PROFESSOR DE MATEMÁTICA ONLINE
Revista eletrônica da Sociedade Brasileira de Matemática

PMO v.10, n.4, 2022
ISSN: 2319-023X
https://doi.org/10.21711/2319023x2022/pmo1032

Redes Neurais no Ensino Básico

Márcio Batista André Oliveira Martins

Resumo

Neste trabalho propomos o uso, por parte dos professores de matemática, de Redes Neurais Artificiais, uma área da inteligência artificial que vem crescendo muito nos últimos anos. O trabalho apresenta como as redes neurais artificiais podem ser empregadas na educação básica, com o intuito de inserir os alunos em um projeto de matemática e tecnologias inovadoras. O trabalho tem como tema central a construção de uma rede neural para reconhecimento de dígitos manuscritos com códigos em linguagem Python.

Palavras-chave: Matemática; Redes Neurais; Ensino Médio; Python.

Abstract

In this work we propose the use, by mathematics teachers, of Artificial Neural Networks, an area of the artificial intelligence, which has attracted many scholars in the recent years. We present how the artificial neural networks can be used in high school, in order to introduce students in a project of mathematics and innovative technologies. The main purpose of this paper is the construction of a neural network for the recognition of handwritten digits using codes in Python language.

Keywords: Mathematics; Neural Network; High School; Python.

1. Introdução

Diante da necessidade de uma participação ativa no acelerado processo das transformações do mundo tecnológico, a qual tem sido discutida como um dos desafios presentes na educação e na prática dos educadores, a escola vem sendo cobrada por uma modernização na prática educativa para se adequar à realidade atual de seus alunos, tanto social como profissional.

O Novo Ensino Médio traz mudanças importantes para o ensino, para serem implementadas nas escolas a partir de 2021. Espera-se que as modificações estejam em prática até 2022. Entre as propostas é cobrado que se tenha menos aulas expositivas e mais atividades como projetos, oficinas e atividades práticas significativas, que sejam determinantes na formação técnica e profissional dos alunos.

Nessa perspectiva, o presente trabalho propõe aos professores de matemática a utilização de redes neurais e *deep learning* no ensino básico, mais especificamente a utilização de redes neurais para reconhecimento de dígitos manuscritos. Tal proposta objetiva deixar o ensino de matemática mais significativo, visando combater o desinteresse dos alunos, visto que, temas que envolvem tecnologia são agradáveis aos estudantes tornando as aulas mais atrativas e participativas.

SBM SOCIEDADE BRASILEIRA DE MATEMÁTICA

1.1. História das redes neurais

As redes neurais artificiais são modelos matemáticos que apresentam um grande poder de processamento que se estabelece na estrutura fisiológica do cérebro humano, com objetivo de reproduzir suas funções.

As redes neurais foram desenvolvidas inicialmente pelo neurofisiologista Warren McCulloch e pelo matématico Walter Pitts, ainda na década de 1940. Eles criaram um modelo computacional simplificado de como neurônios artificiais podem trabalhar para realizar cálculos usando lógica proposicional, construindo, assim, a primeira arquitetura de rede neural artificial, ainda sem capacidade de aprendizado. Desde então, foram criadas muitas outras arquiteturas de redes neurais, porém, a aplicação das técnicas esbarrou no poder computacional da época.

Durante os anos 80 novas arquiteturas foram inventadas, juntamente com novas técnicas de treinamento, o que provocou uma nova demanda de interesse em pesquisas na área, mas o avanço continuou lento. Nos anos 90, novas técnicas de aprendizado de máquina foram inventadas como o Support Vector Machine (SVM) que pôs as redes neurais em segundo plano.

A partir dos anos 2000, as redes neurais voltaram ao centro das atenções devido ao crescimento do poder de processamento dos computadores além da quantidade de dados disponíveis, o que facilitou a implementação e o treinamento dessas redes e, consequentemente, avanço do Deep Learning.

As Redes Neurais Artificiais são formadas por células computacionais denominadas neurônio ou unidades de processamento conectadas, onde cada neurônio é alimentado por todos os neurônios da camada imediatamente anterior. Tal modelo pode solucionar uma grande variedade de problemas como reconhecimento facial, classificação de imagens, recomendar vídeos de acordo com o interesse dos usuários de uma plataforma ou aprender vencer campeões de jogos como o xadrez.

1.2. Neurônio artificial

O neurônio biológico, com tamanho aproximado de 100μm, é composto por um corpo celular com muitas extensões de ramificações chamadas dendritos, além de uma longa extensão chamada Axônio.

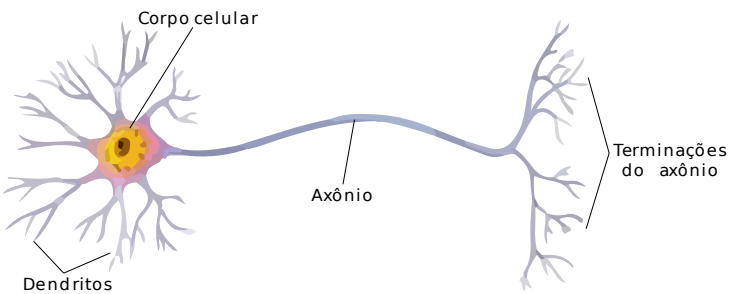


Figura 1: Neurônio biológico
Fonte:[8]

Os dendritos têm como função a recepção de estímulos nervosos vindos de outros neurônios ou do ambiente, que são transmitidos para o corpo celular, também chamado de soma, que combina

as informações e as processa, gerando um novo impulso que depende da intensidade e frequência do impulso anterior. O novo estímulo é transportado pelo axônio ao encontro de outro neurônio. O contato das terminações do axônio de um neurônio com os dendritos de outro é chamado de sinapse, desse modo, o sinal do neurônio flui da esquerda para a direita transmitindo estímulos elétricos através de sinapses.

As redes neurais são compostas por unidade de processamento, inspiradas em neurônios biológicos, denominadas neurônios artificiais, dispostos em camadas e densamente interligados. Um neurônio artificial é representado no diagrama abaixo:

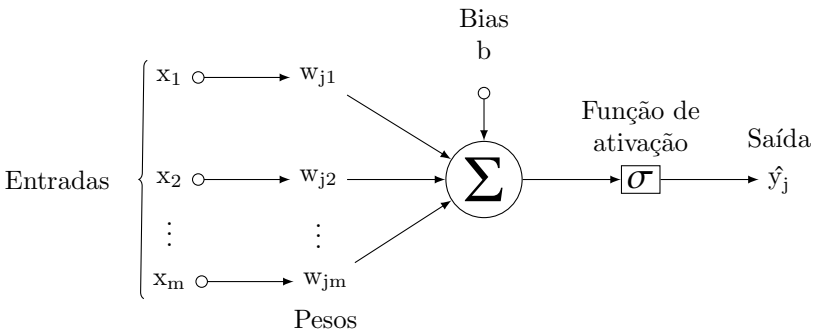


Figura 2: Neurônio artificial
Fonte: [12]

Elementos de um neurônio artificial:

- $\mathbf{X}_1 b = [1, x_1, x_2, \ldots, x_m]$ é um vetor, pertencente ao conjunto de treinamento, onde suas componentes são os sinais de entrada; inserimos como primeira componente 1, para trabalhar com o bias embutido no vetor de pesos;
- $\mathbf{W} = \left[w_{j0} = b, w_{j1}, w_{j2}, \ldots, w_{jm}\right]$ é o vetor onde suas componentes são os pesos sinápticos que são aprendidos durante o treinamento;
- b é termo de bias, que é um parâmetro externo do neurônio;
- v_j é chamado de campo local induzido;
- σ é uma função de ativação;
- $\hat{y}_j$ é o sinal de saída do neurônio.

O integrador representado pele letra Σ realiza a soma das entradas ponderadas por esses pesos, isto é, $v_j = \sum_{i=0}^{m} w_{ji} x_i$. A saída da rede é obtida aplicando a função de ativação σ em v_j. Podemos simplificar esses cálculos usando representação matricial como segue:

$$v_j = \mathbf{X}_1 b \cdot W^T = \begin{bmatrix} 1 & x_1 & x_2 & \ldots & x_m \end{bmatrix} \cdot \begin{bmatrix} w_{j0} = b \\ w_{j1} \\ w_{j2} \\ \vdots \\ w_{jm} \end{bmatrix} = x_0 w_{j0} + x_2 w_{j2} + \ldots + x_m w_{jm} = \sum_{i=0}^{m} x_i w_{ji}$$

Aplicando a função de ativação σ no campo de ativação v_j, obtemos a saída do neurônio $\hat{y}_j$.

$$\hat{y}_j = \sigma(v_j)$$

Observe que, no neurônio, cada entrada está associada a um peso denominado peso sináptico que pode assumir valores positivos ou negativos, dependendo do comportamento da conexão ser excitatório ou inibitório, respectivamente.

