

Apostila de Machine Learning

Por William Ludovico Homem
PET Engenharia Mecânica Ufes



UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
PROGRAMA DE EDUCAÇÃO TUTORIAL ENGENHARIA MECÂNICA

WILLIAM LUDOVICO HOMEM

APOSTILA DE MACHINE LEARNING

VITÓRIA

2020

SUMÁRIO

1. INTRODUÇÃO.....	4
2. PYTHON E SUAS BIBLIOTECAS	4
2.1. GOOGLE COLABORATORY	5
2.2. SINTAXE E ESTRUTURAS DE DADOS.....	6
2.3. BIBLIOTECAS	12
2.3.1. NUMPY.....	13
2.3.2. MATPLOTLIB	15
3. APRENDIZAGEM DE MÁQUINA	18
3.1. CONCEITOS GERAIS	19
3.2. DESCIDA DO GRADIENTE	22
3.3. REDES NEURAIS	26
3.3.1. BACKPROPAGATION.....	30
4. ESTUDO DE CASO: DADOS MNIST	32
4.1. PRÉ-PROCESSAMENTO E ANÁLISE EXPLORATÓRIA.....	32
4.2. CRIAÇÃO E TREINAMENTO DO MODELO	37
5. REFERÊNCIAS BIBLIOGRÁFICAS	46

1. INTRODUÇÃO

O curso de Machine Learning tem o objetivo de apresentar os conceitos básicos por trás do uso de Aprendizagem de Máquina por meio da construção de uma aplicação que será usada para identificar dígitos manuscritos. Para tanto, toda a programação será auxiliada por bibliotecas escritas em Python, como Numpy, Matplotlib e TensorFlow.

As aulas, bem como essa apostila, serão divididas em dois grandes módulos. O primeiro será voltado para apresentação da linguagem Python, sua sintaxe e suas estruturas. Vale ressaltar, entretanto, que será feito apenas um apanhado geral da linguagem. O objetivo dessa seção é apresentar o conteúdo básico que será usado no estudo de caso.

No segundo módulo, por sua vez, será criado um algoritmo capaz de identificar números escritos à mão, oriundos do MNIST *Dataset*. No decorrer do desenvolvimento, serão apresentadas as estruturas básicas presentes em redes *feedforward*, assim como o seu funcionamento.

Espera-se que, ao término do curso, o congressista tenha ciência da filosofia por trás dessa área da Inteligência Artificial e possa, posteriormente, ter uma aprendizagem facilitada sobre modelos mais complexos e usuais, além de trabalhar em seus próprios projetos e banco de dados.

2. PYTHON E SUAS BIBLIOTECAS

Antes de discutir sobre Aprendizagem de Máquina, é necessária a definição de uma linguagem de programação para que o modelo possa ser escrito e processado por um computador. Dentre as muitas existentes, opta-se pela utilização da linguagem Python, por ser amplamente utilizada no campo de Ciência de Dados e *Machine Learning*, graças ao advento de bibliotecas como Pandas, TensorFlow e Keras.

Python foi concebido no final de 1989 por Guido van Rossum e se popularizou por ser uma das primeiras linguagens a trazer o conceito de Orientação a Objeto. Com uma filosofia simplista e explícita, essa linguagem *open source*, hoje, é classificada como uma linguagem de alto nível, interpretada, imperativa, orientada a objetos e de tipagem dinâmica e forte.

“Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
*Although never is often better than *right* now.*
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!" (PETERS, 2004)

Esse capítulo, então, tem como objetivo ilustrar a sintaxe e as estruturas básicas da linguagem Python, visando, assim, um maior entendimento do congressista no que se refere aos assuntos dos capítulos posteriores.

2.1. GOOGLE COLABORATORY

Conhecido, também, como Google Colab, o Google Colaboratory é um ambiente de programação Jupyter que permite que o usuário crie programas escritos em Python diretamente do navegador, sem nenhuma configuração prévia ou instalação de bibliotecas famosas. Além disso, a plataforma permite uma interação direta com o Google Drive e oferece o requerimento de processamento paralelo de GPU's ou TPU's sem a necessidade de posse desses *hardwares*.

Por essas funcionalidades e facilidades, o curso de *Machine Learning* e essa apostila utilizará esse ambiente. Entretanto, o Google Colab será tratado como uma ferramenta e, portanto, esse texto não abordará especificidades da plataforma.

2.2. SINTAXE E ESTRUTURAS DE DADOS

Em Python, tudo é objeto. Dessa forma, todos os dados são estruturados de forma a possuir função e característica. Embora a Orientação a Objeto não se enquadre na abordagem desse curso, é muito aconselhável que o leitor se inteire sobre o assunto para que o entendimento da linguagem se torne mais orgânico.

Embora a afirmação feita no parágrafo acima seja generalizada, os dados podem possuir classificações distintas. Os valores numéricos, por exemplo, são divididos em inteiros (*integer* ou *int*) e flutuantes (*float*).

Quadro 1: Variáveis Numéricas.

```
num_int = 10
num_float = 10.1

print(f'As variáveis "num_int" e "num_float" são,
respectivamente, do tipo {type(num_int)} e
{type(num_float)}.')
```



```
As variáveis "num_int" e "num_float" são,
respectivamente, do tipo <class 'int'> e <class 'float'>.
```

Fonte autoral.

O Quadro 1 ilustra a saída de uma célula do Google Colab. Observe que os valores “10” e “10.1” foram atribuídos a duas variáveis distintas que, posteriormente, foram utilizadas na função “print”, que imprime uma mensagem na tela. Essa função, por sua vez, requer como parâmetro alguma estrutura de texto (*string*) que é definida entre aspas simples ou duplas.

Note, entretanto, que, antes da *string*, foi acrescentada a letra “f”. Esse acréscimo é um atalho para o uso do método “.format” utilizado para manipular dados do tipo *string* e, nesse caso, permitiu que fosse acrescentado ao texto os valores das variáveis declaradas, inclusas no texto entre chaves. Um outro exemplo da utilização do método “.format” é ilustrado no Quadro 2.

Ainda se tratando do Quadro 1, a função “type” foi utilizada para que fosse retornado o tipo das variáveis declaradas. O retorno da função demonstra que ambas são objetos (*class*), mas um valor é do tipo inteiro e o outro flutuante.

Quadro 2: Método “.format”.

```
nome = 'William' # A variável "nome" carrega o valor de
uma string
```

```
print('Olá, {}'.format(nome)) # Método ".format". A
variável nome irá assumir a posição das chaves.
print(f'Olá, {nome}.') # Método abreviado.
```

↳

```
Olá, William.
Olá, William.
```

Fonte Autoral

O conjunto de valores que em outras linguagens é conhecido como vetor, ou matriz, possui um paralelo com as listas, em Python. A distinção no nome vem a calhar uma vez que as listas permitem a junção de valores distintos, o que não é possível em muitas outras linguagens de programação. Sua declaração e exemplificação podem ser estudadas por intermédio do Quadro 3.

Quadro 3: Listas.

```
lista_numerica = [1,2,3,4,5,6]
lista_misturada = ['William', '24']

print(f'Lista numérica: {lista_numerica}')
print(f'Lista misturada: {lista_misturada}')

# A indexação das listas se dá por meio dos colchetes.
nome = lista_misturada[0]
idade = lista_misturada[1]

print(f'Bem vindo, {nome}. Você tem, realmente, {idade}
anos?')
```

↳

```
Lista numérica: [1, 2, 3, 4, 5, 6]
Lista misturada: ['William', '24']
Bem vindo, William. Você tem, realmente, 24 anos?
```

Fonte autoral.

