verbose=False)* cria o modelo. Veja que usamos como entrada os arrays *X* e *y*. O *epochs* define o número de iterações (épocas).

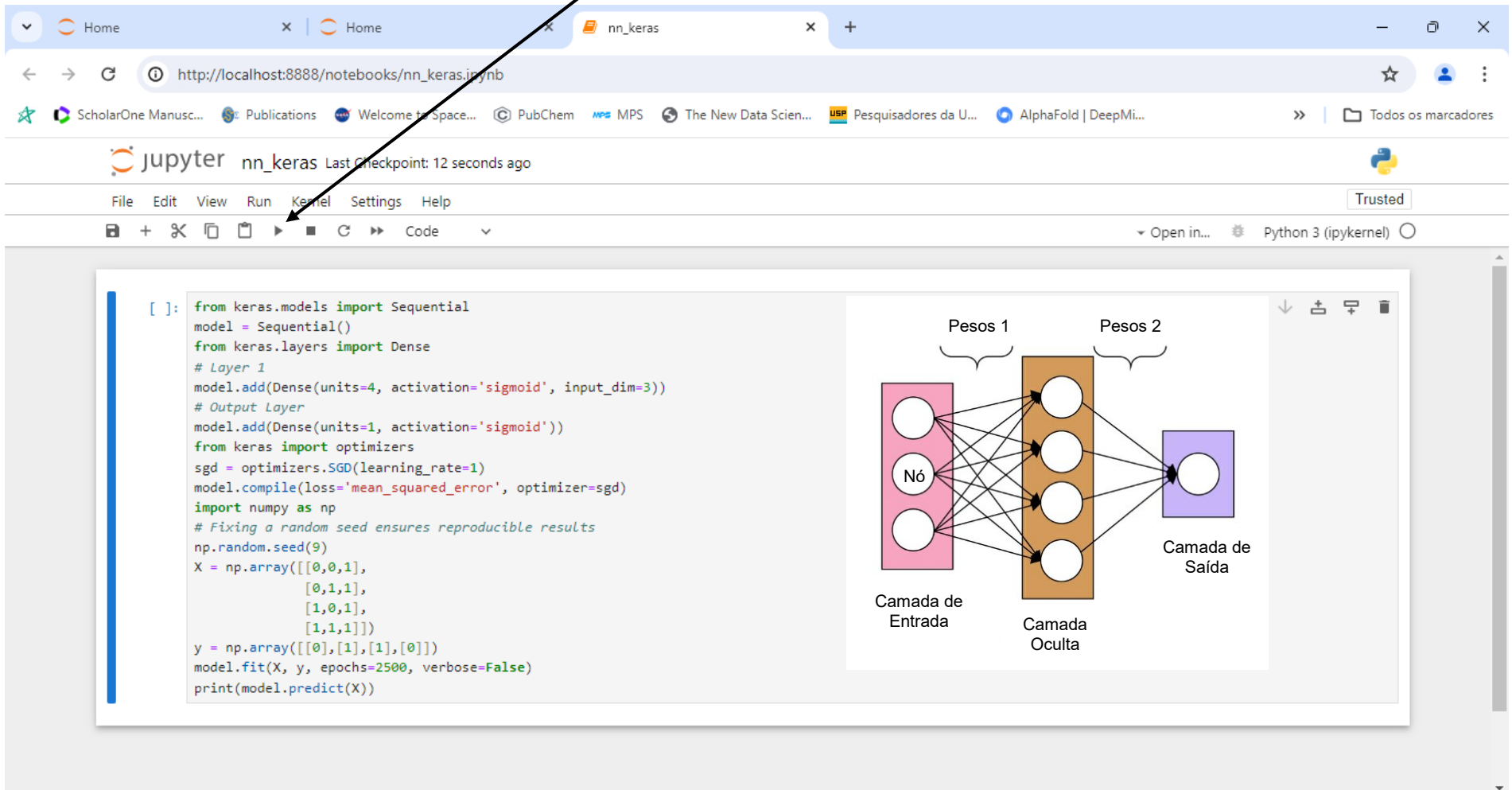


x_1	x_2	x_3	y
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	0

Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 100). Packt Publishing. Edição do Kindle.

Redes Neurais com Keras

Para executar o código clique no botão indicado.

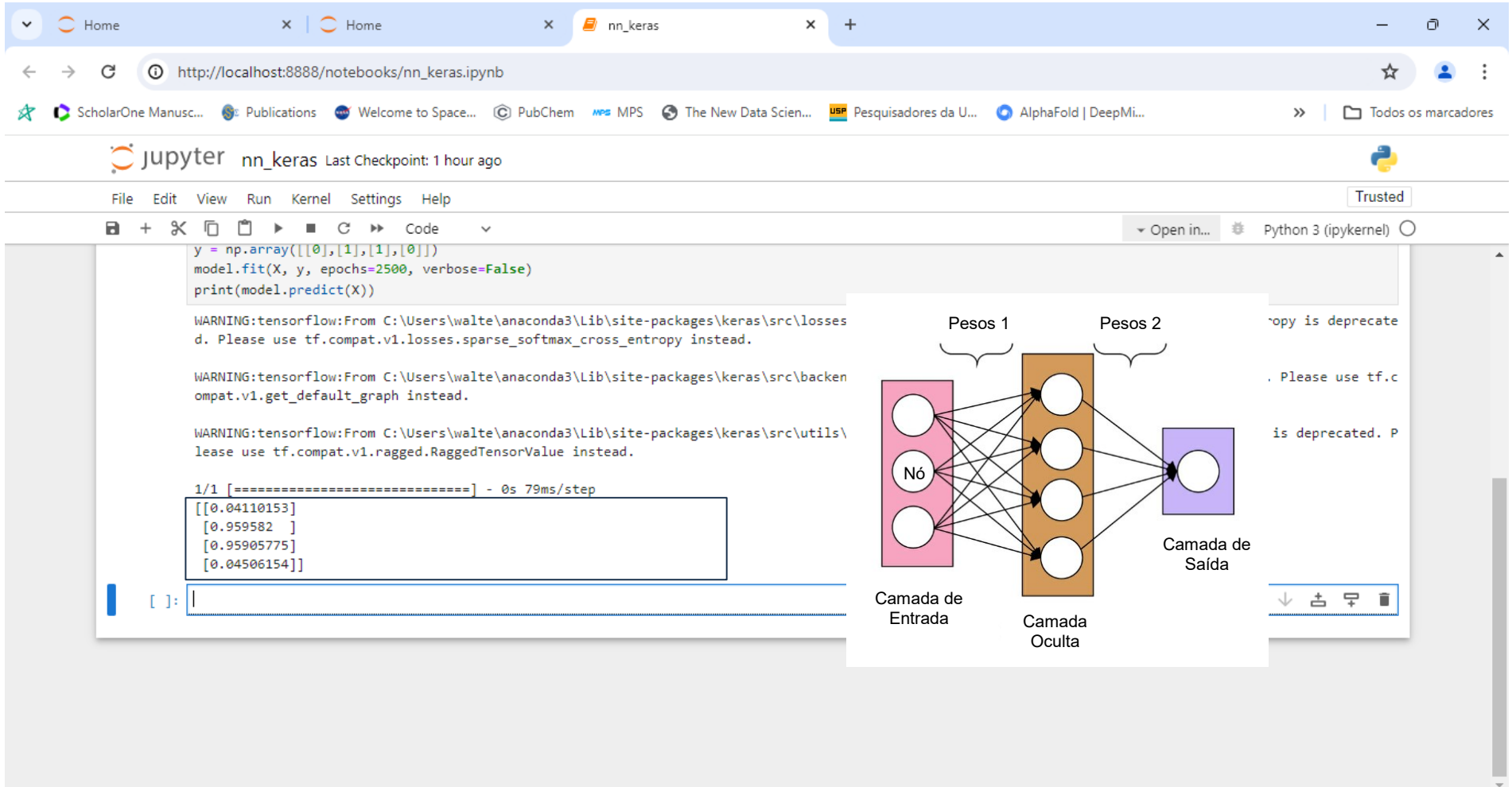


x_1	x_2	x_3	y
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	0

Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 100). Packt Publishing. Edição do Kindle.

Redes Neurais com Keras

Abaixo temos o resultado gerado. Veja que os valores previstos estão próximos aos experimentais (coluna *y*) da tabela abaixo.



x_1	x_2	x_3	y
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	0

Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 100). Packt Publishing. Edição do Kindle.

Modelo para Previsão de Diabetes

Diabetes é um dos maiores problemas de saúde pública da atualidade. Devido ao sedentarismo e ao consumo de alimentos ultraprocessados o percentual de diabetes do tipo 2 tem crescido em países desenvolvidos, bem como em países em desenvolvimento. Estima-se que 8,5 % da população adulta desenvolva diabetes (Loy, 2019). A prevenção e a detecção precoce do diabetes do tipo 2 é fundamental para combater o aumento de casos. Especificamente na detecção, as abordagens de aprendizado de máquina podem contribuir. Neste exemplo iremos desenvolver um modelo para previsão de diabetes do tipo 2 a partir de oito features. O foco é num conjunto de dados disponível sobre a incidência de diabetes do tipo 2 numa população de indígenas da tribo Pima dos Estados Unidos. Esse grupo apresenta a maior incidência de diabetes registrada mundialmente. Os indígenas dessa tribo apresentam uma mutação ligada à resistência a grandes períodos de fome, mas que aumenta a tendência para a diabetes se a população estiver numa dieta de alimentos ultraprocessados. O conjunto de dados está disponível no Kaggle: *Pima Indians Diabetes Database*. O código discutido aqui está baseado no capítulo 2 do livro de Loy, 2019.

Modelo para Previsão de Diabetes

O conjunto de dados (*Pima Indians Diabetes Database*) consiste de medidas de diagnóstico coletadas de uma amostra da população feminina dos indígenas da tribo Pima. Para cada medida temos um rótulo indicando se o paciente desenvolveu diabetes até cinco anos após a medida inicial. Abaixo temos as primeiras linhas do arquivo *diabetes.csv*.

Molegro Data Modeller

File Edit Preparation Modelling Visualization Window Modules Help

Selection | Descriptors: All | Coloring: Default | Search

Workspace Explorer

Items
- Workspace: Unnamed
  - Datasets [1]
    - **diabetes**

Properties

Property	Value

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFun	Age	Outcome
1	6	148	72	35	0	33.6	0.627	50	1
2	1	85	66	29	0	26.6	0.351	31	0
3	8	183	64	0	0	23.3	0.672	32	1
4	1	89	66	23	94	28.1	0.167	21	0
5	0	137	40	35	168	43.1	2.288	33	1
6	5	116	74	0	0	25.6	0.201	30	0
7	3	78	50	32	88	31	0.248	26	1
8	10	115	0	0	0	35.3	0.134	29	0
9	2	197	70	45	543	30.5	0.158	53	1
10	8	125	96	0	0	0	0.232	54	1
11	4	110	92	0	0	37.6	0.191	30	0
12	10	168	74	0	0	38	0.537	34	1
13	10	139	80	0	0	27.1	1.441	57	0
14	1	189	60	23	846	30.1	0.398	59	1
15	5	166	72	19	175	25.8	0.587	51	1
16	7	100	0	0	0	30	0.484	32	1
17	0	118	84	47	230	45.8	0.551	31	1
18	7	107	74	0	0	29.6	0.254	31	1
19	1	103	30	38	83	43.3	0.183	33	0
20	1	115	70	30	96	34.6	0.529	32	1
21	3	126	88	41	235	39.3	0.704	27	0
22	8	99	84	0	0	35.4	0.388	50	0

Modelo para Previsão de Diabetes

Temos nove colunas no conjunto de dados. A última coluna indica o resultado do desenvolvimento do diabetes em até cinco anos após as medidas (rótulo 1). O zero indica que o paciente não desenvolveu o diabetes. Os features são oito no total.