Exemplo 1. Considere o conjunto $D = \{[65, 2.5], [110, 3.2], [70, 2.2], [125, 3.1]\}$, os elementos do conjunto têm como primeira componente o peso(g) e como segunda componente o pH de limões e laranjas, e a matriz $R = [0, 1, 0, 1]$ traz os rótulos do conjunto D, onde 0 representa limão e 1 representa laranja. Nesse caso o neurônio a ser treinado terá duas entradas, uma para peso, outra para pH. A saída do neurônio será 0 para indicar que é um limão e, 1 para indicar que é uma laranja.

1.3. O termo bias

O termo bias é um parâmetro acrescentado na equação do campo local induzido $v = w_{ji}x_i + b_j$, o qual podemos interpretar como uma equação de uma reta, onde w_{ji} é o coeficiente angular e b_j é o coeficiente linear da reta. Em uma reta, fazendo variar o coeficiente angular, giramos a reta no sentido horário ou anti-horário. Por sua vez, o coeficiente linear, o termo de bias b_j, define o intercepto com o eixo y, desse modo, o bias aumenta a liberdade de posicionamento da reta. Cabe observar que sem o termos bias teriamos como intercepto sempre o 0, isto é, uma equação linear. A reta que separa as duas classes de um conjunto de dados binários é chamada reta de decisão.

Veja os gráficos abaixo

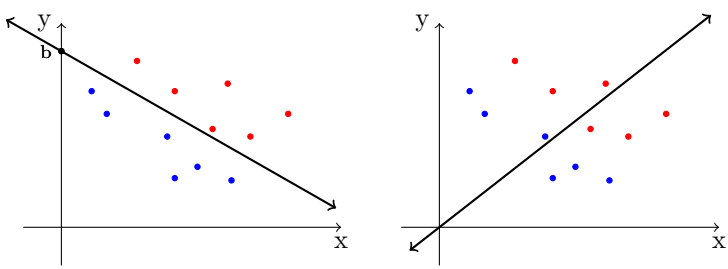


Figura 3: À esquerda temos o valor do bias livre, À direita fixado em 0, observe que se não existir um termo de bias diferente de 0, o hiperplano representado pela reta não conseguirá separar as classes, visto que, a reta só pode rotacionar em torno da origem.

Fonte: Elaborada pelos autores

1.4. Funções de ativação

As funções de ativação são transformações não lineares aplicadas no campo de ativação de cada neurônio agregando um maior poder de processamento às redes neurais. Assim, a importância da não linearidade de tais funções é permitir modelar problemas mais complexos, pois devido a serem os campos de ativação operações lineares, sem essas transformações as redes neurais só

poderiam ser aplicadas a problemas lineares. Veja abaixo os gráficos das funções de ativação Degrau $\phi(x) = \begin{cases} 1, & \text{se } x \geq 0 \\ 0, & \text{se } x < 0 \end{cases}$ e da função Sigmoid $\sigma(x) = \frac{1}{1+\exp(-x)}$.

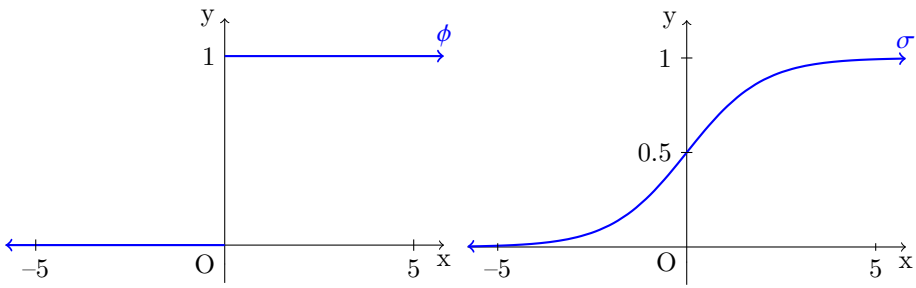


Figura 4: À esquerda: Função Degrau. À direita: Função Sigmoid.
Fonte: Elaborada pelos autores.

2. Redes neurais de camada única

As redes neurais de camada única, também chamadas de perceptroons, têm como característica principal que todas as entradas são conectadas diretamente com as saídas da rede neural. Tais redes só são capazes de resolver problemas linearmente separáveis, isto é, que as classes podem ser separadas por retas, como provou Rosemblatt, em 1958. Os perceptrons construídos com apenas um neurônio na camada de saída são limitados a classificação de problemas binários, aumentando a quantidade de neurônios da camada de saída é possível trabalhar com mais de duas classes.

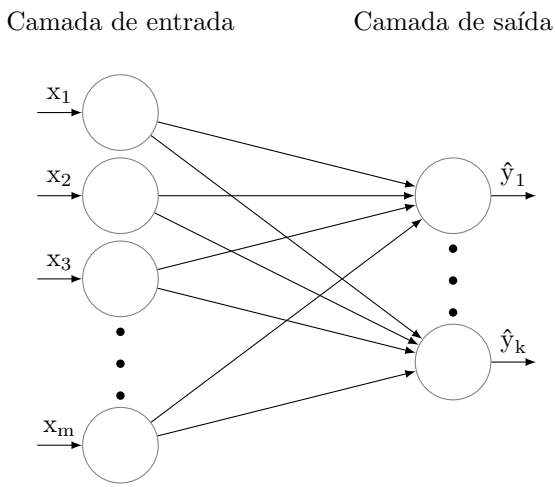


Figura 5: Rede perceptron com m entradas e k saídas.
Fonte: Elaborada pelos autores.

3. Treinamento de um perceptron de camada única

Para treinar uma rede preceptron de camada única vamos utilizar as equações de atualização de pesos obtidas com o algoritmo Least Mean Squares (LMS) Mínimos quadrados médios. Considere o conjunto T de treinamento com k classes e n amostras:

$$T = \left\{ \left(\mathbf{x}^{(1)}, \mathbf{y}^{(1)} \right), \left(\mathbf{x}^{(2)}, \mathbf{y}^{(2)} \right), \ldots, \left(\mathbf{x}^{(n)}, \mathbf{y}^{(n)} \right) \right\} \tag{1}$$

Onde:

- $\mathbf{x}^{(j)} = \left[x_1^{(j)}, x_2^{(j)}, \ldots, x_m^{(j)} \right]$ é um vetor m-dimensional;
- $\mathbf{y}^{(j)} = \left[y_1^{(j)}, y_2^{(j)}, \ldots, y_k^{(j)} \right]$ é um vetor k-dimensional.

Para treinar o perceptron, vamos utilizar o termo de bias embutido, isto é, como primeira componente do vetor de pesos, desse modo acrescentaremos 1 como primeira componente do vetor de entrada. Assim, para o i-ésimo neurônio, um vetor de entrada é da forma $\mathbf{x}^{(j)} = \left[1, x_1^{(j)}, x_2^{(j)}, \ldots, x_m^{(j)} \right]$ e o vetor de pesos é da forma $\mathbf{w}^{(j)} = \left[w_{i0}^{(j)} = b_i, w_{i1}^{(j)}, w_{i2}^{(j)}, \ldots, w_{im}^{(j)} \right]$.