Enquanto as listas podem ser modificadas e manipuladas (métodos “.append”, “.remove” e “.sort”, por exemplo), há, ainda, conjuntos de dados imutáveis: as tuplas. As tuplas são definidas entre parênteses e não admitem, como comentado, nenhuma manipulação, como ilustra a Quadro 4.

Quadro 4: Tuplas.

```
# Listas
lista = [4,1,2,4]
print(f'Lista desordenada: {lista}.')

lista.sort()
```

```

print(f'Lista ordenada: {lista}.')

print('\n')

# Tuplas
tup = (4,1,2,4)
print(f'Tupla: {tup}.')

tup.sort()

```

```

↳
Lista desordenada: [4, 1, 2, 4].
Lista ordenada: [1, 2, 4, 4].

Tupla: (4, 1, 2, 4).
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-11-e28a6babb39a> in <module>()
     12 print(f'Tupla: {tup}.')
     13
--> 14 tup.sort()

AttributeError: 'tuple' object has no attribute 'sort'

```

Fonte autoral.

Visto alguns dos muitos tipos de dados contidos na linguagem Python, é natural a abordagem dos operadores. A Tabela 1 e 2 mostram os operadores matemáticos e lógicos, respectivamente, mais comumente utilizados.

Tabela 1: Operadores Matemáticos.

Operação	Resultado
$x + y$	Soma de x e y
$x - y$	Subtração de x e y
$x * y$	Multiplicação de x por y
x/y	Divisão de x por y
$x//y$	Quociente flutuante de x por y
$x\%y$	Resto de x/y
$complex(re, im)$	Número complexo com parte real igual a re e parte imaginária igual a im
$x ** y$	Potência de x por y

Fonte: docs.python.org

Tabela 2: Operadores Lógicos.

Operação	Significado
<	Menor
>	Maior
<=	Menor ou igual
>=	Maior ou igual
==	Igual
!=	Diferente
<i>x or y</i>	Se <i>x</i> ou <i>y</i> forem verdadeiros, retorna verdadeiro
<i>x and y</i>	Se <i>x</i> e <i>y</i> forem verdadeiros, retorna verdadeiro
<i>not x</i>	Se <i>x</i> for verdadeiro, retorna falso. Se <i>x</i> for falso, retorna verdadeiro.

Fonte: docs.python.org

Tendo o conhecimento de como operar e manipular os dados de forma matemática e lógica, pode-se definir estruturas mais robustas e úteis. As estruturas condicionais, por exemplo, são de grande utilidade e podem ser facilmente criadas pela palavra reservada “if”, como ilustra o Quadro 5. Note que após a condição há a pontuação “dois pontos”, sinalizando a consequência da condicional. Essa, por sua vez, deve ficar recuada à esquerda por um “tab”; a indentação em Python é parte de sua sintaxe.

Quadro 5: Estrutura Condicional.

```

num = input('Digite um número:\n') # Função que registra
um valor do usuário
num = int(num) # Transforma o valor em um inteiro

if num < 0:
    print('O fatorial de um número negativo não é definido.
')
elif num == 0:
    fat = 1
else:
    pass

```

☞

Digite um número:

5

Fonte autoral.

Note que no exemplo há a utilização de duas funções ainda não apresentadas: “input” e “int”. A primeira recebe um registro feito pelo usuário por meio do teclado e a segunda transforma aquele valor em um inteiro (as funções “float” e “string” possuem fins similares). Há, ainda, junto à estrutura condicional, o “elif”, que é verificado apenas após a verificação do “if”, e o “else”, que é executado apenas se as condições não forem satisfeitas.

Pode-se observar no Quadro 5 que o código ali contido tem o objetivo de calcular o fatorial de um número. É necessária, então, a inserção de um código no bloco “else” que, de fato, calcule essa operação. Para tanto, pode-se utilizar a estrutura de repetição “for”.

Quadro 6: Estrutura de Repetição “for”.

```
num = input('Digite um número:\n') # Função que registra
um valor do usuário
num = int(num) # Transforma o valor em um inteiro
fat = 1 # Inicialização da variável "fat"

if num < 0:
    print('O fatorial de um número negativo não é definido.
    ')
elif num == 0:
    fat = 1
else:
    for i in range(1, num + 1):
        fat *= i

print(f'O fatorial de {num} é {fat}.')
```

↳

```
Digite um número:
5
O fatorial de 5 é 120.
```

Fonte autoral.

O Quadro 6 traz, além da estrutura “for”, um tipo de dado chamado “range”, que organiza um conjunto de números igualmente espaçados e pertencentes ao conjunto $[n^{\circ} \text{ inicial}, n^{\circ} \text{ final})$. A linha “*fat *= i*” é uma forma abreviada de se escrever “*fat = fat * i*”. Há, além do “for”, uma outra estrutura de repetição comum chamada “while”, ilustrada no Quadro 7.

Quadro 7: Estrutura de Repetição “while”.

```
num = 1
```

```
while num != 0:
    num = int(input('Digite 0:\n'))

print("Programa encerrado")
```

```
↳
Digite 0:
3
Digite 0:
2
Digite 0:
1
Digite 0:
0
Programa encerrado
```

Fonte autoral.

Durante a escrita do código, é aconselhável o uso de funções, possibilitando, assim, uma escrita mais limpa e resumida. Em Python, cria-se funções por meio da estrutura “def”, como bem ilustrado no Quadro 8. Note que, junto ao nome da função, declara-se os parâmetros da mesma. Vale ressaltar, também, que não é necessário que a função retorne um valor.

Quadro 8: Função.

```
def fatorial(num):
    fat = 1 # Inicialização da variável "fat"

    if num < 0:
        raise Exception('O fatorial de um número negativo não
é definido.')
    elif num == 0:
        fat = 1
    else:
        for i in range(1, num + 1):
            fat *= i

    return fat

num = 5
print(f'O fatorial de {num} é {fatorial(num)}.')
```

```
↳
O fatorial de 5 é 120.
```

Fonte autoral.

2.3. BIBLIOTECAS

Python é uma linguagem de programação gratuita e *open source*, o que favorece a criação de bibliotecas e plataformas que podem ser aplicadas em diversas áreas tecnológicas. Algumas dessas bibliotecas são amplamente conhecidas e usadas no desenvolvimento de modelos de Aprendizagem de Máquina, tais como Scikit-Learn e TensorFlow. O presente capítulo apresentará brevemente as bibliotecas Numpy e Matplotlib. As demais bibliotecas que serão utilizadas serão abordadas ao longo do texto.

Todavia, antes de se utilizar essas bibliotecas, é necessária a importação das mesmas. Para tanto, existem diferentes maneiras de se realizar tal operação, na qual cada uma possui suas particularidades. O modo mais simples se baseia na importação de todos os módulos de uma determinada biblioteca por meio do comando “from [nome da biblioteca] import *”, em que o asterisco significa *all*. Há, também, a possibilidade de importar apenas um módulo específico, como ilustra o Quadro 9.

Quadro 9: Importações de Bibliotecas.

<pre>from math import * sqrt(4)</pre>
↳ 2.0
<pre>from math import sqrt sqrt(9)</pre>
↳ 3.0

Fonte autoral.