Molegro Data Modeller

File Edit Preparation Modelling Visualization Window Modules Help

Selection | Descriptors: All | Coloring: Default | Search

Workspace Explorer

Items
- Workspace: Unnamed
  - Datasets [1]
    - **diabetes**

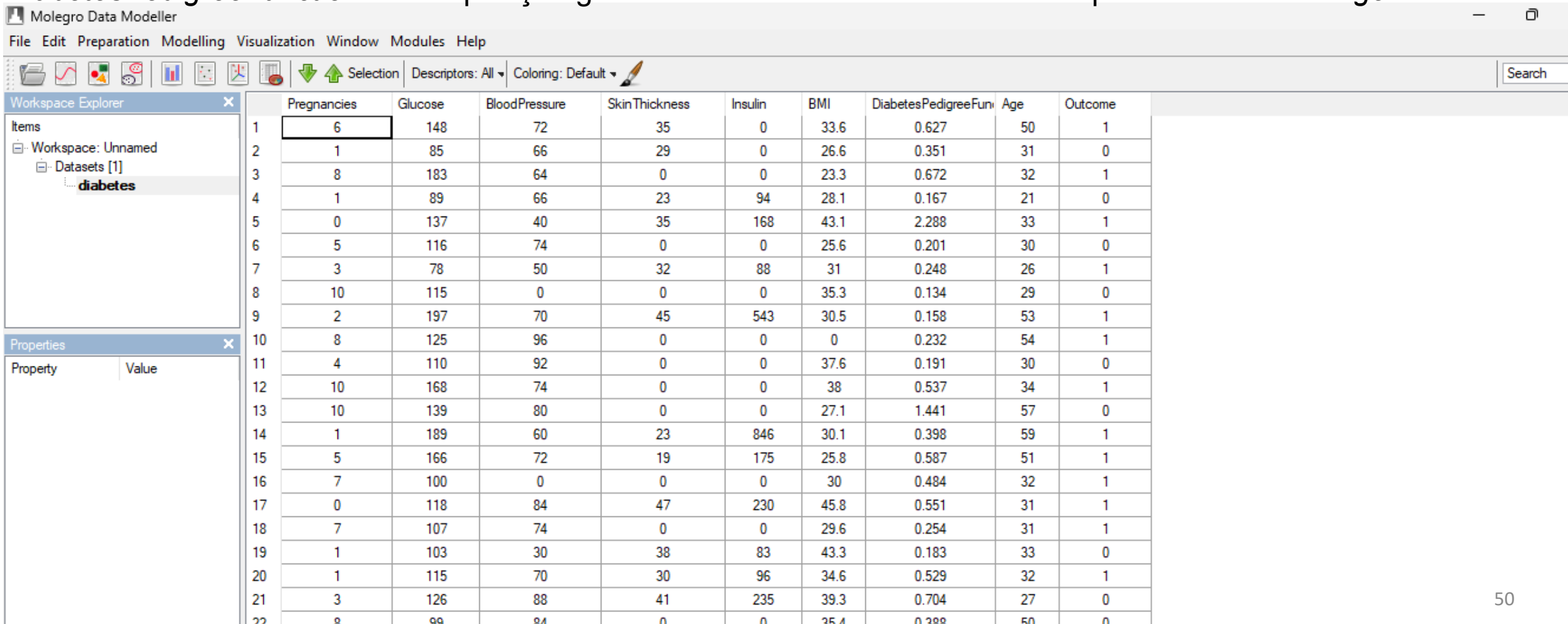
Properties

Property	Value

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFun	Age	Outcome
1	6	148	72	35	0	33.6	0.627	50	1
2	1	85	66	29	0	26.6	0.351	31	0
3	8	183	64	0	0	23.3	0.672	32	1
4	1	89	66	23	94	28.1	0.167	21	0
5	0	137	40	35	168	43.1	2.288	33	1
6	5	116	74	0	0	25.6	0.201	30	0
7	3	78	50	32	88	31	0.248	26	1
8	10	115	0	0	0	35.3	0.134	29	0
9	2	197	70	45	543	30.5	0.158	53	1
10	8	125	96	0	0	0	0.232	54	1
11	4	110	92	0	0	37.6	0.191	30	0
12	10	168	74	0	0	38	0.537	34	1
13	10	139	80	0	0	27.1	1.441	57	0
14	1	189	60	23	846	30.1	0.398	59	1
15	5	166	72	19	175	25.8	0.587	51	1
16	7	100	0	0	0	30	0.484	32	1
17	0	118	84	47	230	45.8	0.551	31	1
18	7	107	74	0	0	29.6	0.254	31	1
19	1	103	30	38	83	43.3	0.183	33	0
20	1	115	70	30	96	34.6	0.529	32	1
21	3	126	88	41	235	39.3	0.704	27	0
22	8	99	84	0	0	35.4	0.388	50	0

Modelo para Previsão de Diabetes

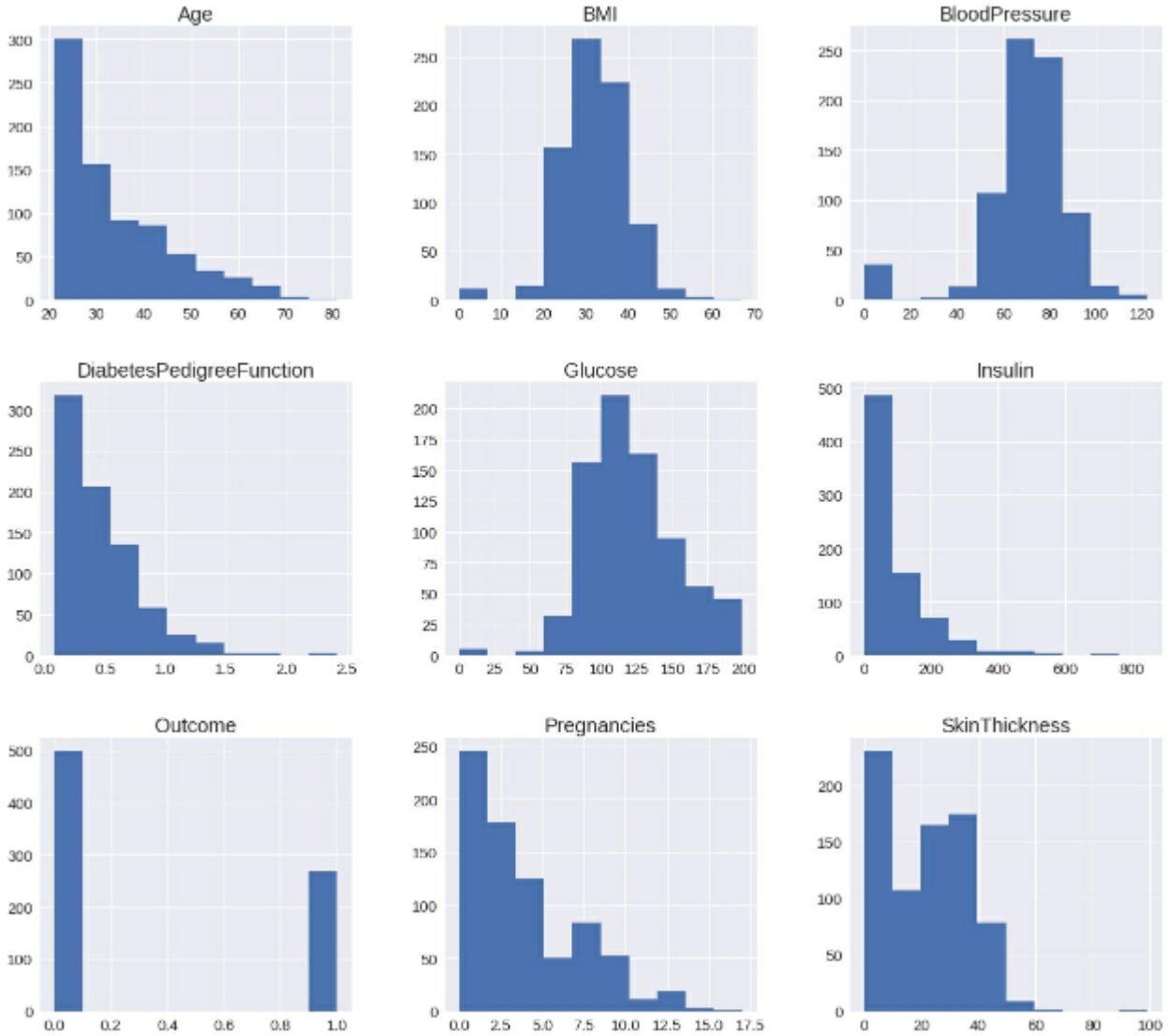
Pregnancies: Número de gravidez *Glucose*: Concentração da glicose no plasma sanguíneo
BloodPressure: Pressão diastólica *SkinThickness*: Espessura da dobra da pela medida a partir do tríceps
Insulin: Concentração de insulina *BMI*: Índice de massa corporal
DiabetesPedigreeFunction: Predisposição genética baseado no histórico familiar para diabetes *Age*: Idade



	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFun	Age	Outcome
1	6	148	72	35	0	33.6	0.627	50	1
2	1	85	66	29	0	26.6	0.351	31	0
3	8	183	64	0	0	23.3	0.672	32	1
4	1	89	66	23	94	28.1	0.167	21	0
5	0	137	40	35	168	43.1	2.288	33	1
6	5	116	74	0	0	25.6	0.201	30	0
7	3	78	50	32	88	31	0.248	26	1
8	10	115	0	0	0	35.3	0.134	29	0
9	2	197	70	45	543	30.5	0.158	53	1
10	8	125	96	0	0	0	0.232	54	1
11	4	110	92	0	0	37.6	0.191	30	0
12	10	168	74	0	0	38	0.537	34	1
13	10	139	80	0	0	27.1	1.441	57	0
14	1	189	60	23	846	30.1	0.398	59	1
15	5	166	72	19	175	25.8	0.587	51	1
16	7	100	0	0	0	30	0.484	32	1
17	0	118	84	47	230	45.8	0.551	31	1
18	7	107	74	0	0	29.6	0.254	31	1
19	1	103	30	38	83	43.3	0.183	33	0
20	1	115	70	30	96	34.6	0.529	32	1
21	3	126	88	41	235	39.3	0.704	27	0
22	8	99	84	0	0	35.4	0.388	50	0

Modelo para Previsão de Diabetes

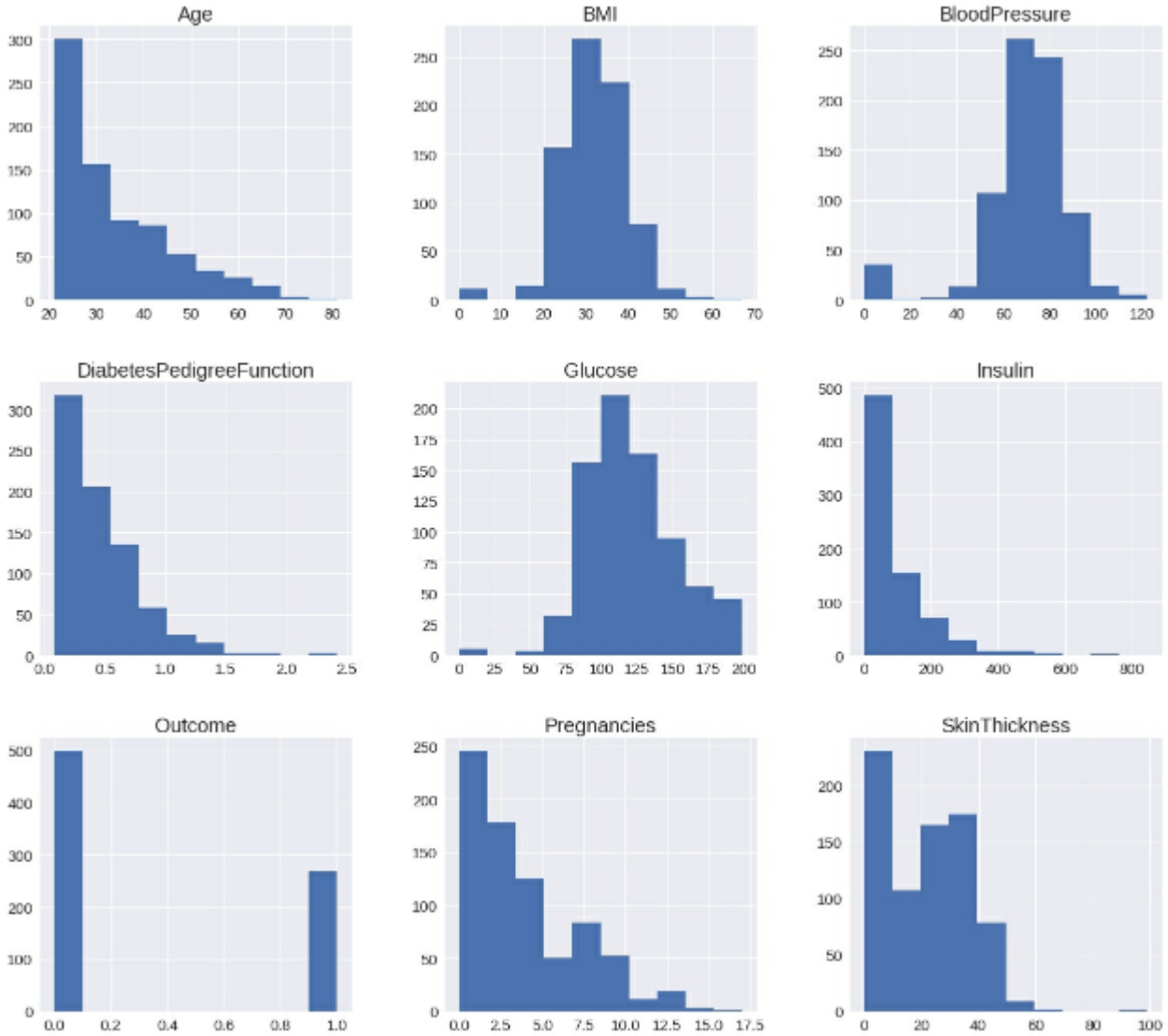
Ao lado temos os histogramas dos features e rótulo (*Outcome*) disponíveis no conjunto de dados (*Pima Indians Diabetes Database*). Algumas variáveis apresentam a distribuição na forma de curva sino, um comportamento esperado para dados de uma amostra da população. Outras variáveis não apresentam essa distribuição, como a idade. Vemos que a maior parte da amostra é relativamente jovem (entre 20 e 30 anos). Alguns resultados merecem tratamento posterior, como por exemplo BMI, pressão sanguínea (*BloodPressure*) e espessura da pele (*SkinThickness*). Esses features trazem zero para uma grandeza que obviamente não pode ser zero. Features que não podem ser zero indicam a necessidade de uma tratamento desses dados.



Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 119). Packt Publishing. Edição do Kindle.

Modelo para Previsão de Diabetes

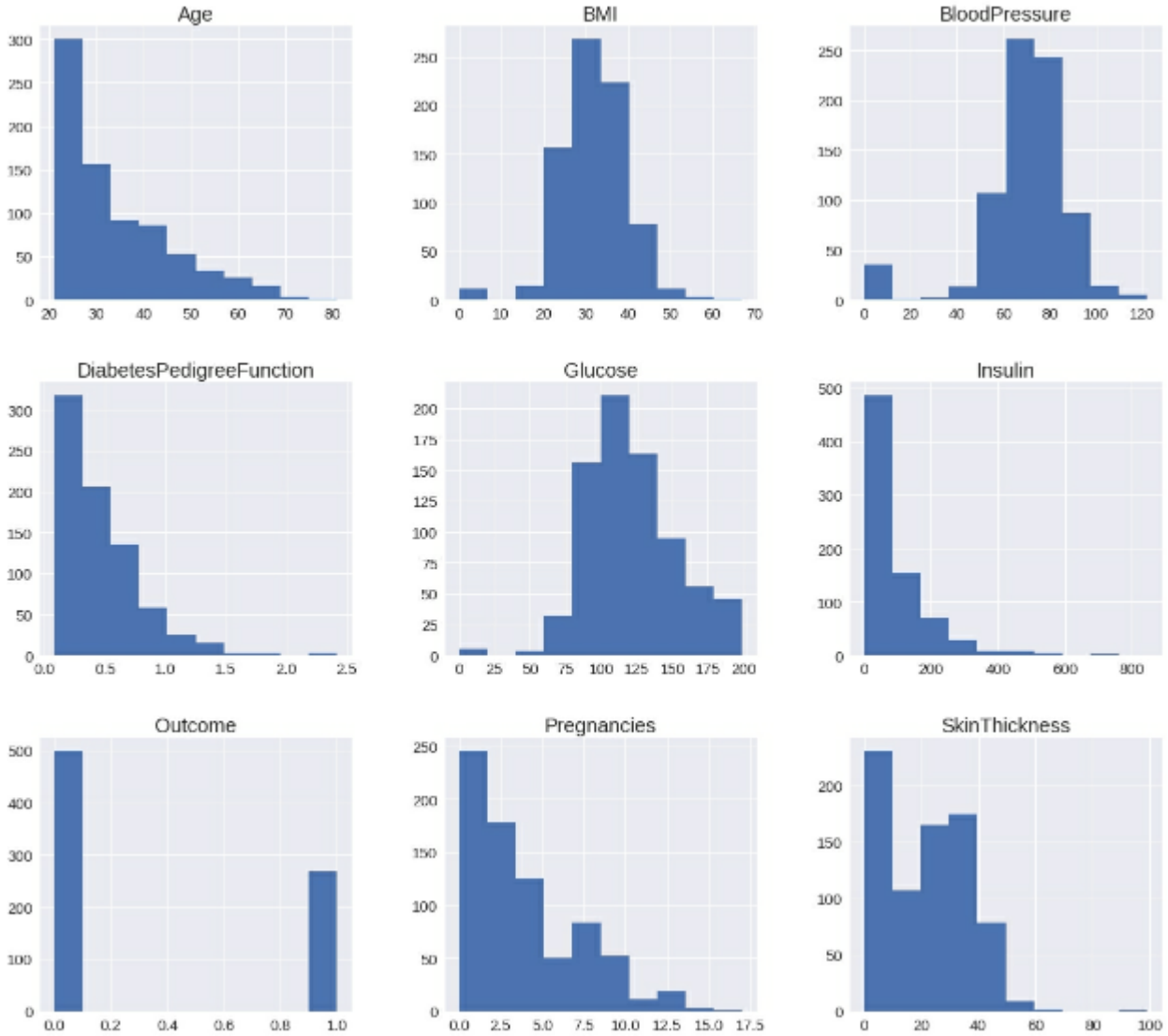
Outro aspecto de destaque é que as variáveis não estão em escala. Elas variam ordens de magnitude, por exemplo *DiabetesPedigreeFunction* vai de 0 até ~2,5 enquanto a variável insulina varia entre 0 e 800. Essa variação pode causar problemas na atribuição dos pesos no processo de treinamento da rede. Para evitar esse tipo de contratempo, escalonaremos os dados.



Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 119). Packt Publishing. Edição do Kindle.

Modelo para Previsão de Diabetes

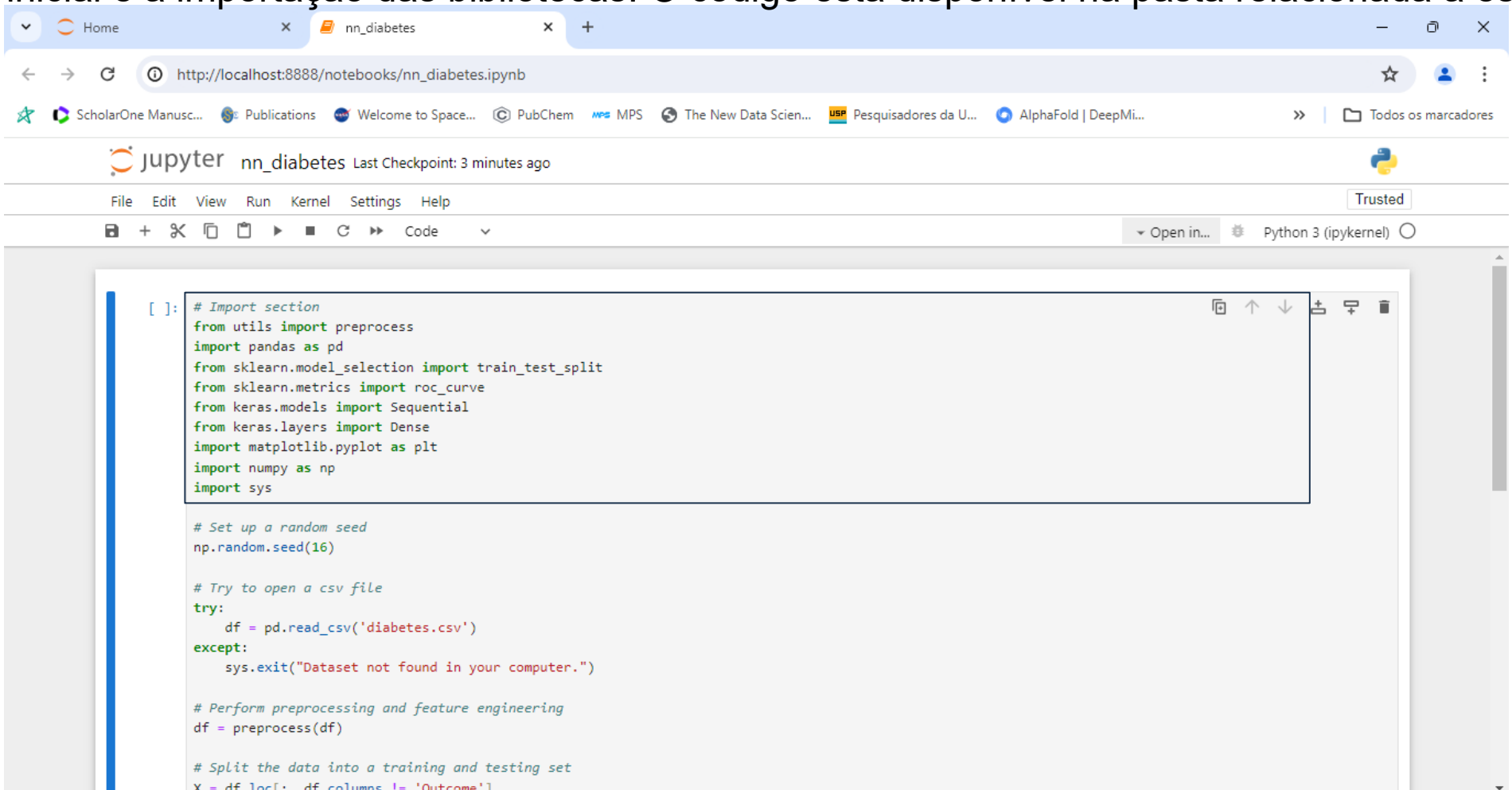
Na geração de modelos de aprendizado de máquina é comum realizamos uma etapa de preparação dos dados antes da geração dos modelos. Essa etapa é chamada de **preprocessamento dos dados**. O código (*nn_diabetes.ipynb*) com a implementação de uma rede neural para previsão de diabetes está descrito nos próximos slides. O referido código chama uma função (*utils.py*) que substitui os zeros dos features que não podem ser zero (e.g., BMI) pelo valor médio e usa a biblioteca Scikit-Learn (Pedregosa et al., 2011) para realizar o escalonamento dos features (classe *sklearn.preprocessing.StandardScaler*).



Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 119). Packt Publishing. Edição do Kindle.