Sendo $\epsilon_{i(j)}$ o erro do i-ésimo neurônio da camada de saída, ao ser inserido na rede o j-ésimo exemplo do conjunto de treinamento, é definido por

$$\epsilon_i^{(j)} = y_i^{(j)} - \hat{y}_i^{(j)}. \tag{2}$$

Onde:

$$\hat{y}_j = \sigma(v_j)$$

é a predição da rede para a j-ésimo exemplo do conjunto de treinamento , σ a função de ativação do neurônio e

$$\begin{aligned} v_j &= \mathbf{x}^j b \cdot \mathbf{w}^{(j)^T} \\ &= \begin{bmatrix} 1 & x_1^{(j)} & x_2^{(j)} & \ldots & x_m^{(j)} \end{bmatrix} \cdot \begin{bmatrix} w_{i0}^{(j)} = b_i \\ w_{i1}^{(j)} \\ w_{i2}^{(j)} \\ \vdots \\ w_{im}^{(j)} \end{bmatrix} \\ &= x_0^{(j)} w_{i0}^{(j)} + x_1^{(j)} w_{i1}^{(j)} + \ldots + x_m^{(j)} w_{im}^{(j)} \\ &= \sum_{i=0}^{m} x_r^{(j)} w_{ir}^{(j)} \end{aligned} \tag{3}$$

O treinamento da rede é a atualização dos pesos para que predição se torne suficientemente precisa, para a correção do valor do peso w_{ir} referente do J-ésimo exemplo do conjunto de treinamento para o $(j+1)$-ésimo utilizaremos a seguinte equação.

$$w_{ir}^{(j+1)} = w_{ir}^{(j)} + \eta \epsilon_i^{(j)} x_r^{(j)}. \tag{4}$$

Onde η é a taxa de aprendizado da rede.

Exemplo 2. Vamos construir um perceptron de camada única, para isso considere o conjunto de dados fictícios abaixo:

Temperatura(°C)	Umidade(%)	Label(rótulo)
21	91	chuvoso
20	70	ensolarado
18	70	chuvoso
29	85	ensolarado
23	80	chuvoso
23	70	ensolarado

Figura 6: Conjunto de dados fictícios de umidade, temperatura e dia chuvoso/ensolarado
Fonte: Elaborada pelos autores.

Vamos treinar uma rede perceptron de camada única para fazer a predição se o dia é chuvoso ou ensoralado, com base na temperatura e na umidade. Para isso vamos construir a matriz X com os dados de entrada na rede, retirados da tabela acima, e a matriz Y com os rótulos, onde 1 representa chuvoso e 0 representa ensolarado. Vamos utilizar com função de ativação a função degrau.

$$X = \begin{bmatrix} 21 & 20 & 18 & 29 & 23 & 23 \\ 91 & 70 & 70 & 85 & 80 & 70 \end{bmatrix}$$

$$Y = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Utilizaremos como matriz de pesos inicial a matriz de 1 linha e 3 colunas, com todos os seus elementos nulos – observe que o primeiro elemento se trata do termo bias.

$$W_0 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$

Utilizaremos $\eta = 0.01$ como taxa de aprendizado, que determina a velociade com que a rede aprende.

Como estamos trabalhando com o bias embutido na primeira coluna da matriz de pesos, será acrescentado o número como primeiro elemento nos vetores de entrada.

$$x_1 b = \begin{bmatrix} 1 & 21 & 91 \end{bmatrix}$$

$$x_2 b = \begin{bmatrix} 1 & 20 & 70 \end{bmatrix}$$

$$x_3 b = \begin{bmatrix} 1 & 18 & 70 \end{bmatrix}$$

$$x_4 b = \begin{bmatrix} 1 & 29 & 85 \end{bmatrix}$$

$$x_5 b = \begin{bmatrix} 1 & 23 & 80 \end{bmatrix}$$

$$x_6 b = \begin{bmatrix} 1 & 23 & 70 \end{bmatrix}$$

Vamos inserir os dados na rede, que terá apenas um neurônio, três entradas e uma saída.

Os dados entradas serão inseridos um a um na rede, veja:

Para a primeira entrada $x_1b = \begin{bmatrix} 1 & 21 & 91 \end{bmatrix}$:

1. Calcular o campo local induzido para a primeira entrada;

$$v = x_1b \cdot W^T = \begin{bmatrix} 1 & 21 & 91 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = 1 \cdot 0 + 21 \cdot 0 + 91 \cdot 0 = 0$$

2. Aplicar a função de ativação degrau no campo local induzido;

$$y = \phi(v) = \phi(0) = 1$$

3. Calcular o erro E da predição da rede;

$$E_1 = Y_1 - y_1 = 1 - 1 = 0$$

4. Atualizar os pesos da rede.

$$W_1 = W_0 + eta \cdot E_1 \cdot x_1b$$
$$W_1 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} + 0.01 \cdot 0 \cdot \begin{bmatrix} 1 & 21 & 91 \end{bmatrix}$$

$$W_1 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$

Note que W_1 é igual a W_0, isto é, os pesos não foram atualizados. Isso ocorreu por que a rede acertou a predição. Os pesos só são atualizados quando a rede erra a predição.

Vamos repetir o processo para a segunda entrada: $x_1b = \begin{bmatrix} 1 & 20 & 70 \end{bmatrix}$

1. Calcular o campo local induzido para a primeira entrada;

$$v_2 = x_2b \cdot W_1^T = \begin{bmatrix} 1 & 20 & 70 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = 1 \cdot 0 + 21 \cdot 0 + 91 \cdot 0 = 0$$

2. Aplicar a função de ativação degrau no campo local induzido;

$$y = \phi(v) = \phi(0) = 1$$

3. Calcular o erro E da predição da rede;

$$E_2 = Y_2 - y_1 = 0 - 1 = -1$$

4. Atualizar os pesos da rede.

$$W_2 = W_1 + \text{eta} \cdot E_2 \cdot x_2 b$$
$$W_1 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} + 0.01 \cdot (-1) \cdot \begin{bmatrix} 1 & 20 & 70 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} -0,01 & -0,2 & -0,7 \end{bmatrix}$$

Fazendo o mesmo procedimento para as entradas que faltam, isto é, de x_3 até x_6, teremos completado a inserção de todos os elementos do conjunto de treinamento, chamaremos a apresentação de todos os dados do conjunto de treinamento de uma *época*. Para concluir o treinamento de maneira satisfatória devemos inserir o conjunto de treinamento varias vezes, no código abaixo utilizamos 30000 épocas.

```
>>> import numpy as np
>>> e = 30000 #define o número de épocas
>>> a = 6 #define o número de amostras
>>> t = np.array([21, 20, 18, 29, 23, 23]) #array com atributos Temperatura
>>> u = np.array([91, 70, 70, 85, 80, 70]) #array com atributos Umidade
>>> b = 1 #Define o bias como 1
>>> X = np.vstack((t, u)) # Cria uma matriz com 2 linhas e 6 colunas, onde a
    primeira linha é o atributo temperatura e a segunda linha é o atributo umidade.
     X são os dados de entrada da rede.
>>> Y = np.array([1, 0, 1, 0, 1, 0]) #São as clases a qual pertence os vetores
    colunas da matriz de atributos X.
>>> eta = 0.01 #define a taxa de aprendizado.
>>> W = np.zeros([1, 3]) #define a matriz de pesos iniciais como sendo uma matriz
    nula com 1 linha e 3 colunas
>>> erros = np.zeros(6)
>>> def f(n): #função de ativação
    if n < 0.0:
        return(0)
    else:
        return(1)
>>> for i in range(e): #Rede
    for t in range(a):
        Xb = np.hstack((b, X[:,t]))#insere o numero b=1 no vetor de entrada da rede
    .

        V = np.dot(W, Xb) #Calcula o campo local induzido.

        Ys = f(V) # calcula a saída da rede.

        erros[t] = Y[t] - Ys #calcula o erro da rede

        W = W + eta*erros[t]*Xb #treina a rede, atualizando os pesos.

>>>print("vetor de erros (erros)= " + str(erros))
...vetor de erros (erros)= [0. 0. 0. 0. 0. 0.]
```

Podemos agora, com a rede treinada, fazer a predição se o dia é chuvoso ou ensolarado com base na temperatura e umidade. Por exemplo, considere um dia com temperatura 21C e umidade de 79%.

```
>>>def p(n): #função Predição
    if f(np.dot(W, n)) == 1:
        print("O dia é chuvoso")
    else:
        print("O dia é ensolarado")
>>>q = np.array([1, 21, 79])
>>>p(q)
...O dia é chuvoso
```

4. Redes multicamada

As redes neurais multicamadas, também chamadas de perceptrons de múltiplas camadas (MLP, Multilayer perceptron), tem como característica principal a existência de uma ou mais camadas ocultas. Observe abaixo um diagrama de uma rede multicamada com alimentação para frente.