Imagine, entretanto, que a importação da biblioteca “math”, biblioteca matemática nativa do Python, faça parte de um grande desenvolvimento que faça uso de muitos módulos. Pode acontecer de que mais de uma biblioteca tenha a função “sqrt” e, portanto, haveria uma divergência no que tange seu uso. Para evitar tal problema, pode-se especificar, durante a chamada, a biblioteca em que um determinado módulo faça parte, como bem exemplifica o Quadro 10.

Quadro 10: Modos de importação.

<pre>import math math.sqrt(4)</pre>
--

↳ 2.0
<pre>import math as m # Importa a biblioteca "math", apelidando-a de "m" m.sqrt(9)</pre>
↳ 3.0

Fonte autoral.

2.3.1. NUMPY

Numpy é um pacote matemático para uso de computação científica em Python. Suas funções robustas e otimizadas para Álgebra Linear e o seu uso baseado em matrizes n-dimensionais, faz com que essa biblioteca seja uma das mais utilizadas em sua área de atuação.

Para sua importação, a própria documentação oficial do pacote sugere que seja feita de forma a “apelidar” a biblioteca como “np”. Dessa forma, a leitura e escrita do código se torna mais fácil e diminui a ocorrência de eventuais erros quanto a chamada de funções. O Numpy traz uma série de novos objetos para o Python, incluindo novos tipos de dados. Um dos mais importantes, por ser a base de toda a biblioteca, é o Array. O Array, muito parecido com as matrizes matemáticas, é uma rede de dados em que os itens são todos do mesmo tipo, diferentemente das listas. Sua criação e indexação, embora possam parecer um pouco mais complicadas, continuam simples, como ilustrado no Quadro 11.

Quadro 11: O *array* do Numpy.

<pre>import numpy as np # Criação de uma matriz qualquer M = np.array([[1,2,3],[4,5,6],[7,8,9]]) print(f'Matriz:\n{M}.\n') # Acesso à primeira linha print(f'Primeira linha:\n{M[0]}.\n') # Acesso à primeira coluna (O atributo ".T" retorna a matriz M transposta) print(f'Primeira coluna:\n{M.T[0]}.\n') # Acesso ao item 2x2 print(f'Item 2x2:\n{M[1,1]}.'.')</pre>
--



Matriz:

```
[[1 2 3]
 [4 5 6]
 [7 8 9]].
```

Primeira linha:

```
[1 2 3].
```

Primeira coluna:

```
[1 4 7].
```

Item 2x2:

```
5.
```

Fonte autoral.

Em Numpy, é muito fácil realizar operações algébricas laboriosas, como a solução de um sistema linear, uma multiplicação de matrizes ou até mesmo a obtenção de autovalores e autovetores, operações ilustradas no Quadro 12. A determinação dos autovetores e autovalores, por sua vez, se dá por meio do retorno de um *array* cuja a primeira posição se refere aos autovalores e a segunda posição aos autovetores normalizados.

Quadro 12: Operações algébricas com Numpy.

```
# Criação da matriz "A" e do vetor "b"
A = np.array([[1,2,-3],[3,-1,2],[2,1,1]])
b = np.array([[1,0,2]])

# Multiplicação b*A
mult = np.dot(b, A)
print(f'b*A = {mult}.\n')

# Solução do sistema Ax = b
x = np.linalg.solve(A, b.T)
print(f'A solução do sistema Ax = b é:\n{x}.\n')

# Determinação dos autovetores e autovalores da matriz A
eig = np.linalg.eigh(A)
print(f'Autovalores: {eig[0]}.')
print(f'Autovetores:\n{eig[1]}.'
```



```
b*A = [[ 5  4 -1]].
```

A solução do sistema $Ax = b$ é:

```
[[0.0625]
 [1.3125]
 [0.5625]].
```

```
Autovalores: [-3.19963827 -0.34393147  4.54356975].
Autovetores:
[[-0.61035896 -0.36717764 -0.70188498]
 [ 0.78531074 -0.39646334 -0.47550379]
 [ 0.1036773  0.84142581 -0.5303335 ]].
```

Fonte autoral

2.3.2. MATPLOTLIB

Trabalhar e manipular um grande conjunto de dados é a chave para a criação de modelos de Aprendizagem de Máquina funcionais. Enquanto a “máquina” possui uma certa facilidade em interpretar os dados de forma numérica, o ser humano, normalmente, possui mais familiaridade com uma interpretação visual. Dessa forma, a apresentação desses dados de forma gráfica se mostra uma etapa importante no desenvolvimento desses modelos.

Para tanto, o minicurso e essa apostila utilizarão o Matplotlib, capaz de oferecer ferramentas de plotagem bi e tridimensional. O presente capítulo apresentará essa biblioteca por meio da construção de um gráfico 2-D da função Sigmoid, muito utilizada como função de ativação na construção de Redes Neurais.

$$\sigma = \frac{1}{1 + e^{-x}} \quad (\text{Eq. 1})$$

Assim como muitas outras ferramentas matemáticas, o Matplotlib utiliza uma impressão baseada em pontos coordenados. Dessa forma, se mostra necessária a criação de *arrays* que representem o domínio e a imagem da função determinada pela Eq. 1. Para a determinação do domínio, utiliza-se, como ilustrado no Quadro 13, a função “linspace”, presente no Numpy, que cria um conjunto numérico igualmente espaçado. Essa função, por sua vez, recebe como parâmetros o valor inicial, o valor final e o número de itens contidos no *array*. Após isso, aplica-se a Eq. 1 à variável “x”, ponto a ponto.

Feito isso, a plotagem gráfica se segue de forma muito simples. Ao chamar as funções “plot” e “show” contidas em “matplotlib.pyplot”, um gráfico sem configuração é retornado.

Quadro 13: “Plotagem” da função Sigmoid.

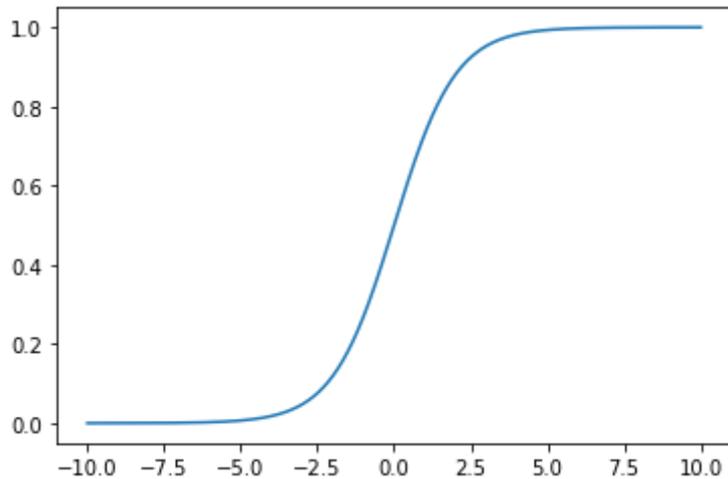
```
import matplotlib.pyplot as plt

# Criação do domínio
x = np.linspace(-10, 10, 1000)

# Criação da imagem
y = 1/(1 + np.exp(-x))
```

```
# Plotagem
plt.plot(x, y)
plt.show()
```

↳



Fonte autoral.