Modelo para Previsão de Diabetes

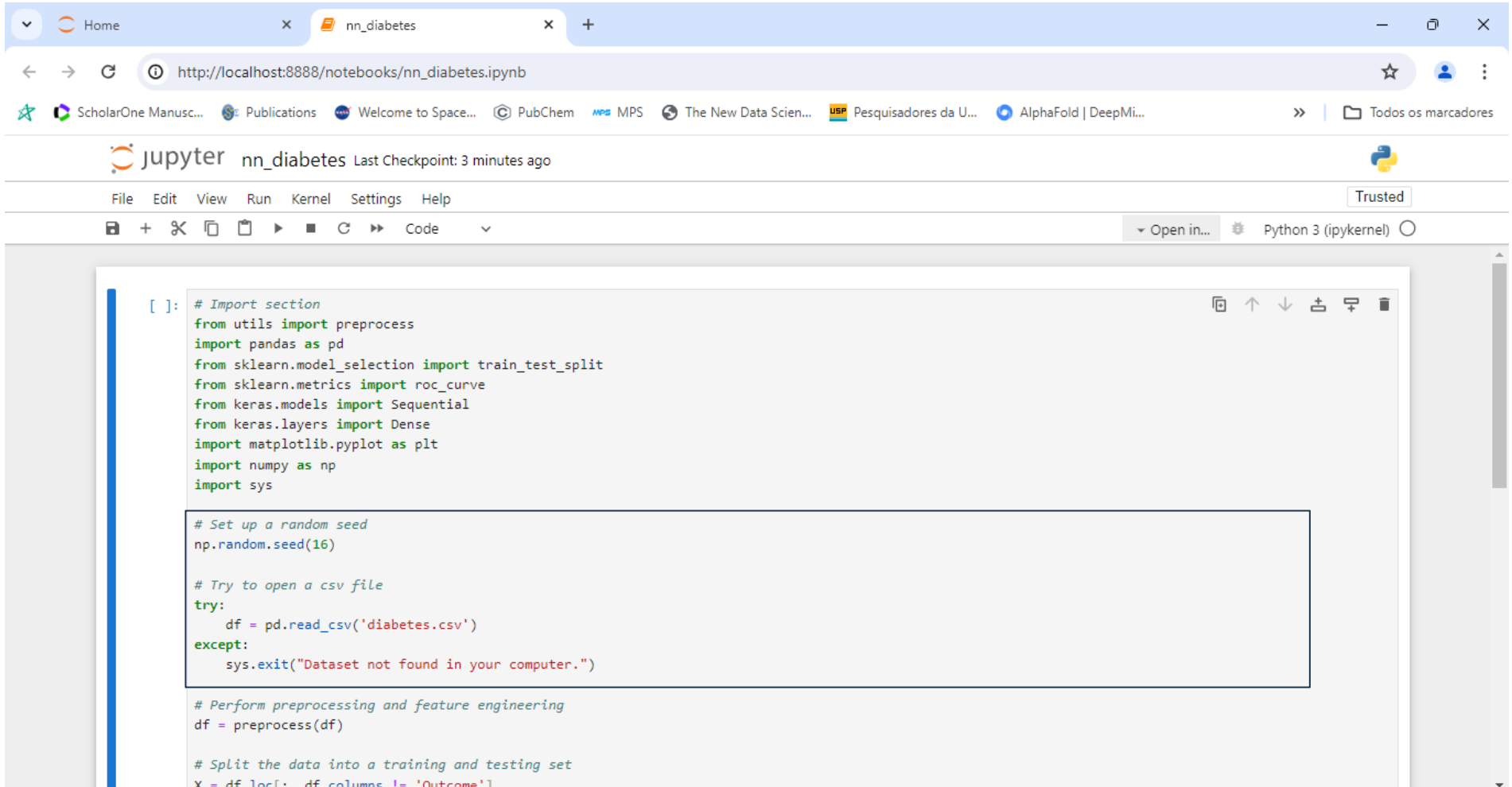
Abaixo temos as primeiras linhas do código *nn_diabetes.ipynb* para geração de modelo de previsão de diabetes. Este código é baseado no código *main.py* do capítulo 2 do livro Loy, 2019. Como o código não cabe numa única tela, vamos discuti-lo por setores e destacar a parte analisada com um retângulo sobreposto ao código. A parte inicial é a importação das bibliotecas. O código está disponível na pasta relacionada a esta aula.



Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 153). Packt Publishing. Edição do Kindle.

Modelo para Previsão de Diabetes

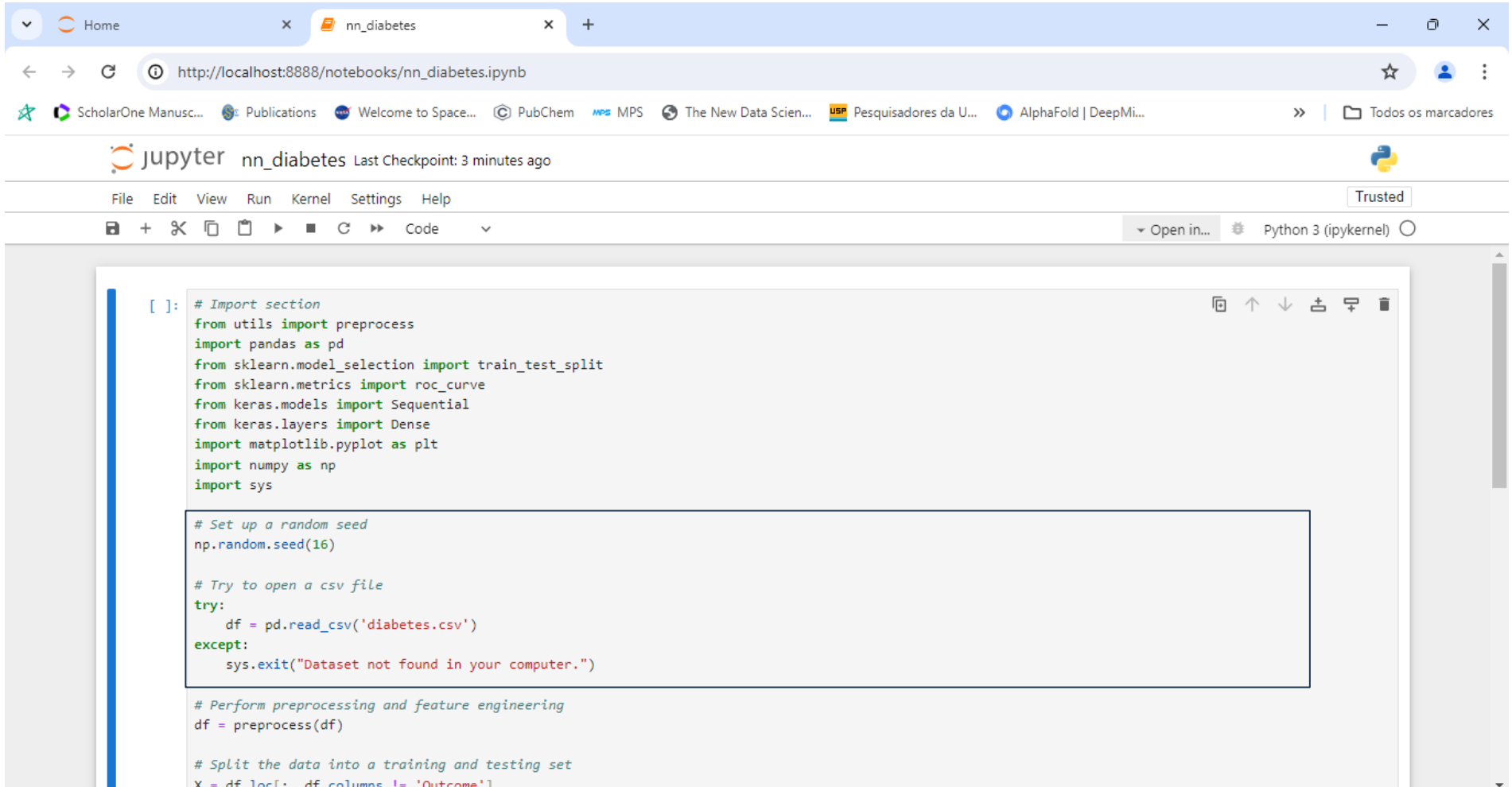
Agora definimos a semente aleatória (*np.random.seed(16)*). Na sequência temos uma porção do código que verifica se o arquivo *diabetes.csv* (*Pima Indians Diabetes Database*) e o código estão na mesma pasta. Se não estiverem, acaba a execução do código.



Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 153). Packt Publishing. Edição do Kindle.

Modelo para Previsão de Diabetes

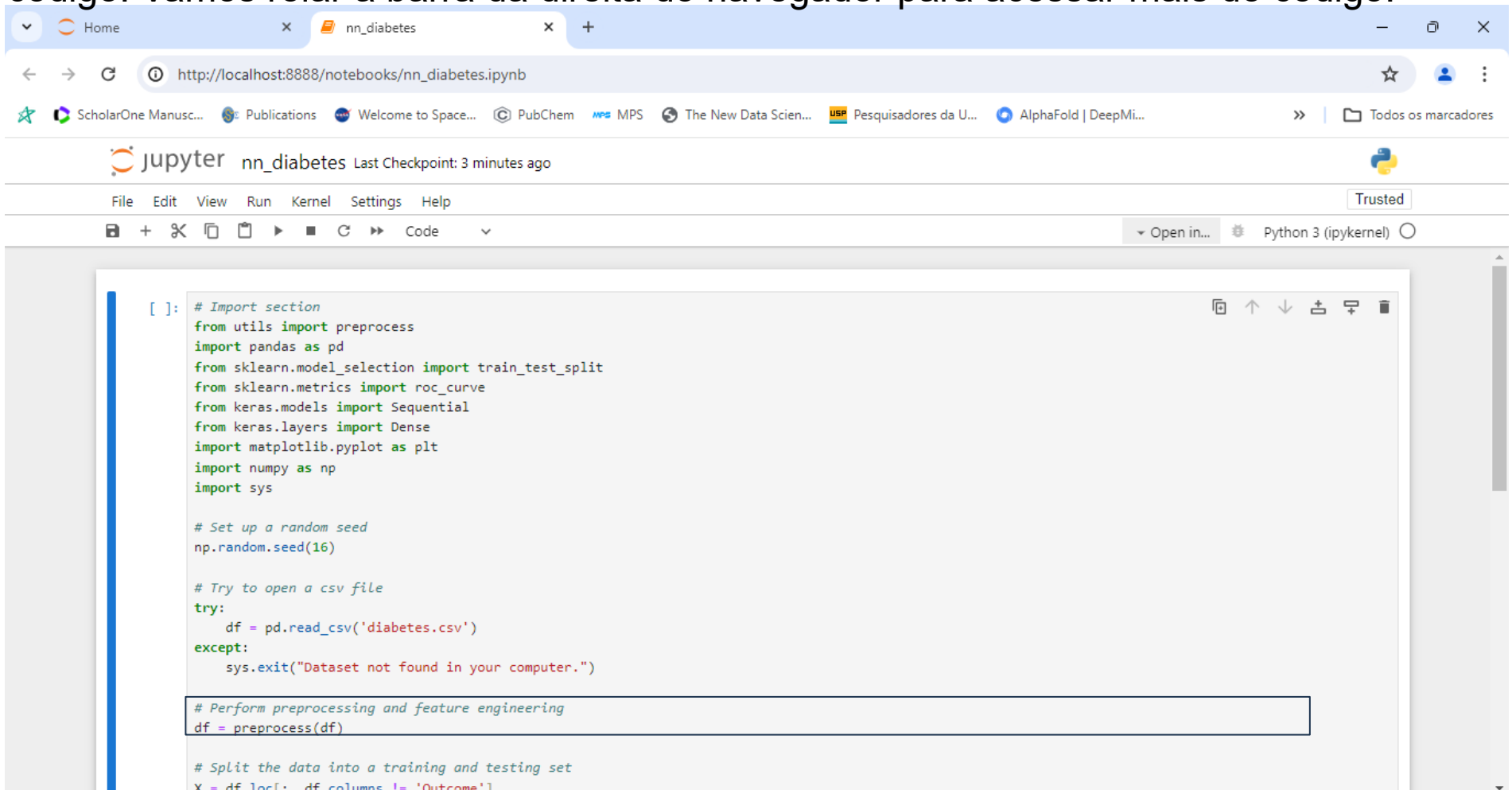
A linha *df = pd.read_csv('diabetes.csv')* lê o conjunto de dados (*Pima Indians Diabetes Database*) e atribui à variável *df,* um *data frame* da biblioteca Pandas.



Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 153). Packt Publishing. Edição do Kindle.