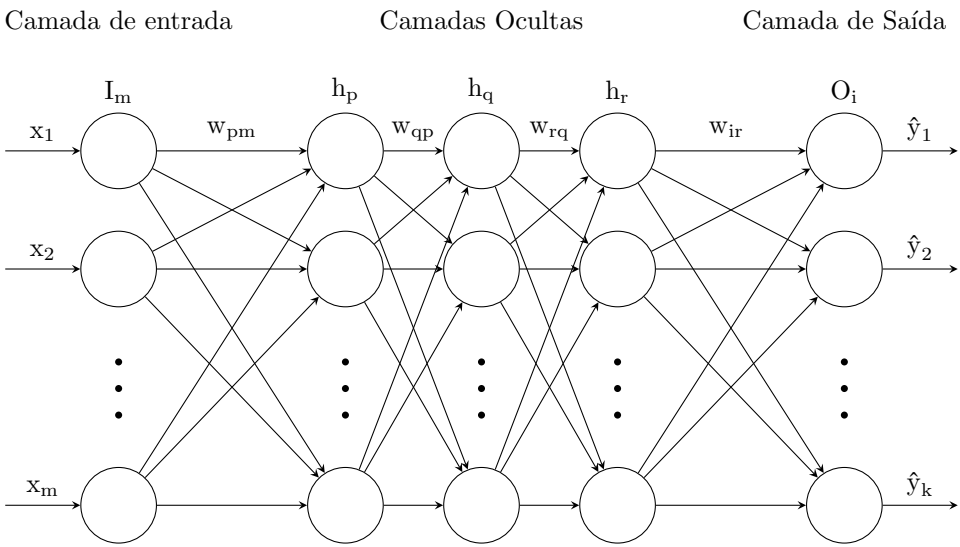


Figura 7: Rede neural artificial
Fonte: Elaborada pelos autores

Observe que a rede neural acima é composta pela camada de entrada I_m, com m entradas; temos três camadas ocultas h_p, h_q e h_r com p, q e r neurônios respectivamente; a camada O_i é a camada de saída da rede com k neurônios. Os pesos w_{pm}, w_{qp}, w_{rq} e w_{ir} são os pesos presentes em cada uma das ligações entre os neurônios de camadas subsequentes.

Vamos construir uma rede neural multicamada; para isso considere o conjunto T de treinamento com k classes e n amostras

$$T = \left\{ \left(\mathbf{x}^{(1)}, \mathbf{y}^{(1)} \right), \left(\mathbf{x}^{(2)}, \mathbf{y}^{(2)} \right), \ldots, \left(\mathbf{x}^{(n)}, \mathbf{y}^{(n)} \right) \right\},$$

onde

- $\mathbf{x}^{(j)} = \left[x_1^{(j)}, x_2^{(j)}, \ldots, x_m^{(j)}\right]$ é um vetor m-dimensional;
- $\mathbf{y}^{(j)} = \left[y_1^{(j)}, y_2^{(j)}, \ldots, y_k^{(j)}\right]$ é um vetor k-dimensional.

Vamos construir e treinar uma rede com a camada de entrada, com m entradas, três camadas ocultas e a camada de saída com k neurônios.

O treinamento de uma rede neural é determinar os pesos sinápticos w_{pm}, w_{qp}, w_{rq} e w_{ir} que fazem as ligações entre as camadas I_m e h_p, h_p e h_q, h_q e h_r, e por fim h_r e O_i, respectivamente, e os bias, b_p, b_q, b_r e b_i de modo que o erro total, medido por uma função de custo, seja mínimo ou satisfatório, isto é, que a rede tenha uma boa precisão nas predições para que foi treinada. Considere que σ_p, ϕ_q, ζ_r e ξ_i são respectivamente as funções de ativação dos neurônios das camadas h_p, h_q, h_r e O_i. Note que cada neurônio pode ter uma função de ativação específica.

Para treinar o perceptron vamos utilizar o termo de bias embutido, isto é, como primeira componente do vetor de pesos, desse modo acrescentaremos 1 como primeira componente do vetor de entrada. Assim, por exemplo, para o p-ésimo neurônio da camada h_p, um vetor de entrada é da forma $\mathbf{x}^{(j)}b = \left[1, x_1^{(j)}, x_2^{(j)}, \ldots, x_m^{(j)}\right]$ e o vetor de pesos é da forma $\mathbf{w}^{(j)} = \left[w_{p0}^{(j)} = b_p, w_{p1}^{(j)}, w_{p2}^{(j)}, \ldots, w_{pm}^{(j)}\right]$.

Para iniciar o treinamento, com algoritmo *Backpropagation*, devemos escolher uma função de custo, utilizaremos o MSE (Mean Squared Error) Erro médio quadrático, amplamente utilizado em aprendizado supervisionado. Seja $\epsilon_i(j)$ o erro do i-ésimo neurônio da camada de saída O_i ao ser inserido na rede o j-ésimo exemplo do conjunto de treinamento, definido por

$$\epsilon_i^{(j)} = y_i^{(j)} - \hat{y}_i^{(j)}. \tag{5}$$

Assim a soma $E^{(j)}$ dos erros de todos os neurônios da camada de saída da rede ao passarmos o j-ésimo exemplo do conjunto de teste é

$$E^{(j)} = \frac{1}{2}\sum_{i=1}^{k} {\epsilon_i^{(j)}}^2 = \frac{1}{2}\sum_{i=1}^{k}\left(y_i^{(j)} - \hat{y}_i^{(j)}\right)^2, \tag{6}$$

onde k é o número de neurônios da camada de saída. Sendo n o número de amostras do conjunto de treinamento, definimos o erro total E_T da rede após a apresentação de todo o conjunto de treinamento como

$$E_T = \frac{1}{n}\sum_{j=1}^{n} E^{(j)}. \tag{7}$$

onde E_T, a função de custo da rede, é uma função de várias variáveis dadas pelos pesos sinápticos e os níveis de bias. A atualização dos pesos ocorre a cada apresentação de um exemplo do conjunto de treinamento da rede, um a um, até a apresentação completa de todo o conjunto de treinamento, isto é, uma época. Por simplificação de notação, omitiremos o índice (j) que indica a apresentação do j-ésimo exemplo do conjunto de treinamento; na rede, por exemplo, utilizaremos $\hat{y}_i$ em vez de $\hat{y}_i^{(j)}$, desde que esteja claro que todos os cálculos abaixo se referem à apresentação do j-ésimo exemplo do conjunto de treinamento.

Para treinar uma rede neural devemos encontrar os pesos e os bias que minimizam a sua função de custo. P ara isso, vamos utilizar os conceitos de máximos e mínimos do cálculo diferencial, através do algoritmo gradiente descendente. Note que

$$\hat{y_i} = \xi\left(u_{O_i}\left(w_{ir}\right)\right), \tag{8}$$

Onde u_{O_i}, dado em função de w_{ir} , é chamado de campo de ativação do i-ésimo neurônio da camada de saída O_i, definido por

$$u_{O_i}(w_{ir}) = \sum_{r=0}^{R} w_{ir} h_r. \tag{9}$$

onde R é o número de neurônios da camada h_r.

Substituindo (8) em (6), temos

$$E = \frac{1}{2}\sum_{i=1}^{k}\left(y_i - \xi\left(u_{O_i}\left(w_{ir}\right)\right)\right)^2. \tag{10}$$

Utilizando a regra da cadeia, podemos derivar a função de custo E com relação aos pesos sinápticos w_{ir}, assim,

$$\frac{\partial E}{\partial w_{ir}} = \frac{\partial E}{\partial \epsilon_i}\frac{\partial \epsilon_i}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial u_{O_i}}\frac{\partial u_{O_i}}{\partial w_{ir}}. \tag{11}$$

Calculando cada uma das derivadas do lado direito da equação (11), obtemos:

$$\begin{aligned}\frac{\partial E}{\partial w_{ir}} &= \epsilon_i \cdot (-1) \cdot \xi'\left(u_{O_i}\right) h_r \\ &= -\epsilon_i \cdot \xi'\left(u_{O_i}\right) h_r.\end{aligned} \tag{12}$$

Assim, a variação do peso w_{ir} pode ser definida como

$$\begin{aligned}\Delta w_{ir} &= -\eta\frac{\partial E}{\partial w_{ir}} \\ &= -\eta\frac{\partial E}{\partial \epsilon_i}\frac{\partial \epsilon_i}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial u_{O_i}}\frac{\partial u_{O_i}}{\partial w_{ir}} \\ &= \eta\epsilon_i \cdot \xi'\left(u_{O_i}\right) h_r,\end{aligned} \tag{13}$$

onde η é a taxa de aprendizado da rede. Observe que o sinal negativo na equação (13) é usado para inverter a direção do gradiente que aponta na direção de crescimento da curva, buscando então a direção para a mudança de peso que reduza o valor do erro da saída da rede. Podemos reescrever a equação (13) da seguinte forma:

$$\Delta w_{ir} = \eta \delta_i h_r, \tag{14}$$

onde δ_i é o gradiente local definido como sendo a derivada parcial do erro E com relação ao campo de ativação u_{O_i}, isto é, $\delta_i = \frac{\partial E}{\partial u_{O_i}} = \epsilon_i \cdot \xi'\left(u_{O_i}\right)$.