Às vezes, é desejado plotar mais de um gráfico em uma mesma figura. Para tanto, basta plotar a outra função logo após a primeira. Como forma de diferenciar essas funções, pode-se estilizar os traçados e acrescentar uma legenda, acréscimos exemplificados pelo Quadro 14.

Quadro 14: Estilização gráfica.

```
# Configuração do tamanho da imagem
plt.figure(figsize = (7,5))

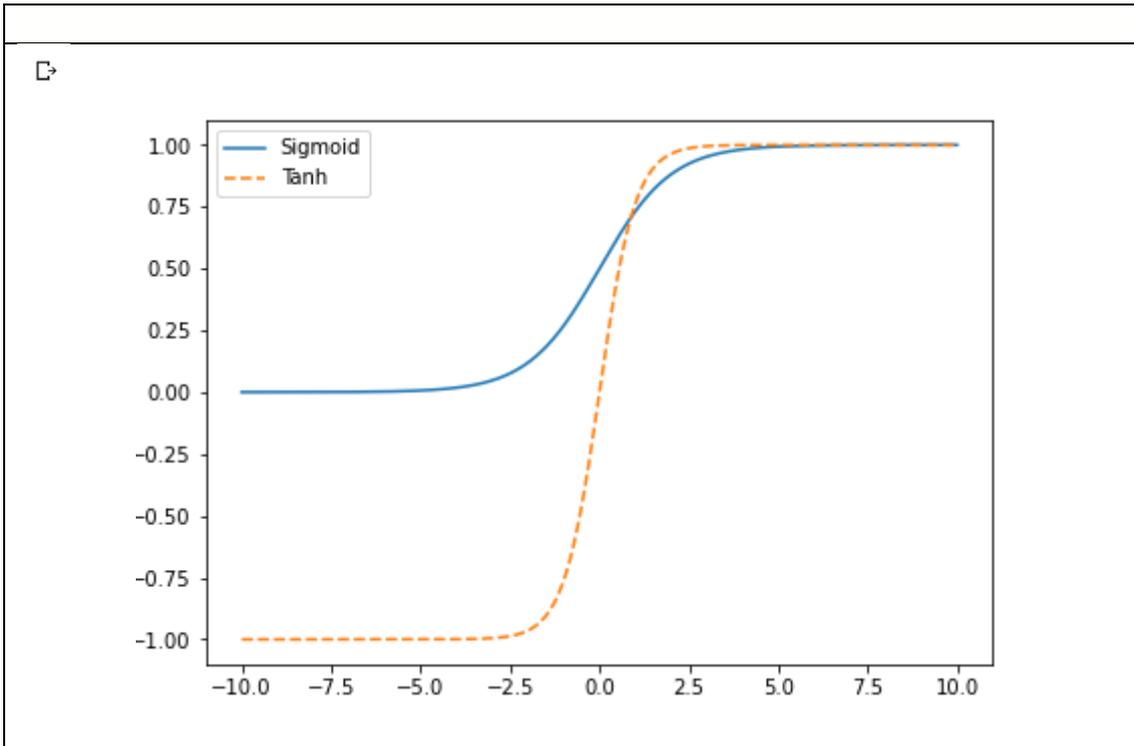
# Criação do domínio
x = np.linspace(-10, 10, 1000)

# Criação da imagem
sig = 1/(1 + np.exp(-x))
tanh = np.tanh(x)

# Plotagem da função Sigmoid
plt.plot(x, sig, label = 'Sigmoid')

# Plotagem da função Tanh
plt.plot(x, tanh, '--', label = 'Tanh')

plt.legend()
plt.show()
```



Fonte autoral

Se, entretanto, seja necessário “plotar” esses gráficos de forma separada, pode-se utilizar a função “subplot”, que recebe como parâmetros a quantidade de linhas, a quantidade de colunas e a identificação da figura, como ilustra o Quadro 15. Foram acrescentados, também, os atributos “title”, “xlabel” e “ylabel” para definir, respectivamente, os títulos da figura e dos eixos “X” e “Y”.

Quadro 15: Subplot.

```
import matplotlib.pyplot as plt

# Configuração do tamanho da imagem
plt.figure(figsize = (7,5))

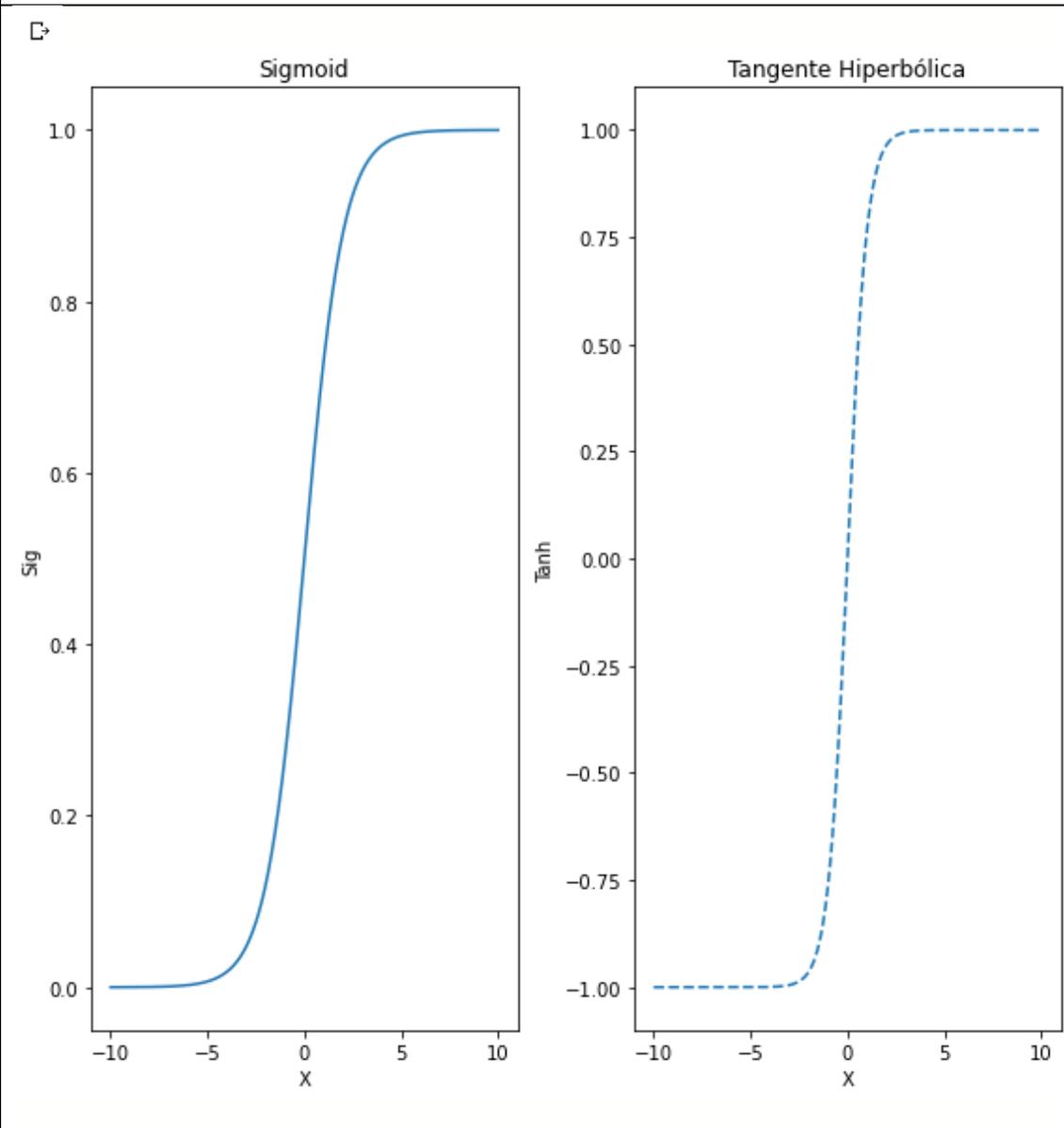
# Criação do domínio
x = np.linspace(-10, 10, 1000)

# Criação das imagens
sig = 1/(1 + np.exp(-x))
tanh = np.tanh(x)

# Plotagem da função Sigmoid
plt.subplot(1,2,1)
plt.title("Sigmoid")
plt.xlabel('X')
plt.ylabel('Sig')
plt.plot(x, sig)
```

```
# Plotagem da função Tanh
plt.subplot(1,2,2)
plt.title('Tangente Hiperbólica')
plt.xlabel('X')
plt.ylabel('Tanh')
plt.plot(x, tanh, '--')

plt.tight_layout()
plt.show()
```