Modelo para Previsão de Diabetes

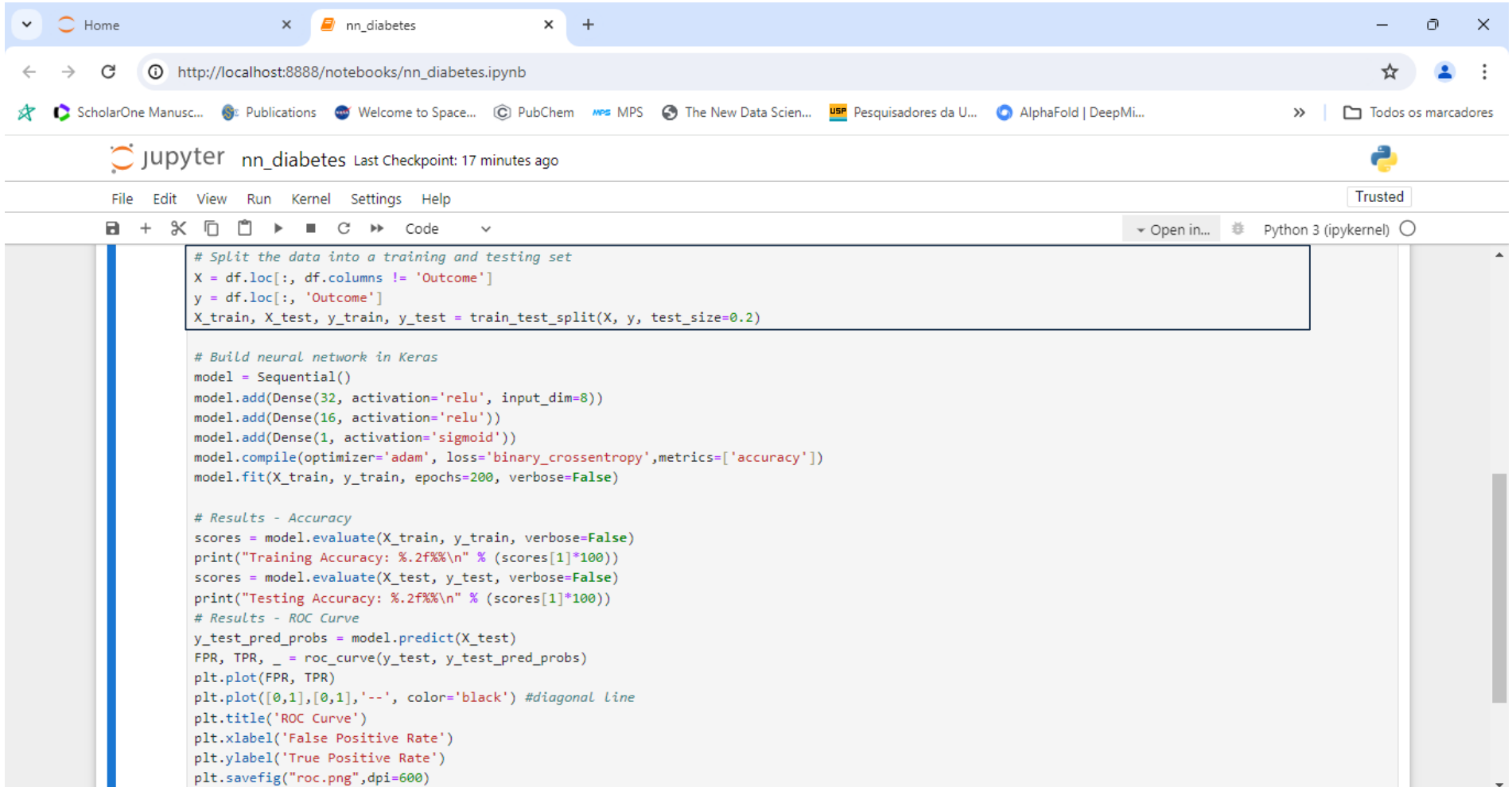
Em destaque abaixo, temos uma linha de código que realiza o preprocessamento dos dados (trata os dados faltantes e escalona os features). Tudo isso com uma linha de código (*df = preprocess(df)*). A função *preprocess()* faz parte do código *utils.py* (disponível na pasta de códigos relacionada a esta aula), e foi importado no início do código. Vamos rolar a barra da direita do navegador para acessar mais do código.



Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 153). Packt Publishing. Edição do Kindle.

Modelo para Previsão de Diabetes

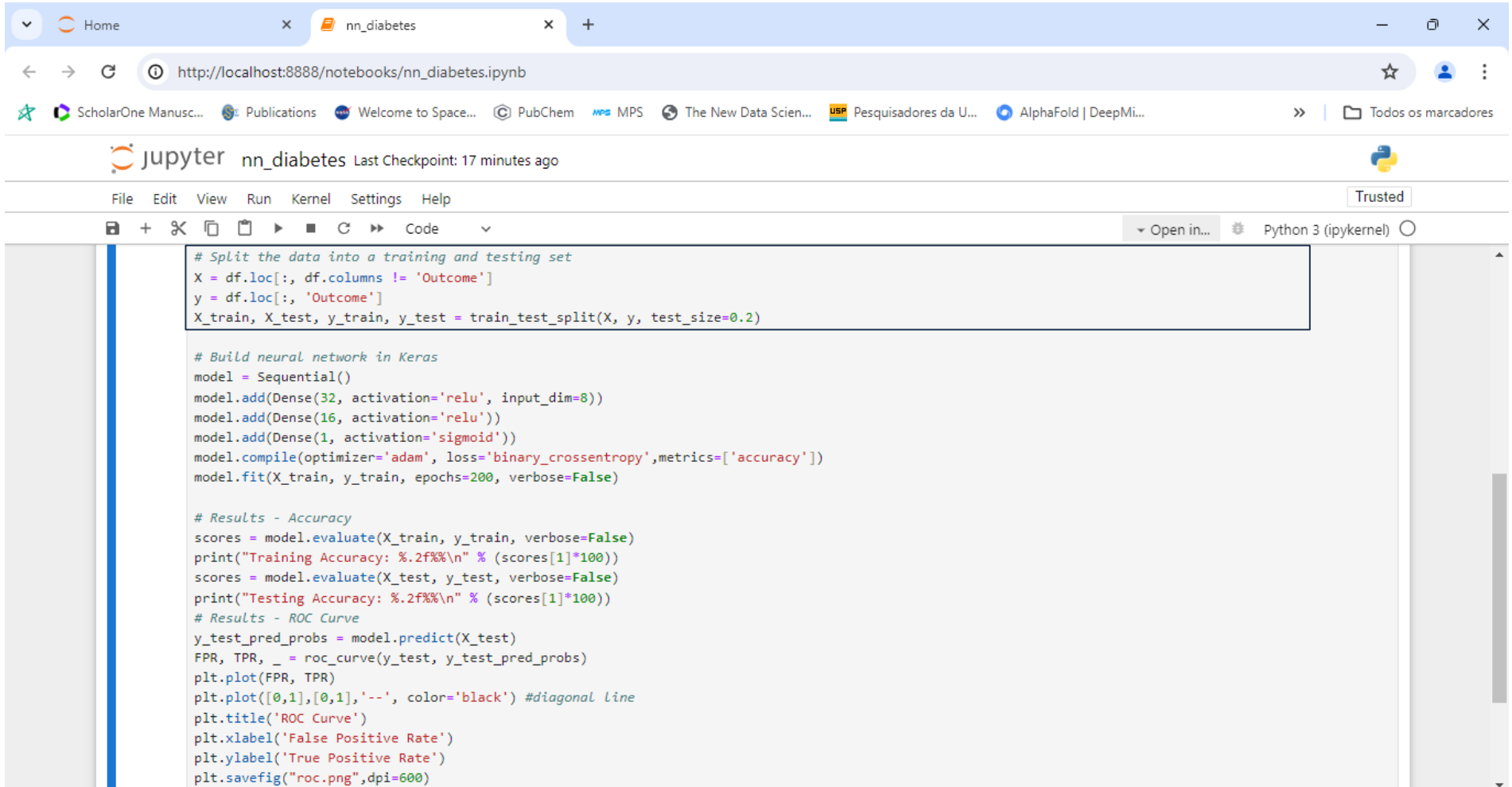
Nesta parte em destaque do código, definimos quais colunas são features (variável *X*) e qual é a variável alvo (variável *y*). A variável *X* recebe um array de dados que envolve todas as linhas que não tem como cabeçalho a string *'Outcome'*. A variável alvo (*y*) tem os dados da coluna com cabeçalho definido pela string *'Outcome'*.



Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 153). Packt Publishing. Edição do Kindle.

Modelo para Previsão de Diabetes

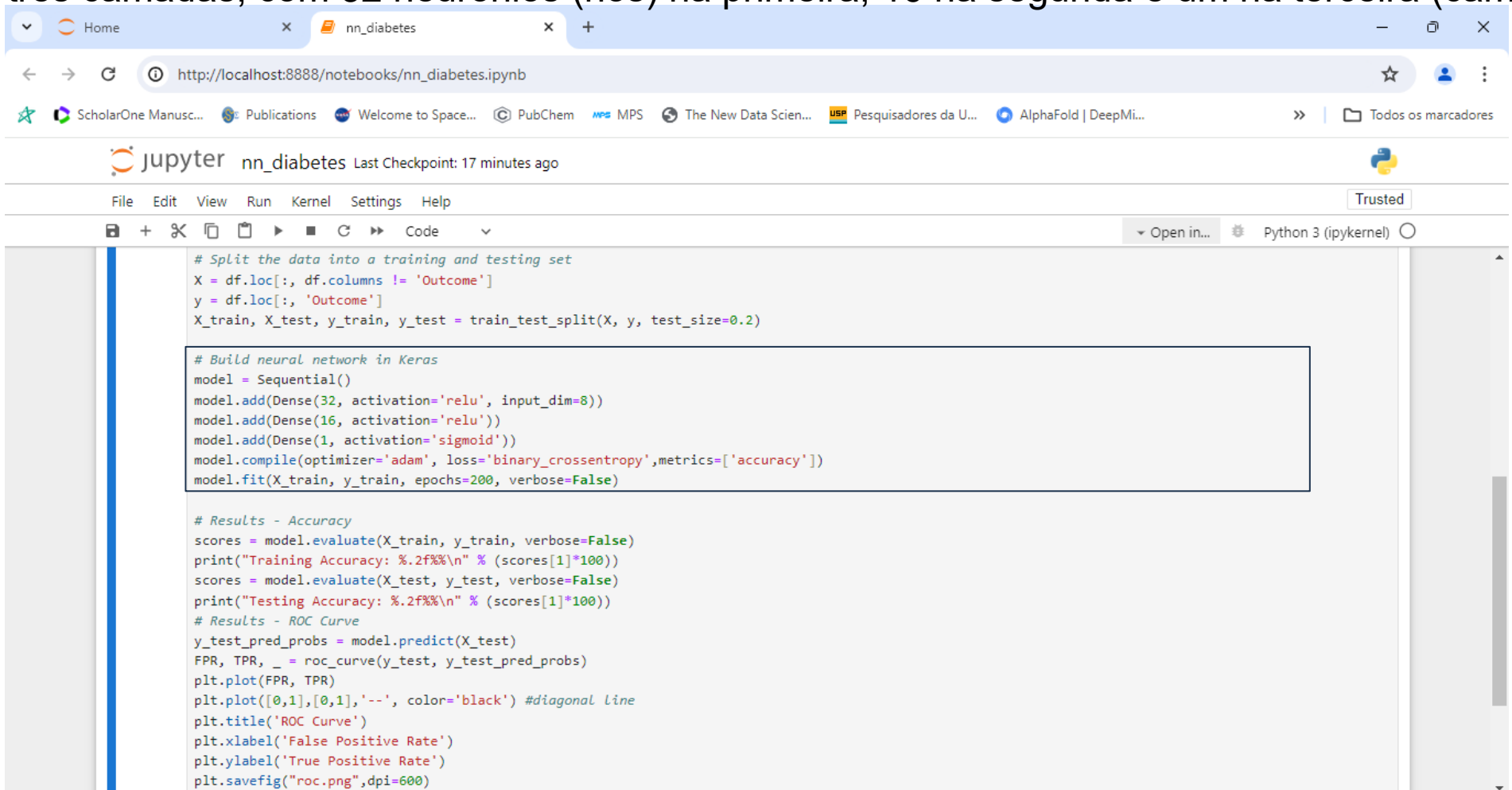
Depois dividimos o conjunto de dados entre conjunto de treinamento e de teste, usando 20 % dos dados (*test_size=0.2*) para o conjunto de teste. Agora estamos prontos para construir a nossa rede neural.



Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 153). Packt Publishing. Edição do Kindle.

Modelo para Previsão de Diabetes

Como já visto no código anterior, usamos o comando *Sequential()* para definir um modelo vazio. Para introduzir camadas de neurônios (nos), usamos o comando *model.add()*. Informamos o número de features (*input_dim*) e o número de unidades (*units*), que é o número de neurônios da camada (nos). Construímos uma rede neural de três camadas, com 32 neurônios (nos) na primeira, 16 na segunda e um na terceira (camada de saída).



Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 153). Packt Publishing. Edição do Kindle.

Modelo para Previsão de Diabetes

Veja que temos liberdade de definir camadas com funções de ativação distintas. Na primeira e segunda camadas usamos a função ReLu (mostrada abaixo). Para a terceira camada usamos a nossa conhecida, a função sigmoide.