A correção de um peso w_{ir} é feita através da seguinte equação:

$$\begin{aligned} w_{ir}(j+1) &= w_{ir}(j) + \Delta w_{ir}(j) \\ &= w_{ir}(j) + 2\eta \delta_i h_r. \end{aligned} \tag{15}$$

Observe que $\Delta w_{ir}^{(j)}$ se aproxima de 0 quando $\frac{\partial E^{(j)}}{\partial w_{ir}^{(j)}}$ se aproxima de 0, isto é, um possível minímo local da função de custo. Desse modo, quanto menor $\frac{\partial E^{(j)}}{\partial w_{ir}^{(j)}}$, menor a taxa de correção dos pesos e, consequentemente, mais o custo da rede se aproxima de um valor desejável.

De modo análogo, a variação dos pesos w_{rq}, w_{qp},w_{pm} é dada pelas seguintes equações:

$$\begin{aligned} \Delta w_{rq} &= -\eta \frac{\partial E}{\partial w_{rq}} \\ &= \eta \zeta'\left(u_{h_r}\right) h_q \sum_{i=1}^{k} \delta_i w_{ir} \\ &= \eta \delta_r h_q, \end{aligned} \tag{16}$$

onde $\delta_r = \zeta'\left(u_{h_r}\right) \sum_{i=0}^{k} \delta_i w_{ir}$;

Onde k é o número de neurônios da camada de saída O_i.

$$\begin{aligned} \Delta w_{qp} &= -\eta \frac{\partial E}{\partial w_{qp}} \\ &= \eta \phi'\left(u_{h_q}\right) h_p \sum_{r=0}^{R} \delta_r w_{rq} \\ &= \eta \delta_q h_p, \end{aligned} \tag{17}$$

onde $\delta_q = \phi'\left(u_{h_q}\right) \sum_{r=0}^{R} \delta_r w_{rq}$; e

onde R é o número de neurônios da camada de saída h_r.

$$\begin{aligned} \Delta w_{pm} &= -\eta \frac{\partial E}{\partial w_{pm}} \\ &= \eta \sigma'\left(u_{h_p}\right) I_m \sum_{q=0}^{Q} \delta_q w_{qp} \\ &= \eta \delta_p I_m, \end{aligned} \tag{18}$$

onde $\delta_p = \phi'\sigma'\left(u_{h_p}\right)\sum_{q=0}^{Q}\delta_q w_{qp}$,

onde Q é o número de neurônios da camada de saída h_q.

Assim, a correção dos pesos w_{rq}, w_{qp},w_{pm} é feita de acordo com as equações abaixo:

$$\begin{aligned} w_{rq}(j+1) &= w_{rq}(j) + \Delta w_{rq}(j) \\ &= w_{rq}(j) + \eta\delta_r h_q. \end{aligned} \tag{19}$$

$$\begin{aligned} w_{qp}(j+1) &= w_{qp}(j) + \Delta w_{qp}(j) \\ &= w_{qp}(j) + \eta\delta_p h_p. \end{aligned} \tag{20}$$

$$\begin{aligned} w_{pm}(j+1) &= w_{pm}(j) + \Delta w_{pm}(j) \\ &= w_{pm}(j) + \eta\delta_p I_m. \end{aligned} \tag{21}$$

4.1. Algoritmo Backpropagation

1. *Inicialização*: Temos que criar os pesos e bias iniciais.

2. *Input* x: Apresente os exemplos de treinamento de uma época à rede neural.

3. *Feedforward*: Para cada neurônio j, da camada l, $l = 2, 3, \ldots, L$, onde L é a profundidade da rede, isto é, o número de camadas da rede, calcular o campo local induzido

$$v_j^l(n) = \sum_{i=0}^{m} w_{ji}^l \cdot y_i^{l-1}(n),$$

onde $y_i^{l-1}(n)$ é o sinal de saída do neurônio i da camada l–1; na apresentação do n-ésimo exemplo de treinamento, w_{ji}^l é o peso do neurônio j da camada l. Observe que, $y_0^{l-1}(n) = +1$ e $w_{j0}^l = b_j^l(n)$ que é o bias do neurônio j da camada l.

Sendo σ a função de ativação, a saída do neurônio j da camada l é

$$\hat{y}_j^l = \sigma(v_j^l(n)).$$

Note que $y_j^0(n) = x_j$, isto é, a j-ésima componente do vetor de entrada x(n). Se o neurônio j pertence à camada de saída $l = L$, então

$$y_j^L(n) = \hat{y}_j,$$

calcular o sinal de erro

$$\epsilon_j(n) = y_j - \hat{y}_j,$$

onde y_j é a j-ésima componente do vetor de rótulos.

4. *Retropropagação*: Para cada $l = L, L-1, L-2, \ldots, 2$, Calcular o gradiente local, definido por:

$$\delta_i^l = \begin{cases} \epsilon_j^L \cdot \phi'\left(v_j^L\right), & \text{se j pertence à camada de saída;} \\ \phi'\left(v_j^l\right) \sum_{i=1}^{n} \delta_i^{l+1} w_{ij}^{l+1}, & \text{se j pertence à camada oculta.} \end{cases}$$

5. *Gradiente descendente*: Para cada $l = L, L-1, L-2, \ldots, 2$ atualize os pesos de acordo com a equação,

$$w_{ji}^l(n+1) = w_{ji}^l(n) + \eta \delta_j^l y_i^{l-1}.$$

Note que os bias também são atualizados por essa equação, visto que, $w_{j0}^l = b_j^l$.

5. Interpretação geométrica da derivada a partir da taxa média de variação

Para que os alunos compreendam como o algoritmo utilizado trabalha para treinar as redes neurais, podemos usar o conceito de derivada a partir de sua interpretação geométrica, isto é, a inclinação da reta tangente a uma curva, o qual é deduzido a partir de aproximações por retas secantes e da taxa de variação das mesmas.

Considere uma curva que seja o gráfico de uma função $y = f(x)$ e $P = (x_0, f(x_0))$, um ponto de f, onde será traçada a reta tangente. Atribuindo a x_0 um acréscimo $\Delta x = h$, temos que a variável $y_0 = f(x_0))$ sofrerá um acréscimo Δy; assim, saímos do ponto $P(x, y)$ para um ponto $Q = (x_0 + \Delta x, y_0 + \Delta y)$. Veja o gráfico abaixo.

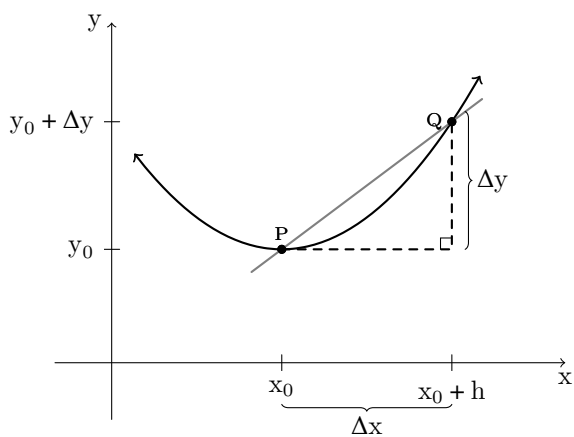


Figura 8: Reta secante
Fonte: [13]

Observe que o coeficiente angular da reta secante $r = \overleftrightarrow{PQ}$ é $m = \frac{\Delta y}{\Delta x}$. Mantendo fixo o ponto P e fazendo Q variar na direção de P, temos que h vai se aproximando de zero, mas não zero. Veja o gráfico abaixo.