Fonte autoral.

3. APRENDIZAGEM DE MÁQUINA

Você, por acaso, já se perguntou como que os carros autônomos conseguem identificar obstáculos? Ou, então, como o Facebook descobre que você está na foto de um amigo?

A Aprendizagem de Máquina é classificada como uma área da Inteligência Artificial e é a resposta para essas perguntas. Seu objetivo é a criação de códigos capazes de observar e aprender padrões em dados observacionais.

O *Machine Learning* (ML) é o campo da ciência que fornece ao computador a habilidade de aprender uma determinada tarefa sem ser explicitamente programada. Uma criança aprende a identificar um brinquedo, por exemplo, uma vez que a mesma é exposta ao objeto várias vezes e, assim, consegue identificar padrões que o caracteriza. Um modelo de aprendizagem, por sua vez, é exposto a um conjunto de dados robusto e utiliza os exemplos ali contidos para identificar padrões relevantes à tarefa que lhe é destinado.

Existem vários paradigmas de programação robustos no estudo dessa área; uma das mais difundidas atualmente é a Rede Neural, que será abordada mais à frente. Esta, por sua vez, é baseada no funcionamento biológico de um neurônio e tenta simular a propagação de informação entre neurônios por meio das sinapses neurais.

O desenvolvimento desse campo da Inteligência Artificial possibilitou, nos últimos anos, a criação de diversas ferramentas e técnicas, em diversas áreas do saber, que auxiliam na realização de tarefas até então muito custosas, como na pós renderização existente na *Deep Learning Super Sampling* contida nas placas de vídeo mais atuais da Nvidia. Na Engenharia Mecânica, o uso de modelos de ML, embora um pouco restrito, pode ser observado nas áreas de manufatura, manutenção preditiva, análise mineral etc.

Essa apostila apresentará os conceitos básicos da Aprendizagem de Máquina por meio da construção de um modelo capaz de identificar dígitos manuscritos contidos no banco de dados MNIST (*Modified National Institute of Standards and Technology database*), problema amplamente abordado e comum à introdução do tema.

3.1. CONCEITOS GERAIS

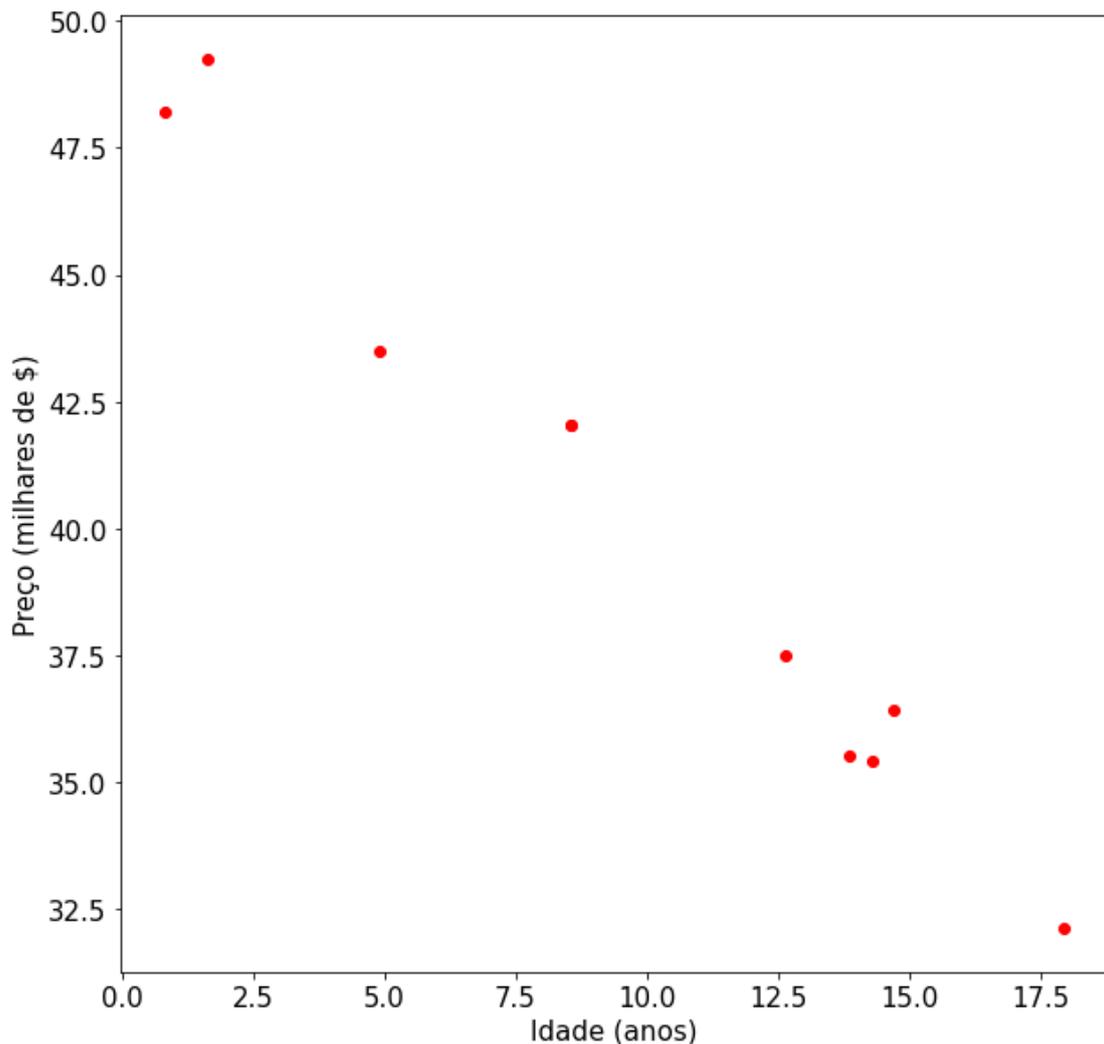
Dentre os muitos tipos de problemas que podem ser abordados por modelos de ML, duas distinções são habituais e comuns: os problemas de regressão e os de classificação. O primeiro tem como objetivo prever um valor numérico pertencente a um intervalo contínuo, como na determinação do valor de uma casa, baseando a escolha com base na sua idade, tamanho, entre outros parâmetros. Problemas de classificação, por outro lado, escolhem um rótulo dentre os existentes, como, por exemplo, na identificação de um gato em uma imagem.