Home | nn_diabetes

http://localhost:8888/notebooks/nn_diabetes.ipynb

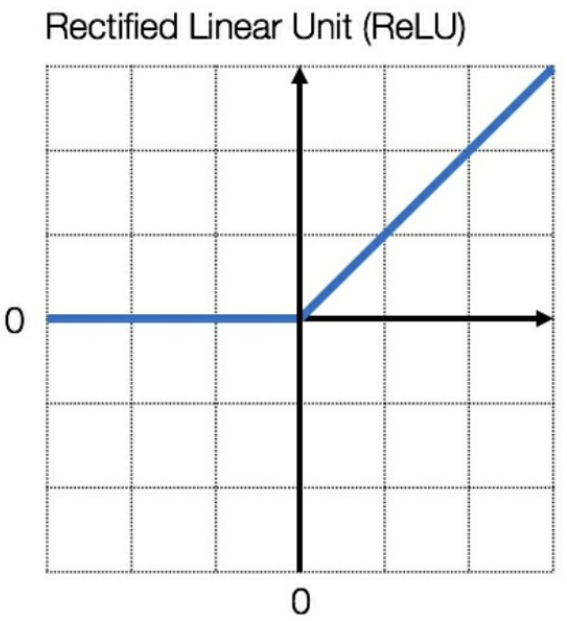
jupyter nn_diabetes Last Checkpoint: 17 minutes ago

File Edit View Run Kernel Settings Help

Trusted

Code | Open in... | Python 3 (ipykernel)

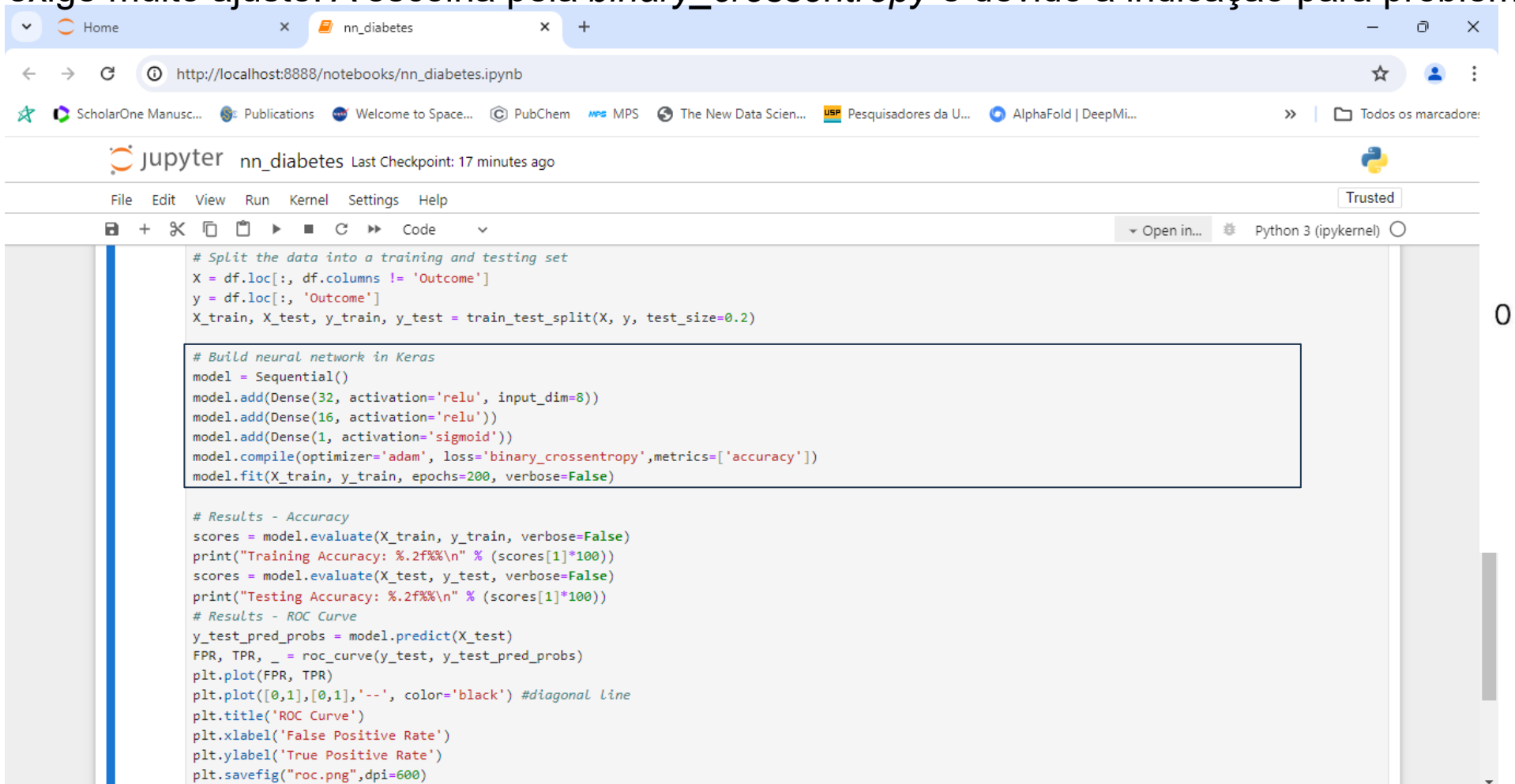
```
# Split the data into a training and testing set
X = df.loc[:, df.columns != 'Outcome']
y = df.loc[:, 'Outcome']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Build neural network in Keras
model = Sequential()
model.add(Dense(32, activation='relu', input_dim=8))
model.add(Dense(16, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='adam', loss='binary_crossentropy',metrics=['accuracy'])
model.fit(X_train, y_train, epochs=200, verbose=False)

# Results - Accuracy
scores = model.evaluate(X_train, y_train, verbose=False)
print("Training Accuracy: %.2f%%\n" % (scores[1]*100))
scores = model.evaluate(X_test, y_test, verbose=False)
print("Testing Accuracy: %.2f%%\n" % (scores[1]*100))
# Results - ROC Curve
y_test_pred_probs = model.predict(X_test)
FPR, TPR, _ = roc_curve(y_test, y_test_pred_probs)
plt.plot(FPR, TPR)
plt.plot([0,1],[0,1],'--', color='black') #diagonal line
plt.title('ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.savefig("roc.png",dpi=600)
```



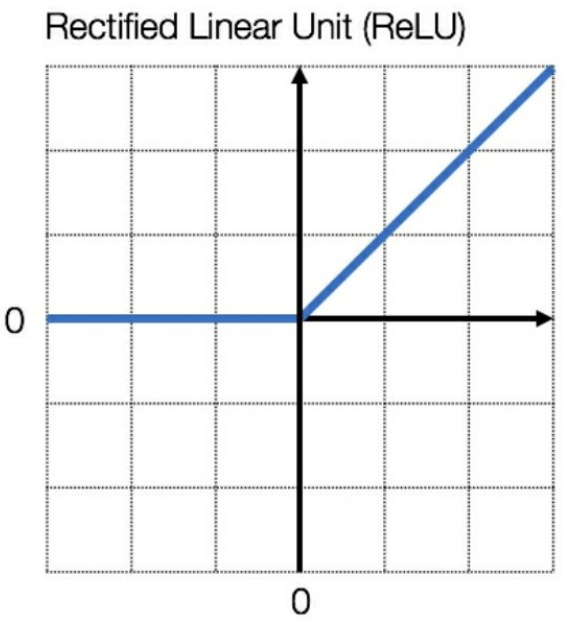
Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 146). Packt Publishing. Edição do Kindle.

Modelo para Previsão de Diabetes

Na primeira camada definimos o número de features com *input_dim = 8*. Na compilação definimos o otimizador (*adam*), a função de perda (*binary_crossentropy*) e a métrica (*accuracy*). Por último estabelecemos o número de iterações (*epochs*). Usamos o otimizador adam que é uma escolha comum para a biblioteca Keras e que não exige muito ajuste. A escolha pela *binary_crossentropy* é devido à indicação para problemas de classificação.



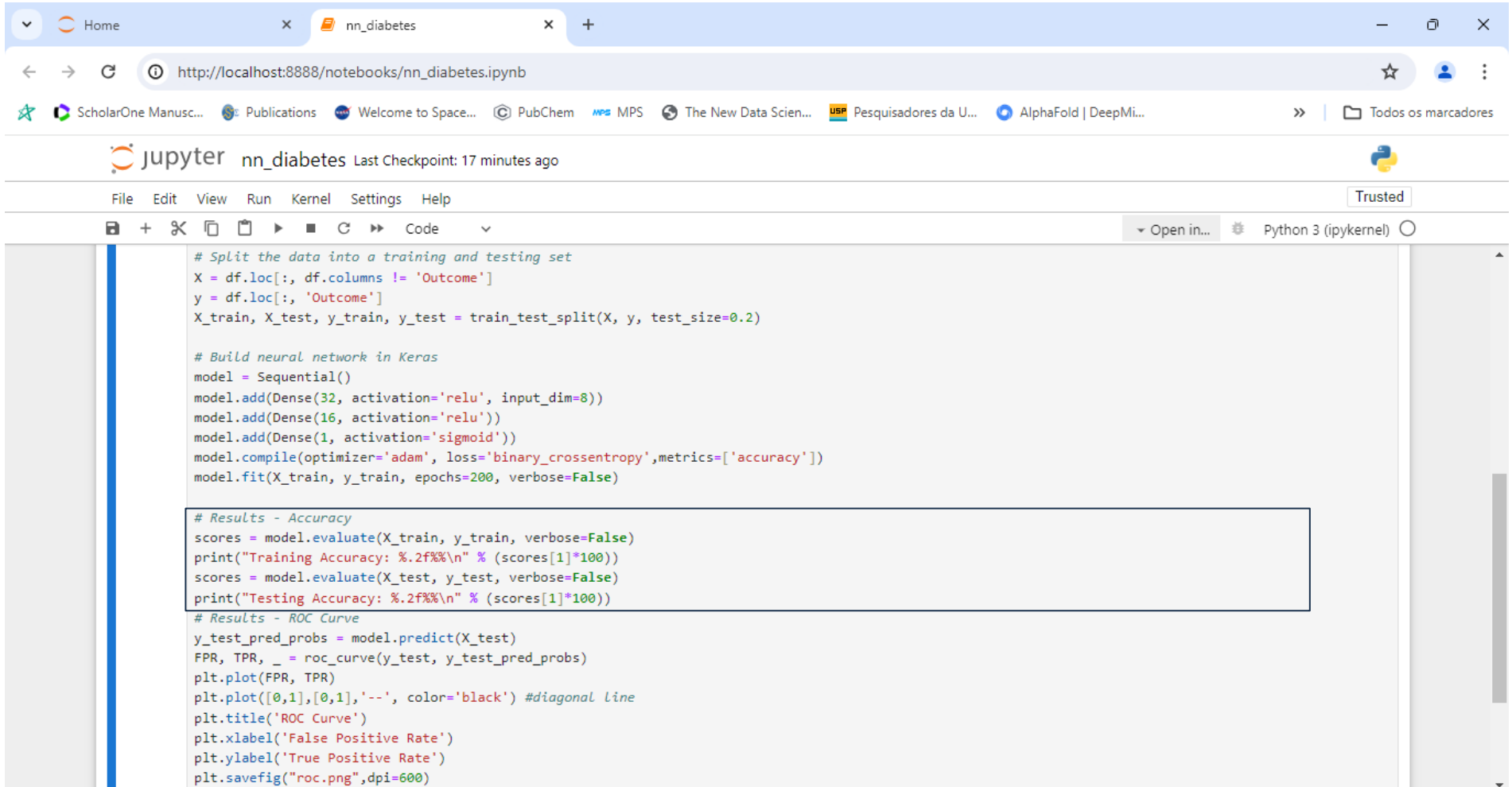

```
# Split the data into a training and testing set
X = df.loc[:, df.columns != 'Outcome']
y = df.loc[:, 'Outcome']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Build neural network in Keras
model = Sequential()
model.add(Dense(32, activation='relu', input_dim=8))
model.add(Dense(16, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='adam', loss='binary_crossentropy',metrics=['accuracy'])
model.fit(X_train, y_train, epochs=200, verbose=False)

# Results - Accuracy
scores = model.evaluate(X_train, y_train, verbose=False)
print("Training Accuracy: %.2f%%\n" % (scores[1]*100))
scores = model.evaluate(X_test, y_test, verbose=False)
print("Testing Accuracy: %.2f%%\n" % (scores[1]*100))
# Results - ROC Curve
y_test_pred_probs = model.predict(X_test)
FPR, TPR, _ = roc_curve(y_test, y_test_pred_probs)
plt.plot(FPR, TPR)
plt.plot([0,1],[0,1],'--', color='black') #diagonal line
plt.title('ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.savefig("roc.png",dpi=600)
```



Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 146). Packt Publishing. Edição do Kindle.