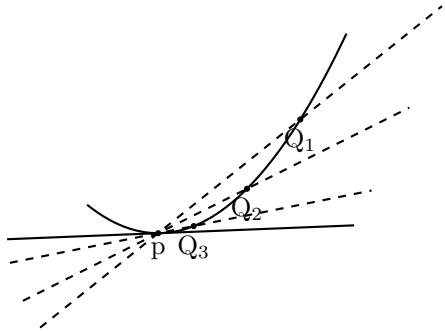


Figura 9: Aproximação de uma reta tangente por retas secantes
Fonte: [13]

Como Q se aproxima de P, mas diferente de P, temos uma posição limite para Q. Podemos definir a reta tangente a f em P, como sendo a reta $r = \overleftrightarrow{PQ}$ quando Q está na sua posição limite e a derivada de f no ponto P é a inclinação ou o coeficiente angular da reta tangente a f em P.

Podemos definir precisamente a reta tangente a uma curva da seguinte forma.

A reta tangente à curva $y = f(x)$ no ponto $P(x_0, f(x_0))$ é a reta passando por P e com inclinação

$$m = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0},$$

desde que o limite exista.

Analogamente podemos definir a derivada como a seguir.

A derivada de uma função $y = f(x)$ em x_0, denotada por $f'(x_0)$, é

$$f'(x_0) = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0},$$

se o limite existir.

Exemplo 3. Considere $y = f(x) = x^2$ e $P = (2, 4)$ um ponto de f e um acréscimo $\Delta x = h$. O acréscimo de y é dado por $\Delta y = (2+h)^2 - 4 = 4 + 4h + h^2 - 4$. Segue que o coeficiente angular da reta secante é

$$t = \frac{\Delta y}{\Delta x} = \frac{4h - h^2}{h}$$

Para determinar a derivada, basta fazer Δx aproximar-se de 0, isto é, fazer h se aproximar de 0, o que implica que a reta secante se aproxima da reta tangente, e, por sua vez, o coeficiente da reta secante aproxima-se do coeficiente da reta tangente. Note que

$$t = \frac{\Delta y}{\Delta x} = \frac{h(4 - h)}{h}.$$

Como h tende a 0, mas nunca é igual a 0, temos:

$$t = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \lim_{h \to 0} \frac{h(4 - h)}{h} = \lim_{h \to 0} (4 - h) = 4.$$

Portanto a derivada de $f(x) = x^2$ em $x = 2$ é 4; isso significa que em $x = 2$ a função f tem uma taxa de crescimento igual a 4, e desse modo, se nosso objetivo é encontrar o mínimo de f, tratando-a como uma função de custo, devemos reduzir x para minimizar f.

6. Redes neurais e a visão computacional

Vamos treinar um modelo de rede neural para reconhecimento de imagens; neste caso, imagens de dígitos manuscritos. Para tal vamos usar o *keras*, o qual é uma API de alto nível para construir e treinar modelos no TensorFlow, que é uma biblioteca completa de código aberto para machine learning. Para usar o TensorFlow; disponível para Ubunto, Windows, macOS e Raspberry Pi; é necessário fazer sua instalação com o gerenciador de pacotes *pip* do Python, usando os códigos abaixo

```
# Requer o pip mais recente
pip install --upgrade pip

# Versão estável atual para CPU e GPU
pip install tensorflow

# Se preferir tente também a compilação de visualização (instável)
pip install tf-nightly
```

Com o Tensorflow instalado, devemos importar as bibliotecas a serem utilizadas para treinar o modelo; neste caso temos o TensorFlow e a API Keras do TensorFlow.

```
>>> import tensorflow as tf #importe tensorflow como tf
>>> from tensorflow import keras #do tensorflow importe keras
```

Além das Bibliotecas auxiliares

```
>>> import numpy as np #importe numpy como np
>>> import matplotlib.pyplot as plt #importe matplotlib.pyplot como plt
```

O próximo passo é importar o conjunto de dados MNIST que se encontra na API keras.

```
>>> (train_images, train_labels), (test_images, test_labels) = tf.keras.datasets.
    mnist.load_data()
```

Com o código acima carregamos o conjunto de dados na forma de quatro NumPy arrays nos seguintes itens:

- Os arrays train_images e train_labels: É o conjunto de treinamento, dividido em duas numpy arrays. A primeira com as classes e a segunda com os rótulos;
- Os arrays test_images e test_labels: É o conjunto de teste configurado da mesma forma do conjuto de treinamento.

No conjunto MNIST, as imagens são arrays NumPy de 28x28, com os pixels variando entre 0 e 255. As labels (rótulos) são arrays de inteiros variando de 0 a 9, onde cada valor tem uma classe que corresponde ao respectivo dígito, conforme a tabela abaixo:

Classe	Label
Zero	0
Um	1
Dois	2
Três	3
Quatro	4
Cinco	5
Seis	6
Sete	7
Oito	8
Nove	9

Figura 10: Tabela de correspondência entre classes e rótulos
Fonte: Elaborada pelos autores

Cada imagem tem apenas um rótulo (label) que representa determinada classe. Como os nomes das classes não estão incluídos no conjunto de dados, podemos armazenar os nomes das classes para uso posterior com o código abaixo.

```
>>> class_names = ['Zero', 'Um', 'Dois', 'Três', 'Quatro',
                   'Cinco', 'Seis', 'Sete', 'Oito', 'Nove']
```

Com os dados importados, podemos, agora, explorar os dados e verificar a sua forma com o comando *.shape*

```
>>> train_images.shape
... (60000, 28, 28)
```

Assim, o conjunto tem 60000 imagens, onde cada imagem é representada por um NumPy array de 28x28

Com o camando *len()* podemos verificar o número de elemento do conjunto *train_labels* que é o conjunto de rótulos do conjunto de treinamento.

```
>>> len(train_labels)
... 60000
```

Do mesmo modo, verificamos as configurações do conjunto de testes

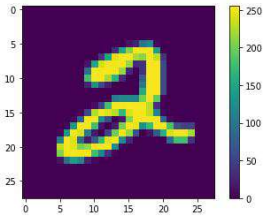
```
>>> test_images.shape
... (10000, 28, 28)
>>> len(test_labels)
... 10000
```

Note que o conjunto dos rótulos é uma numpy array de inteiros, como dito antes; veja.

```
>>> train_labels
... array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

A próxima fase é o pré-processamento dos dados. Se escolhermos uma das imagens do conjunto de treinamento, por exemplo, a número 5, veremos que os pixels estão variando entre 0 e 255

```
>>>plt.figure() #cria uma nova figura ou ativa uma figura existente.
>>>plt.imshow(train_images[5]) #Exibe os dados como uma imagem em 2D.
>>>plt.colorbar() #Adiciona uma barra de cores a um gráfico.
>>>plt.grid(False) #Configura as linhas de grade: Com linhas (True) sem linha (
    False).
>>>plt.show #Mostra a figura
```



Observe, na imagem acima, que a tonalidade dos pixels das imagens do conjunto MNIST varia de 0 a 255. Antes do treinamento do modelo, é conveniente escalar os valores dos pixels entre 0 e 1, para isso, dividimos os valores por 255.0. É importante observar que o conjunto de treinamento e o conjunto de testes devem ser pré-processados da mesma maneira.

```
>>>train_images = train_images / 255.0
>>>test_images = test_images / 255.0
```

Para construir o modelo de uma rede neural devemos, primeiramente, fazer a configuração das camadas e, por fim, compilar o modelo. Como vimos, ao encadear as camadas, o aprendizado é a otimização dos pesos sinápticos, a fim de minimizar a função de custo. Usando *tf.keras* temos modelos prontos que fazem praticamente todo o trabalho automaticamente.

Vamos utilizar, para treinar a rede neural, o modelo sequencial do keras que é apropriado para uma pilha simples de camadas onde cada camada tem exatamente um tensor de entrada e um tensor de saída.

Tensores são arrays multidimensionais, que vão fluindo pelos neurônios ou nós da rede neural.