Ambos problemas, assim como outros existentes, se baseiam em uma mesma premissa, já citada anteriormente: aprender padrões relevantes ao problema por meio da observação

de dados de forma a permitir a solução do mesmo, sem que haja uma programação explícita. Mas como modelar a identificação e aprendizagem desses padrões?

Imagine que você receba um banco de dados que contenha o preço e a idade das casas de um bairro ao norte da Serra, no estado do Espírito Santo. Por ser um bairro pacato, espera-se que os preços das residências não variem muito com a localidade da mesma. Além disso, a maior parte dos imóveis possuem um tamanho médio, tornando, assim, a idade um parâmetro relevante na determinação do valor destes.

Figura 1: Dados relativos às casas



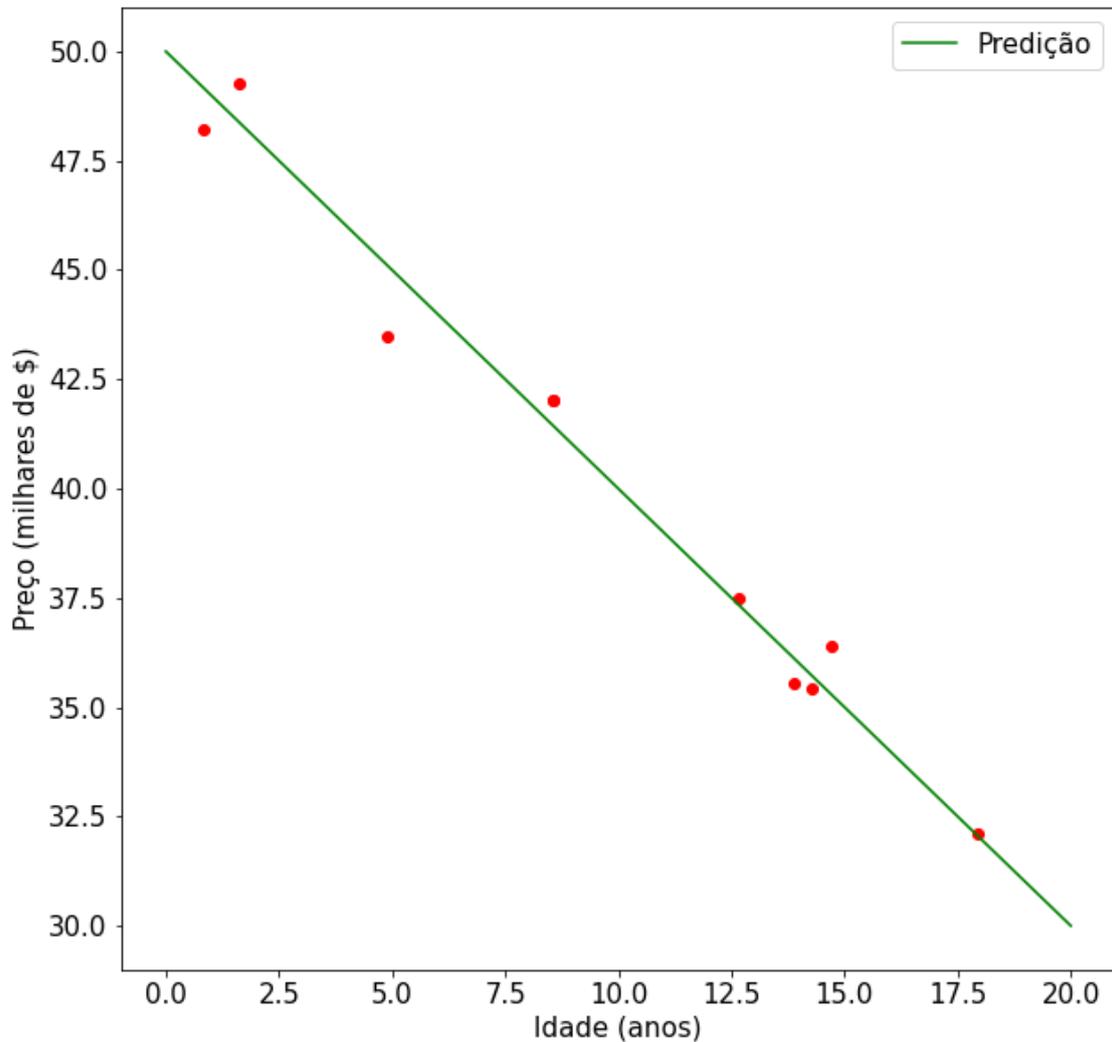
Fonte autoral.

Pode-se observar na Figura 1 que a relação entre a idade e o valor da casa pode ser satisfatoriamente aproximada por uma reta, como ilustra a Figura 2. Dessa forma, pode-se modelar o preço do imóvel pela equação de uma reta, como ilustra a Equação 2. Essa abordagem se chama Regressão Linear.

$$\hat{y} = wx + b, \quad (\text{Eq. 2})$$

onde os coeficientes da reta “ w ” e “ b ” são chamados, comumente, de peso e *bias*.

Figura 2: Regressão Linear



Fonte autoral.

Modelado o problema, basta determinar os coeficientes da reta para que se possa prever o preço de uma residência com base na sua idade. Ressalta-se, entretanto, que, embora no exemplo introdutório apenas uma variável de entrada seja utilizada, em um problema real se utiliza várias entradas e, dessa forma, é necessária a determinação de vários pesos.

A determinação dos coeficientes, que acontece na etapa de aprendizagem, se baseia no treinamento de pesos e *bias* que satisfaçam a reta ilustrada na Figura 2. Para tanto, se mostra necessária a definição de uma métrica que aponte o quão satisfatórios são os coeficientes, um erro.

Esse erro é chamado de Função de Custo e penaliza uma predição ruim. Existem muitas funções de custo, cada uma voltada para uma determinada aplicação, mas a mais comum

em problemas de regressão, e útil em várias aplicações, é o Erro Médio Quadrático (MSE), definido pela Equação 3.

$$MSE = \frac{1}{N} \sum (y - \hat{y})^2 \quad (\text{Eq. 3})$$

onde “ N ” é o número de entradas, “ y ” é o valor correto almejado e “ \hat{y} ” é a predição feita pelo modelo.

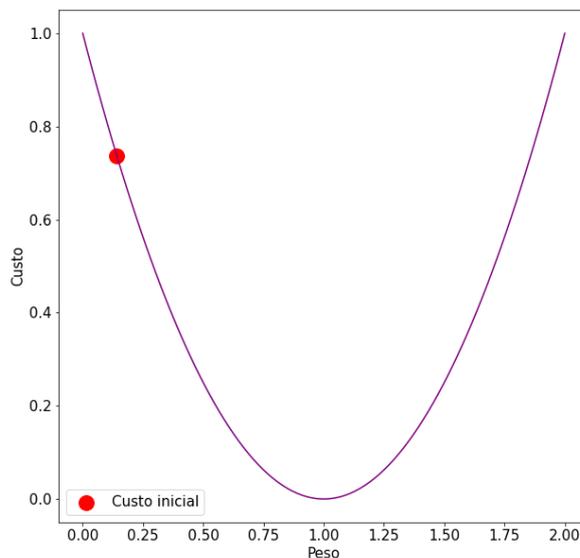
Uma vez definida uma função que aponta a qualidade da predição realizada pelo modelo, procura-se pesos e *biases* que minimizem a Equação 3. Essa minimização pode ser feita de várias formas para um Regressão Linear, mas a metodologia iterativa que será apresentada por esse texto será estendida para problemas e modelos mais complexos.