Modelo para Previsão de Diabetes

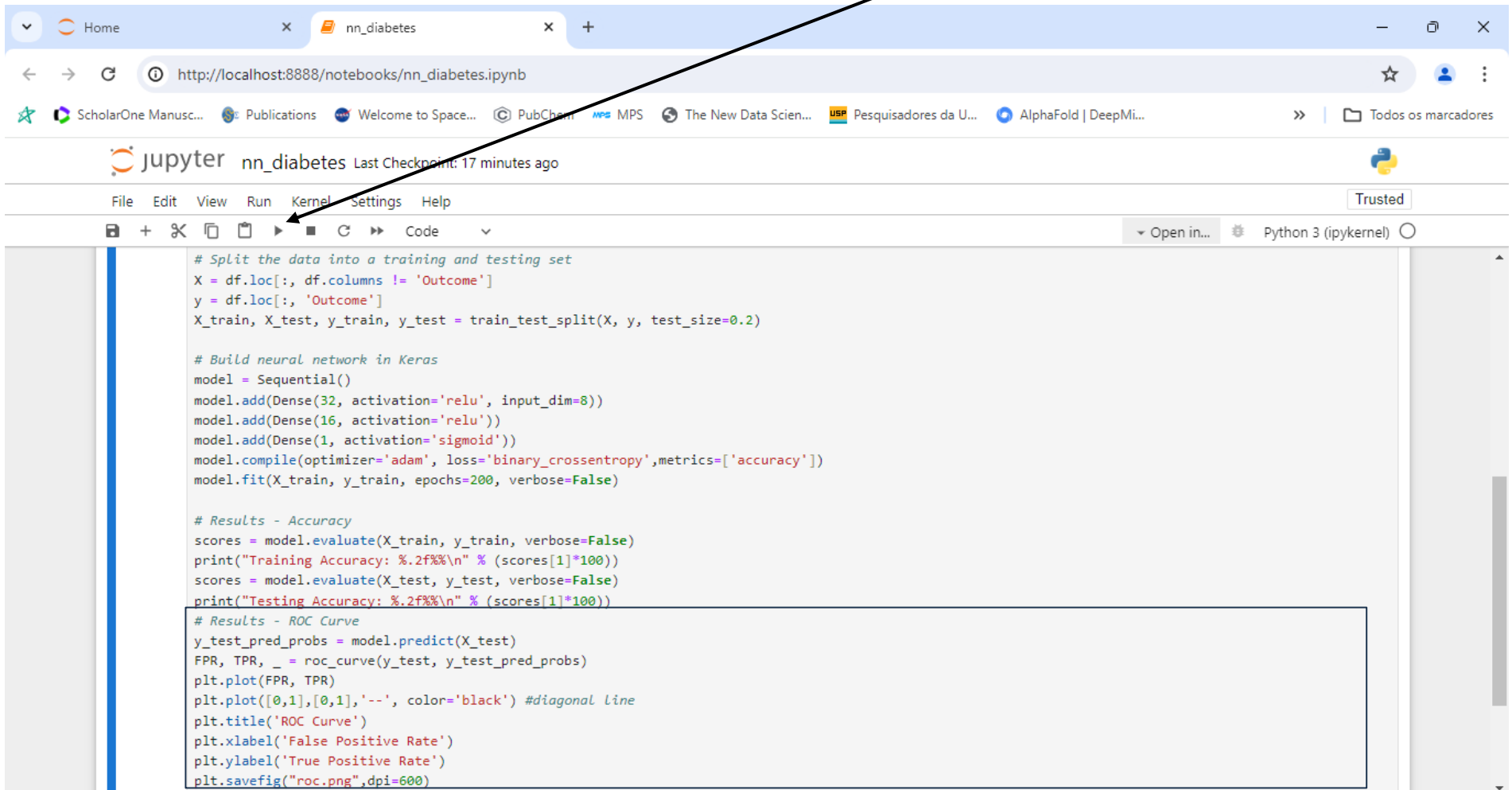
Esta parte do código determina a **acurácia** dos conjuntos treinamento e de teste. A acurácia é a percentagem de amostras classificadas corretamente.



Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 152). Packt Publishing. Edição do Kindle.

Modelo para Previsão de Diabetes

No final do código geramos a **curva ROC** (Fawcett, 2006). Clique no botão indicado para executar o código.

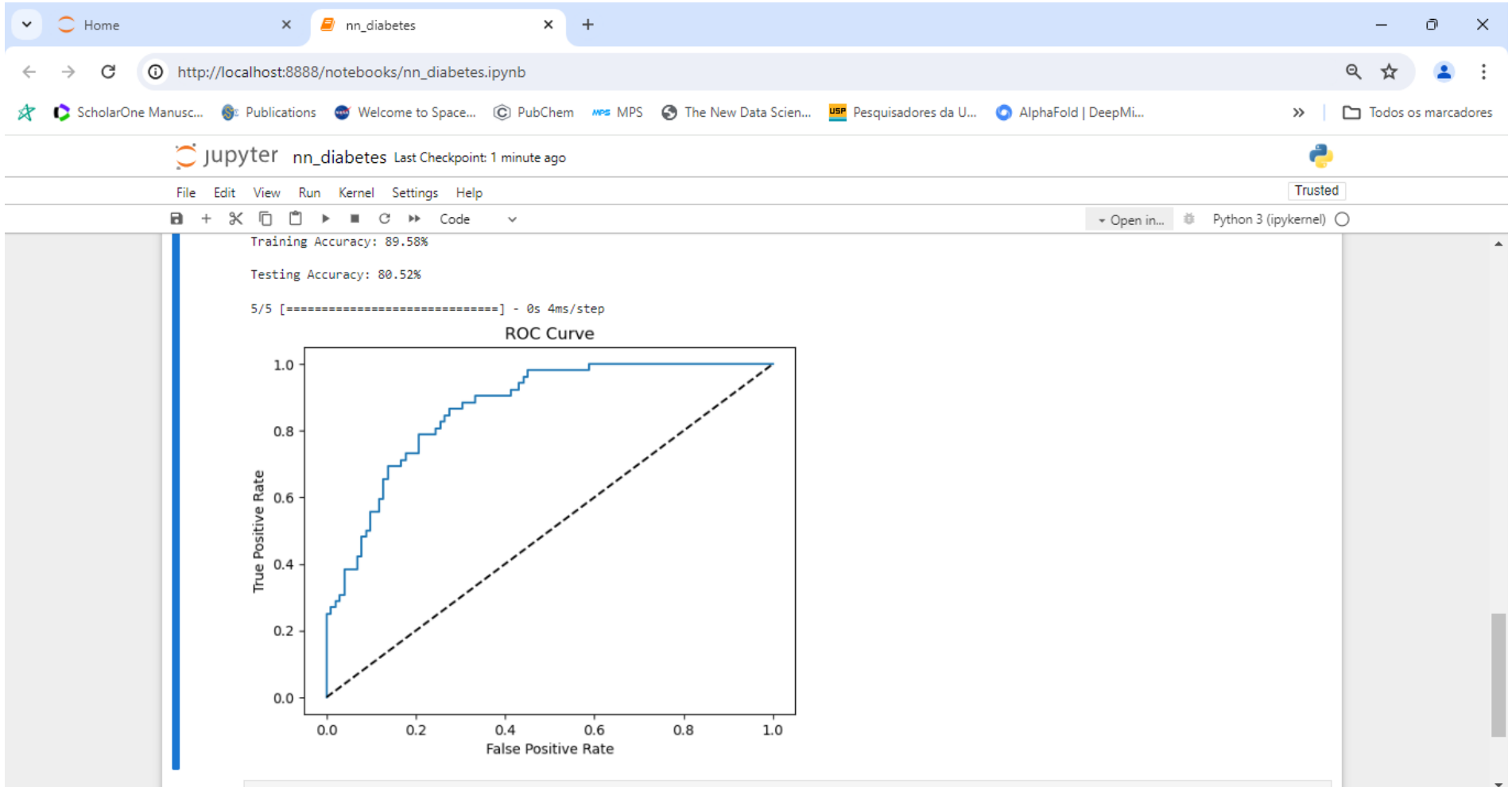


Pode acontecer do código não ser executado no Jupyter por questões de compatibilidade de versões das bibliotecas Keras e TensorFlow. Algo muito comum nesta área. Caso acontecer, você pode tentar rodar o código no seu computador através de linha de comando e usando as versões das bibliotecas indicadas Loy, 2019.

Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 152). Packt Publishing. Edição do Kindle.

Modelo para Previsão de Diabetes

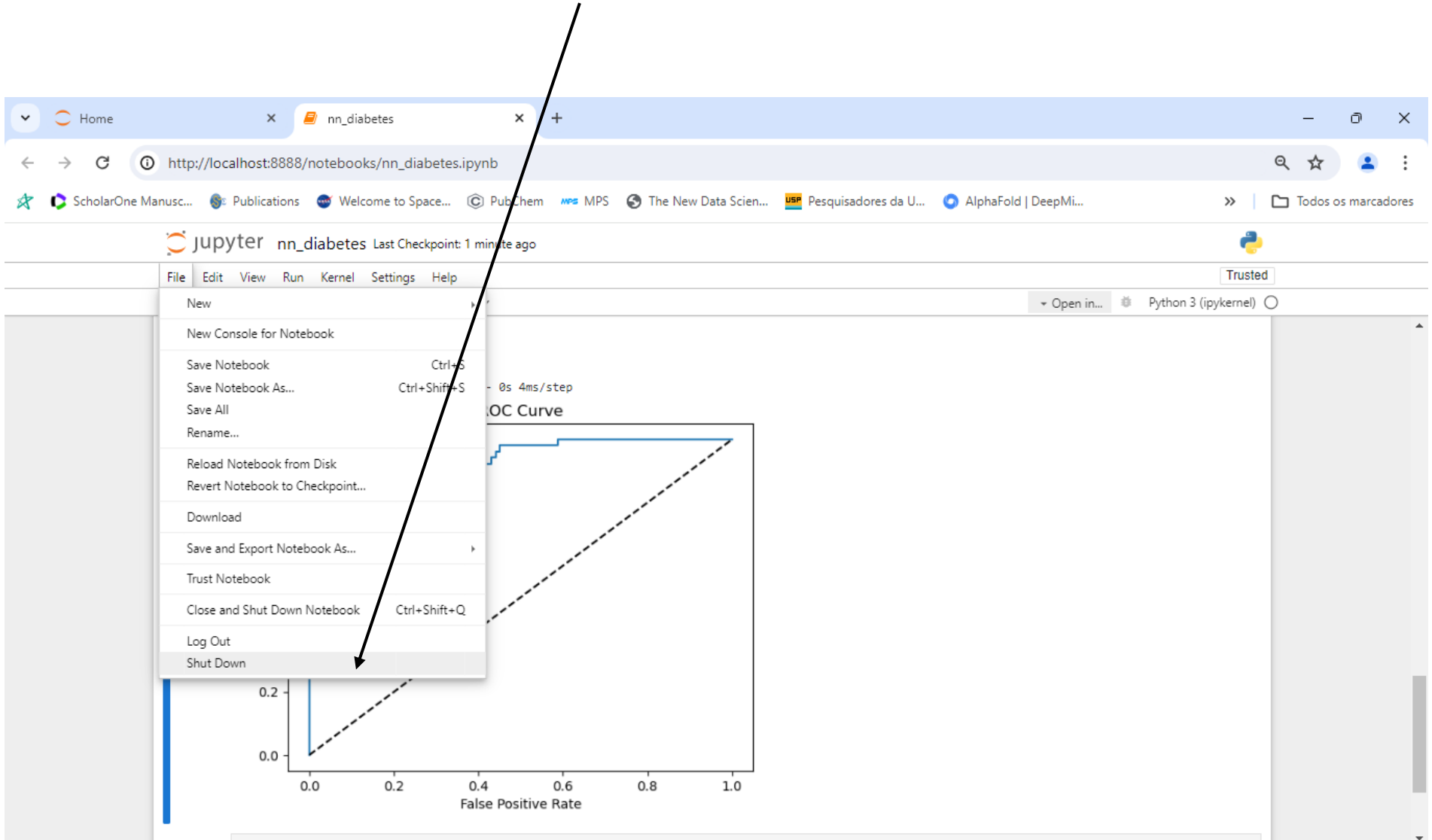
Abaixo temos a curva ROC gerada. O modelo gerado tem as seguintes acurácias.
Training Accuracy: 89.58%
Testing Accuracy: 80.52%



Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 152). Packt Publishing. Edição do Kindle.