```
>>>model = keras.Sequential([
...     keras.layers.Flatten(input_shape=(28, 28)),
...     keras.layers.Dense(128, activation='relu'),
...     keras.layers.Dense(10, activation='softmax')
...])
```

A primeira camada da rede neural, *tf.keras.layers.Flatten*, transforma o formato da imagem atual que é um array de duas dimensões (28x28 pixels) em um array de uma única dimensão de (28*28=784 pixels), isto é, enfileira os pixels. Observe que essa camada só tem tal função de formatação dos dados, não tem parâmetros para aprender.

As duas outras camadas *keras.layers.Dense* com 128 e 10 nós (ou neurônios), respectivamente, são full connected (totalmente conectadas). A segunda camada, com 10 nós, retorna um array de 10 probabilidades, onde cada uma delas indica a probabilidade de que a imagem testada pertença a cada uma das 10 classes, que somadas retorna resultado 1.

O próximo passo é compilar o modelo, mas é necessário fazer algumas configurações adicionais.

```
>>>model.compile(optimizer='SGD',
...                loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=
    True),
...                metrics=['accuracy'])
```

Onde:

- *Optimizer* É como o modelo atualiza os pesos de acordo com a função de perda;
- *loss* ou função loss (função de perda) mede a precisão do modelo em cada passo do treinamento, com objetivo de minimizá-lo.
- *Metrics* Usada para monitoramento dos passos de treinamento e teste. No caso de *accuracy*, trata-se da fração das imagens que foram classificadas corretamente.

6.1. Treinamento do modelo

Para treinar uma rede neural artificial, devemos seguir os passos abaixo:

1. Alimentar a rede com os dados de treinamento, que neste caso são as arrays train_imagens e train_labels;
2. Durante o treinamento o modelo aprende a associar as imagens aos respectivos rótulos;
3. São feitas predições com conjunto de teste que neste caso é a array test_imagens;
4. É feita a verificação da precisão das predições com os rótulos do conjunto de teste, que neste caso é a array test_labels.

Para iniciar o treinamento usamos o método *model.fit()*

```
>>> model.fit(train_images, train_labels, epochs=15)#A rede é alimentada com o
    conjunto de treinamento (train_images, train_labels) por 15 épocas.
... loss: 0.2036 - accuracy: 0.9238 #Obtendo 92,38\% de precisão
```

Agora, vamos avaliar o desempenho do modelo com o conjunto de testes.

```
>>> test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2) #
    verbose=2 detalha o processo como uma linha de log por época
...313/313 - 1s - loss: 0.3387 - accuracy: 0.8843
```

Observe que a accuracy teve uma *performance* um pouco menor com o conjunto de teste em relação ao conjunto de treinamento. Essa diferença entre a accuracy do conjunto de treinamento e o conjunto de teste representa um overfitting (sobreajuste).

Como o modelo já está treinado, vamos usá-lo para fazer algumas predições de imagens.

```
>>> probability_model = tf.keras.Sequential([model,
                                             tf.keras.layers.Softmax()])
```

```
>>> predictions = probability_model.predict(test_images)
```

Com este código, estamos usando o modelo treinado para fazer a predição de todas as imagens do conjunto de testes. Como exemplo, podemos verificar qual a décima predição como a seguir:

```
>>> predictions[10]
... array([9.9999809e-01, 8.1870361e-16, 1.9300867e-06, 2.5660890e-11,
       5.4148463e-18, 4.0930523e-10, 2.9532365e-10, 4.9846087e-09,
       7.2152120e-13, 9.8541431e-10], dtype=float32)
```

Observe que a predição é um array com dez números, que indica a confiança do modelo para qual classe a imagem pertence. O maior valor será a classe predita.

```
>>> np.argmax(predictions[10])
... 0
```

Podemos verificar se o modelo acertou a predição, consultando o conjunto *test_labels* com o seguinte código.

```
>>> test_labels[10]
... 0
```

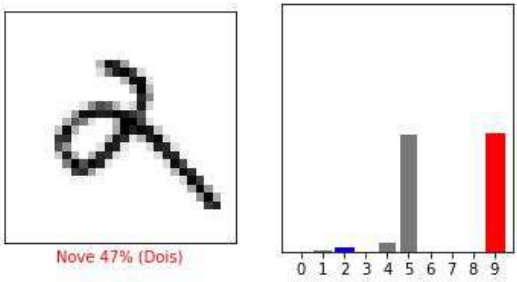
O que mostra que a rede neural fez a previsão correta. Podemos fazer um gráfico de barras indicando as probabilidades de cada classe para uma determinada imagem utilizando os códigos a seguir.

```
>>> def plot_image(i, predictions_array, true_label, img):
...   true_label, img = true_label[i], img[i]
...   plt.grid(False)
...   plt.xticks([])
```

```
...   plt.yticks([])
...   plt.imshow(img, cmap=plt.cm.binary)
...   predicted_label = np.argmax(predictions_array)
...   if predicted_label == true_label:
...     color = 'blue'
...   else:
...     color = 'red'
...   plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
                                  100*np.max(predictions_array),
                                  class_names[true_label]),
                                  color=color)
>>>def plot_value_array(i, predictions_array, true_label):
...   true_label = true_label[i]
...   plt.grid(False)
...   plt.xticks(range(10))
...   plt.yticks([])
...   thisplot = plt.bar(range(10), predictions_array, color="#777777")
...   plt.ylim([0, 1])
...   predicted_label = np.argmax(predictions_array)
...   thisplot[predicted_label].set_color('red')
...   thisplot[true_label].set_color('blue')
```

O código acima introduziu as configurações de plotagem, onde rótulos de predição correta aparecerão na cor azul e os rótulos de predição incorretas aparecerão em vermelho.

```
>>> i = 149
>>>plt.figure(figsize=(6,3))
>>>plt.subplot(1,2,1)
>>>plot_image(i, predictions[i], test_labels, test_images)
>>>plt.subplot(1,2,2)
>>>plot_value_array(i, predictions[i],  test_labels)
>>>plt.show()
```



Com o modelo já treinado e verificado no conjunto de testes, podemos usá-lo para fazer a previsão de uma única imagem específica, seja do conjunto de teste ou uma imagem em pasta. Primeiramente, vamos fazer a previsão de uma única imagem do conjunto de testes. Para isso criamos a variável *img* e armazenamos a segunda imagem do conjunto de testes, além de verificar a sua forma.

```
>>>img = test_images[1]
>>>print(img.shape)
...(28, 28)
```

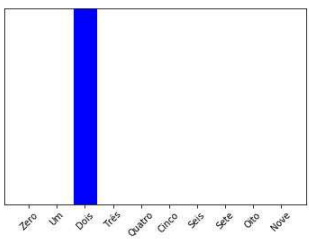
Note que a imagem tem o tamanho de 28x28, mesmo tamanho de entrada da rede. O *tf.keras* é otimizado para fazer previsões em lote de imagens a cada vez. Desse modo temos que criar um lote com apenas uma imagem.

```
>>>img = (np.expand_dims(img,0))
>>>print(img.shape)
...(1, 28, 28)
```

Agora temos um lote com apenas uma imagem de acordo com as configurações de entrada da rede e já podemos prever o rótulo dessa imagem.

```
>>>predictions_single = probability_model.predict(img)
>>>print(predictions_single)
...[[7.6395119e-13 6.5146011e-08 9.9999988e-01 7.3857691e-12 3.4266978e-24
  2.7366560e-11 3.5776493e-11 1.8359487e-18 1.3914259e-09 1.4105835e-19]]
```

```
>>>plot_value_array(1, predictions_single[0], test_labels)
>>>g = plt.xticks(range(10), class_names, rotation=45)
```




```
>>>np.argmax(predictions_single[0])
...2
```

Vamos agora fazer a predição de uma imagem localizada em uma pasta do disco local. Para isso devemos fazer todo o tratamento da imagem de modo que fique compatível com a entrada da rede. O código abaixo importa módulos necessários para o tratamento da imagem

```
>>>from keras.preprocessing.image import ImageDataGenerator, array_to_img,
    img_to_array, load_img
```

O próximo código insere a imagem na variável *img* a partir de um caminho do disco.

```
>>>img2 = load_img('img3.png')
>>>img2
```

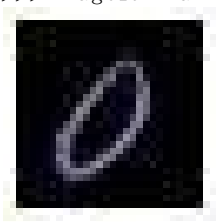
```
>>>print(img.mode)
...RGB #RedGreenBlue
```

Usando o comando *.mode* vemos que o canal de formato de pixel da imagem é RGB. Devemos converter a imagem para o formato $(28, 28)$ e em escala de cinza.