3.2. DESCIDA DO GRADIENTE

Realizando uma breve análise da função de custo MSE, definida na seção anterior, pode-se observar que quanto melhor for a predição realizada pelo modelo, menor será a função de custo, uma vez que a mesma sempre fornece valores positivos. Dessa forma, deve-se procurar pesos e *biases* que tendem a minimizar essa função, em outras palavras, procura-se coeficientes de reta que aproximem a função de custo do zero. Para tanto, é comum a utilização de um algoritmo iterativo de otimização chamado de Descida do Gradiente.

Imagine que o comportamento de uma função de custo qualquer, denominada “ C ”, em relação ao peso, tenha um formato côncavo, como ilustrado na Figura 3. O ponto identificado no gráfico se refere ao valor da função de custo para um peso qualquer, adotado preliminarmente.

Figura 3: Função de custo em relação ao peso

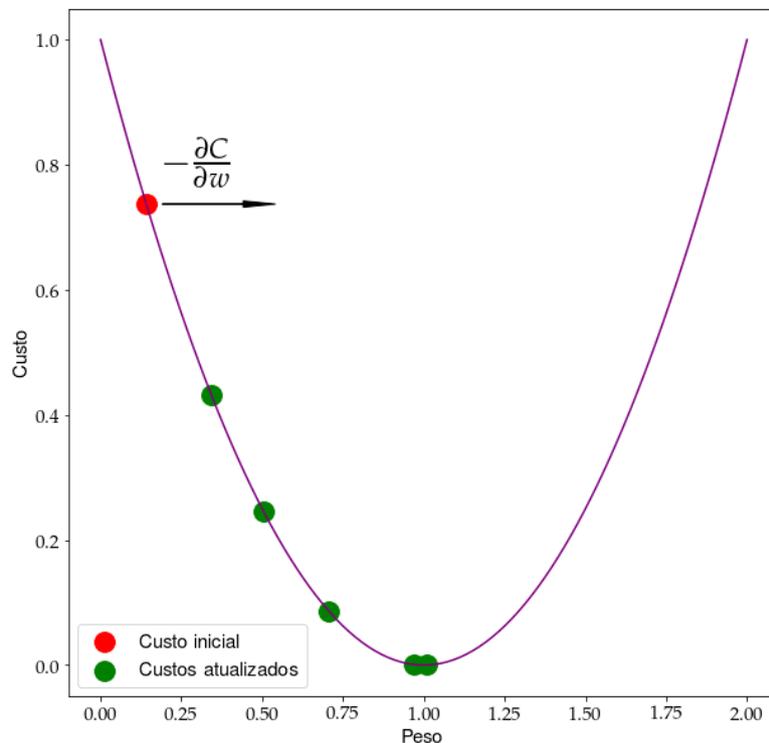


Fonte autoral.

A partir do conhecimento do valor do custo a partir desse ponto genérico, pode-se realizar pequenas modificações no valor do peso para que o ponto inicial “caminhe” na direção do mínimo da função. Para tanto, basta determinar a direção de decrescimento da função e realizar esse ajuste.

É visto nas disciplinas de Cálculo que o vetor gradiente indica a direção de maior crescimento da função a qual ele é calculado. Dessa forma, ao adotar o valor negativo do vetor gradiente, pode-se definir para qual direção deslocar o ponto a fim de alcançar o mínimo da função por meio de atualizações no valor do peso, como ilustra a Figura 4.

Figura 4: Atualização dos pesos.

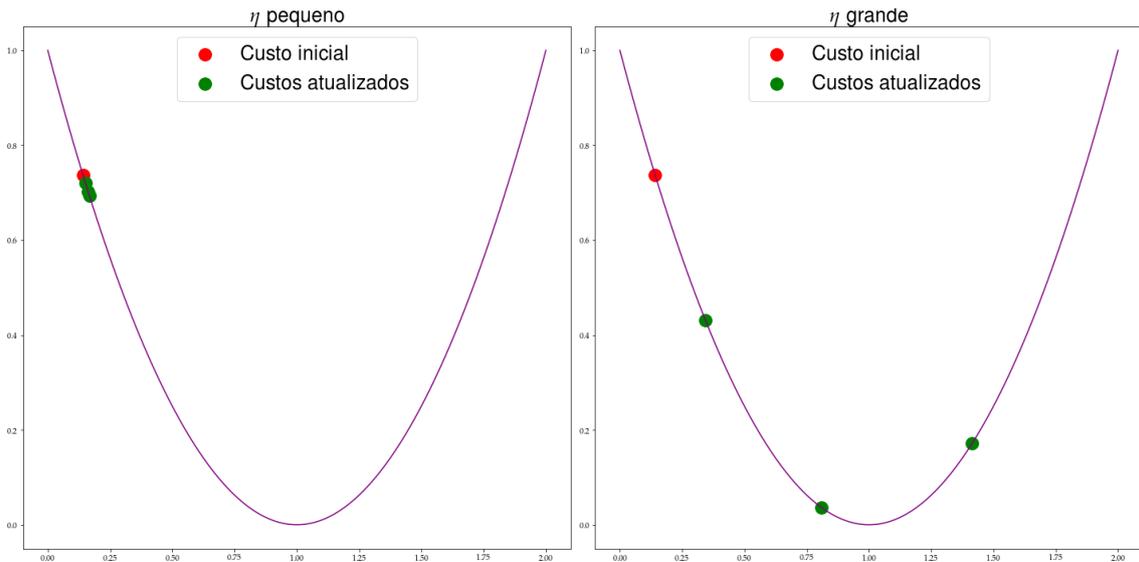


Fonte autoral.

Como o gradiente de uma função possui módulo, direção e sentido, é comum multiplicá-lo por um valor com o intuito de controlar a velocidade de deslocamento do ponto. Esse “valor” é denominado taxa de aprendizagem (*learning rate*), comumente representada pela letra grega “ η ”, e controla a variação da função de custo. Note que a escolha desse hiperparâmetro é de suma importância para o bom funcionamento do algoritmo.

Caso a taxa de aprendizagem seja muito grande, pode haver uma divergência durante a iteração do algoritmo ao ultrapassar o ponto de mínimo da função de custo. Para valores pequenos de “ η ”, por outro lado, pode-se necessitar uma alta demanda computacional para a convergência, uma vez que será necessário um grande número de iterações. Os exemplos citados são ilustrados pela Figura 5.

Figura 5: Determinação do *learning rate*.



Fonte autoral.