Modelo para Previsão de Diabetes

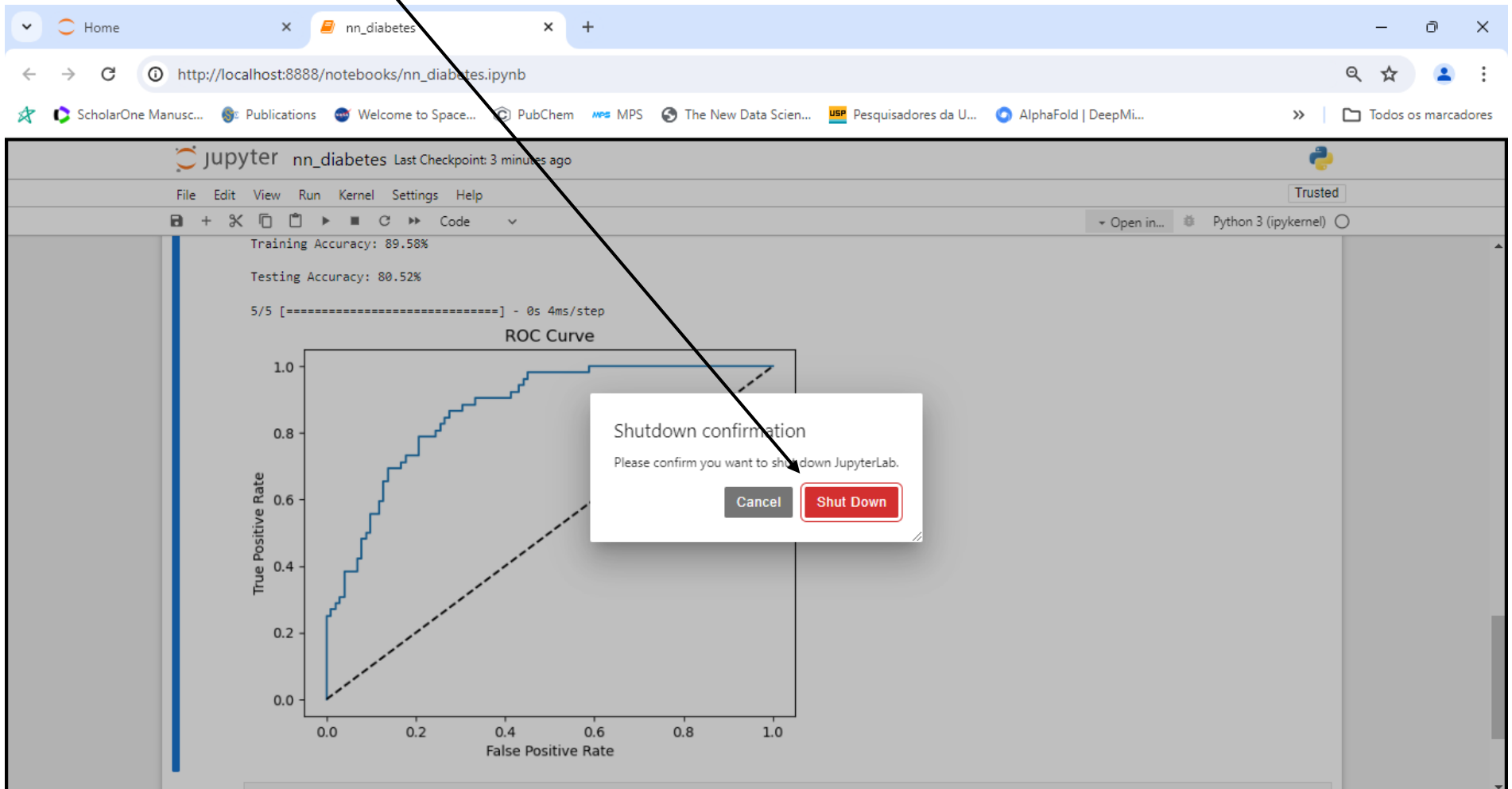
Encerre o Jupyter. Clique *File-> Shut Down*.



Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 152). Packt Publishing. Edição do Kindle.

Modelo para Previsão de Diabetes

Clique em *Shut Down*.



Fonte: Loy, James. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects (p. 152). Packt Publishing. Edição do Kindle.

Exercícios Propostos

1) Implemente em Python usando a biblioteca Keras redes neurais com modelos para as tabelas abaixo. Use o código *nn_keras.ipynb* como protótipo do seu programa. Inicialmente você só precisa mudar os arrays *X* e *y*.

x_1	x_2	x_3	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

x_1	x_2	x_3	y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

x_1	x_2	x_3	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

x_1	x_2	x_3	y
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Exercícios Propostos

2) Modifique os hiperparâmetros dos programas do exercío 1 visando melhorar o poder de previsão. Inclua as seguintes linhas de código aos programas para mostrar o RMSE. Além dos hiperparâmetros *learning_rate* e *epochs* é possível modificar a função de perda e o otimizador. Algumas das opções de função de perda são as seguintes (a primeira já usada): *mean_squared_error, categorical_crossentropy* e *binary_crossentropy.* Para os otimizadores as opções são as seguintes: *SGD, RMSprop, Adam, AdamW, Adadelta, Adagrad, Adamax, Adafactor, Nadam, Ftrl, Lion* e *Loss Scale Optimizer*.

```
# RMSE
from sklearn.metrics import root_mean_squared_error
print("RMSE: ",root_mean_squared_error(y,model.predict(X)))
```

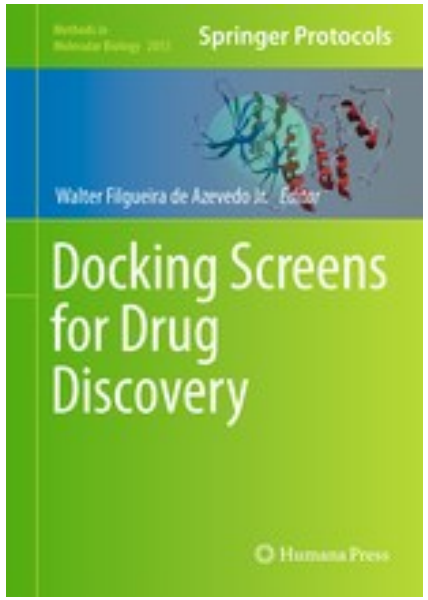
Fonte de informações sobre os otimizadores: Keras Optimizers
Fonte de informações sobre as funções de perda: Keras losses

Exercícios Propostos

3) Modifique o código do programa *nn_diabetes.ipynb* para uma nova arquitetura de rede neural. Use números de camadas e neurônios diferentes. Você pode tentar também mudar a função de perda e o otimizador. Só mantenha fixos o número de neurônios da primeira e última camadas. Tente alguns modelos. Você conseguiu gerar algum modelo com melhor acurácia?

Fonte de informações sobre os otimizadores: Keras Optimizers
Fonte de informações sobre as funções de perda: Keras losses

Autor



Dr. Walter F. de Azevedo, Jr. earned a BSc in Physics (1990), an MSc in Applied Physics (1992), and a DSc in Applied Physics (1997) from the University of São Paulo (Brazil). In his doctoral studies, Dr. Azevedo worked under the supervision of Prof. Yvonne Primerano Mascarenhas (University of São Paulo) and Prof. Sung-Hou Kim (University of California, Berkeley) on a split Doctoral program with a fellowship from the Brazilian Research Council (CNPq). During his first two years at Berkeley, he was under a CNPq fellowship (1993-95). Due to his performance, Prof. S.-H. Kim hired him as Visiting Researcher for the Department of Chemistry, University of California at Berkeley (1995-96).

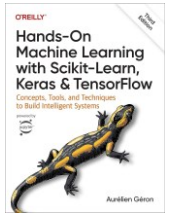
The work developed during these three years at Berkeley resulted in his thesis about the structure of Cyclin-Dependent Kinase 2 (CDK2) in complex with inhibitors (PDB access code: 2A4L) (de Azevedo et al., 1996; de Azevedo et al., 1997). Dr. Azevedo is the first author of both papers, and these publications gathered more than 1,000 citations on the Web of Science. During 1997-98 he had a postdoc position at São Paulo State University (Unesp) with a Fapesp fellowship. He holds a habilitation degree in Physics (livre-docência) from the São Paulo State University (Unesp)(2004). In 1998, Dr. Azevedo participated in a research project with NASA that sent proteins to crystallize in a microgravity environment onboard the Space Shuttle Discovery (STS-95). This research had coverage of Brazilian TV networks. He published a book entitled "Docking Screens for Drug Discovery" with Springer Nature in 2019. This book sold 46,000 copies (April 2024) with over 2 million dollars in sales (https://link.springer.com/book/10.1007/978-1-4939-9752-7). In 2020, the Journal Plos Biology ranked Dr. Azevedo among the most influential researchers in the world (Fields: Biochemistry & Molecular Biology and Biophysics).

Dr. Azevedo has vast editorial experience. He is the frontiers section editor (Bioinformatics/Biophysics) for the Current Drug Targets, section editor (Bioinformatics in Drug Design and Discovery) for the Current Medicinal Chemistry, review editor for Frontiers in Chemistry, associate editor for Exploration of Drug Science, member of the editorial boards Molecular Diversity and the Journal of Molecular Structures, and editor of Docking Screens for Drug Discovery (Methods of Molecular Biology)-Springer Nature. He is a reviewer for over 60 high-impact journals, including Nature Communications and Briefings in Bioinformatics. His research interests are interdisciplinary, with three main emphases: machine learning, complex systems, and computational systems biology. Dr. Azevedo has over 200 scientific publications about protein structures, computer models of complex systems, and simulations of protein systems. These workers have over 7300 citations on the Web of Science (h-index: 48. m-quotient: 1.7), +7800 citations in Scopus (h-index: 50), and +9700 citations on Google Scholar (h-index: 53).

Referências

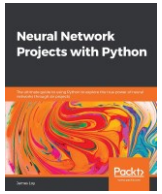
Azevedo FA, Carvalho LR, Grinberg LT, Farfel JM, Ferretti RE, Leite RE, Jacob Filho W, Lent R, Herculano-Houzel S. Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. J Comp Neurol. 2009; 513(5): 532–541. doi: 10.1002/cne.21974. PMID: 19226510 PubMed

Fawcett, T. An introduction to ROC analysis. Pattern Recognit. Lett., 2006, 27, 861–874. PDF



Géron, Aurélien. 2023. Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. 3rd ed. CA 95472: O'Reilly.

Hodgkin AL, Huxley AF. A quantitative description of membrane current and its application to conduction and excitation in nerve. J Physiol. 1952; 117(4): 500–544. doi: 10.1113/jphysiol.1952.sp004764. PMID: 12991237 PubMed



Loy, James. 2019. Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects. Packt Publishing. Edição do Kindle.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Verplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E. Scikitlearn: Machine Learning in Python. J. Mach. Learn. Res., 2011, 12, 2825–2830. PDF

Rosenblatt F. The perceptron: a probabilistic model for information storage and organization in the brain. Psychol Rev. 1958 Nov;65(6):386-408. doi: 10.1037/h0042519. PMID: 13602029. PubMed

Walsh I, Fishman D, Garcia-Gasulla D, Titma T, Pollastri G, ELIXIR Machine Learning Focus Group, Harrow J, Psomopoulos FE, Tosatto SCE. DOME: recommendations for supervised machine learning validation in biology. Nat Methods., 2021, 18(10), 1122–1127. PubMed

Williams RW, Herrup K. The control of neuron number. Annu Rev Neurosci. 1988; 11: 423–453. doi: 10.1146/annurev.ne.11.030188.002231. PMID: 3284447 PubMed


Que a luz da ciência acabe com
as trevas do negacionismo.