```
>>>from matplotlib import image
>>>from matplotlib import pyplot
```

```
>>>img_resize = img.resize((28,28)) #Redimensiona a imagem para (28, 28).
```

```
>>>image_cinza = img_resize.convert(mode="L") #Converte a imagem em escala de cinza
.
>>>image_cinza
```



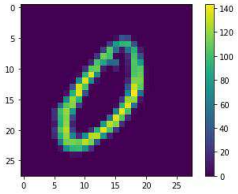
Como a rede foi configurada para fazer previsões em lote de imagens a cada vez, vamos criar um lote com image_cinza.

```
>>>img2 = (np.expand_dims(image_cinza,axis=0))#axis=0 determina a posição onde o
    eixo será inserido
```

```
>>>img2.shape #Retorna a forma da matriz, isto é, uma tupla referente ao número de
    elementos de cada dimensão.
...(1, 28, 28) #Eixo inserido na primeira posição: axis=0
```

Podemos visualizar a imagem com o código abaixo

```
>>>plt.figure()
>>>plt.imshow(img3[0])
>>>plt.colorbar()
>>>plt.grid(False)
>>>plt.show()
```



Note que os valores de pixel estão no intervalo de 0 a 143, devemos redimensionar esses valores para o intervalo de 0 a 1 antes de inserir na rede neural.

```
>>>img3 = img3 / 143.0
```

Agora a imagem está pronta para ser predita pela rede.

```
>>>predict = probability_model.predict(img3)
```

```
>>>predict
...array([[9.4660956e-01, 1.8237397e-04, 4.8826247e-02, 1.5532545e-06,
          1.2159522e-03, 2.3121331e-05, 3.6144556e-04, 2.6822658e-03,
          9.1493923e-05, 5.8982700e-06]], dtype=float32)
```

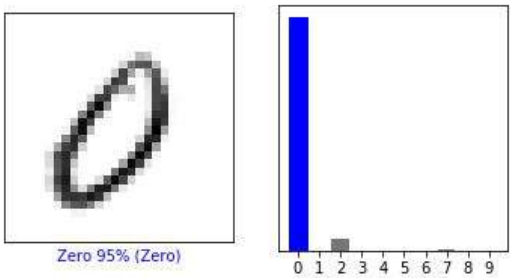
```
>>>np.argmax(predict)#Calcula o maior argumento do array predict
...0
```

Podemos visualizar a previsão através de um gráfico de barras que mostra as probabilidades, como fizemos anteriormente. Em azul a previsão correta e em vermelho as previsões incorretas, para isso temos que criar antes uma numpy array com dtype=uint8 com o rótulo correto para img3.

```
>>>img3_labels = np.array([0,], dtype=np.uint8)
>>>img3_labels
...array([0], dtype=uint8)
```

```
>>>i = 0
>>>plt.figure(figsize=(6,3))
>>>plt.subplot(1,2,1)
>>>plot_image(i, predict[i], img3_labels, img3)
```

```
>>>plt.subplot(1,2,2)
>>>plot_value_array(i, predict[i], img3_labels)
>>>plt.show()
```



7. Considerações Finais

Durante o planejamento e execução de um projeto dessa natureza, o professor encontrará alguns desafios, pois o conteúdo deste projeto abarca uma gama de conceitos novos e com apelo às tecnologias, mas facilmente alcançado através de texto e vídeos na internet. Pontuamos que a BNCC vem exigindo a mudança na praxe e uma adaptação a uma educação científica e tecnológica com base sólida, e, dessa forma, fazem-se necessários textos dessa natureza para auxiliar os professores e indicar possíveis caminhos. Com relação ao uso de tecnologias, devemos considerar que muitas escolas possuem uma infraestrutura de laboratório, e como os *softwares* usados aqui são de livre acesso e com vasto material de tutoriais na internet, acreditamos que tal projeto é exequível e de grande valia na formação de jovens nos variados recantos do país.

Ainda, notamos que é um consenso entre muito educadores que a formação de professores com competência e habilidades para tratar dos temas relevantes e atuais em todas as áreas, em especial a educação em matemática, é um desafio a ser superado e que necessita de muito empenho e diálogo da comunidade. Mas, uma vez atingida essa meta, os professores tornar-se-ão propulsores para o desenvolvimento do conhecimento por meio de metodologias interativas que conduzirão os estudantes a focarem em informações e geração de soluções nos diversos ramos da vida. Com essa perspectiva, as salas de aula tornar-se-ão espaços para a busca e construção de soluções dos problemas pertinentes a nossa sociedade.

Enfim, o ensino sobre redes neurais e aprendizado de máquinas ajuda a amadurecer o conhecimento básico e amplia o desenvolvimento do pensamento computacional. Nessa direção, a BNCC ressalta a importância de se trabalharem as capacidades de *"compreender, analisar, definir, modelar, resolver, comparar e automatizar problemas e suas soluções, de forma metódica e sistemática, por meio do desenvolvimento de algoritmos*", o que caracteriza a necessidade por mais textos com esse objetivo.

Agradecimentos

Os autores gostariam de agradecer o *referee* pela leitura cuidadosa e por todas as sugestões que levaram ao trabalho ser mais preciso e claro em seus tópicos. O primeiro autor é parcialmente financiado pelo CNPq, Brasil, auxílios 308440/2021-8 e 405468/2021-0, e ambos os autores foram parcialmente financiados pela Capes, Brasil, Código de Financiamento 001.

Referências

[1] Ávila, Geraldo. *Limites e Derivadas do Ensino Médio?*. Disponível em: <https://rpm.org.br/cdrpm/60/8.htm>. Acesso em: 02 de agosto de 2020.

[2] Géron, Aurélien. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. 2nd Edition. Sebastopol, CA: O'Reilly, 2019.

[3] Grus, Joel. *Data Science do Zero*. 2ª ed. São Paulo: Novatec, 2014.

[4] Haykin, Simon. *Redes Neurais: Princípios e prática*. 2nd Edition. Porto Alegre: Bookman, 2008.

[5] James, G., Witten, D., Hastie, T., Tibshirani, R. *An Introduction to Statistical Learning*. New York 2013: Springer, 2013 (Corrected at 8th printing 2017).

[6] Matplotlib *Matplotlib Version 3.1.2*. Disponível em: <https://matplotlib.org/3.1.1/index.html>. Acesso em: 18 de Setembro de 2020.

[7] Menezes, Nilo Ney Coutinho. *Introdução à Programação com Python*. 2ª ed. São Paulo: Novatec, 2014.

[8] Metodosupera *Neurônios – glossário do cérebro*. Disponível em: <https://metodosupera.com.br/neuronios-glossario-do-cerebro/>. Acesso em: 12 de Outubro de 2020.

[9] Nielsen, Michael. *Neural Networks and Deep Learning*.Determination Press, 2015.

[10] Python Software Foundation *The Python Tutorial 3.8*. Disponível em: <http://python.org/>. Acesso em: 10 de Setembro de 2020.

[11] Russell, Stuart. *Inteligência Artificial*. Tradução da 3ª Edição. Rio de Janeiro: Elsevier, 2013.

[12] StackExchange *Diagram of an artificial neural network*. Disponível em: <https://tex.stackexchange.com/questions/132444/diagram-of-an-artificial-neural-network>. Acesso em: 12 de Outubro de 2020.

[13] StackExchange *Tikz and Secant Line diagram* . Disponível em: <https://tex.stackexchange.com/questions/460632/tikz-and-secant-line-diagram>. Acesso em: 15 de Outubro de 2020.

[14] StackExchange *A diagram about partial derivatives of f(x,y)* . Disponível em: <https://tex.stackexchange.com/questions/479814/a-diagram-about-partial-derivatives-of-fx-y>. Acesso em: 18 de Outubro de 2020.

[15] TensorFlow *Treinamente de Redes Neurais*. Disponível em: <https://www.tensorflow.org/>. Acesso em: 25 de Setembro de 2020.

Márcio Batista
Universidade Federal de Alagoas
<mhbs@mat.ufal.br>

André Oliveira Martins
Universidade Federal de Alagoas
<andre.marttins7@gmail.com>

Recebido: 04/09/2021

SBM
SOCIEDADE BRASILEIRA DE MATEMÁTICA

Publicado: 05/09/2022