Como mostrado na Figura 4, a atualização do peso é dada pela derivada parcial da função de custo em relação ao peso. De forma análoga, o *bias* também é atualizado. Segue, então, que a atualização do modelo pelo algoritmo de Descida de Gradiente é dada pela Equação 4, onde “ θ ” é o vetor que contém os parâmetros do modelo (pesos e *biases*).

$$\begin{aligned}
 w &\rightarrow w' = w - \eta \frac{\partial C}{\partial w} . \\
 b &\rightarrow b' = b - \eta \frac{\partial C}{\partial b} . \\
 \theta &\rightarrow \theta' = \theta - \eta \nabla C , \qquad \qquad \qquad \text{(Eq. 4)}
 \end{aligned}$$

Enquanto o vetor gradiente da função de custo “ ∇C ” do exemplo exposto até agora possui derivadas parciais em relação a um peso e a um *bias*, apenas, em problemas reais há a necessidade de estimar milhares, ou até mesmo milhões ou bilhões, de pesos e *biases*. Dessa forma, o cálculo dessas derivadas precisa de um grande esforço computacional. Um outro algoritmo, então, chamado de *backpropagation*, que será pincelado mais a frente, é utilizado para realizar essa tarefa.

A atualização de todos os pesos e *biases* a cada predição de cada exemplo contido em um banco de dados pode ser extremamente custosa do ponto de vista computacional dependendo da complexidade do problema ou do modelo. Dessa forma, uma variação da Descida de Gradiente é comumente utilizada: a Descida de Gradiente Estocástica.

Esse algoritmo separa lotes de entradas aleatórias e calcula a função de custo com base na média dessa função aplicada ao lote, como ilustra a Equação 5, e só então atualiza os

pesos e *biases* do modelo. Dessa forma, menos operações são realizadas e menor é o tempo de processamento necessário.

$$C = \frac{1}{n} \sum_x C_x , \quad (\text{Eq. 5})$$

onde “*n*” é o número de exemplos contidos no lote e “*C_x*” é a função de custo aplicada a cada exemplo.

Exemplo:

Aplique o algoritmo da Descida de Gradiente na função de custo $L = (x - x_f)^2$, onde “*x*” é a variável a ser otimizada que deve ser inicializada de forma aleatória e “*x_f* = 10” é o valor que deve ser alcançado. Plote os gráficos de “*x*” e da função de custo em relação às épocas.

Solução:

Pode-se determinar a derivada da função de custo proposta em relação à “*x*” facilmente, de forma analítica. Entretanto, há uma forma numérica de se aproximar a derivada em um ponto por meio da Equação 6, já aplicada à função “*L*”:

$$\frac{\partial L}{\partial x} = \frac{L(x + \delta) - L(x - \delta)}{2\delta} , \quad (\text{Eq. 6})$$

onde “*δ*” é a distância entre dois pontos próximos e arbitrários.

Dessa forma, o algoritmo pode ser aplicado, como ilustra o Quadro 16.

Quadro 16: Resolução do Exemplo 1.

```
x_f = 10
x = np.random.randn()
print(f'Variável "x" inicializada com {x}.')

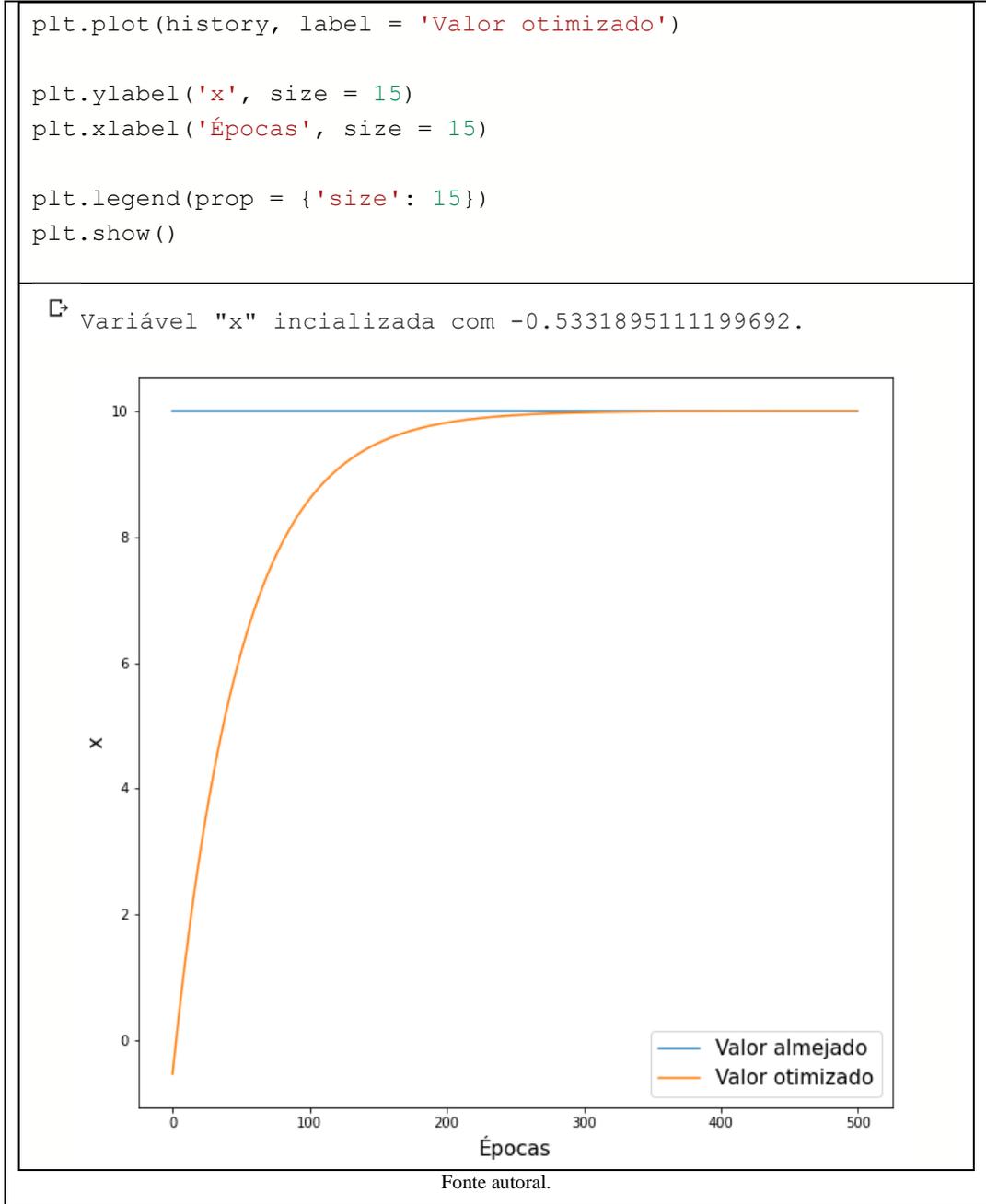
eta = 1e-2
history = []

# Definição da função de custo
loss = lambda x: (x - x_f)**2

# dL/dx
delta = 0.0001
dL_dx = lambda x: (loss(x+delta) - loss(x-delta)) / (2*delta)

# Descida de Gradiente
epochs = 500
for i in range(epochs):
    history.append(x)
    x = x - eta*dL_dx(x)

# Plotagem gráfica da convergência
plt.figure(figsize = (10, 10))
plt.plot([0, epochs], [x_f, x_f], label = 'Valor almejado')
```



3.3. REDES NEURAIAS

Como bem aponta Michael Nielsen (2015) em seu livro “*Neural Networks and Deep Learning*”, o estudo de Redes Neurais Artificiais não é algo novo e data o ano de 1958 com o artigo “*The perceptron: A probabilistic model for information storage and organization in the brain.*”, de Frank Rosenblatt. Nesse trabalho, o cientista propôs um modelo matemático de neurônio artificial, denominado Perceptron, para tentar simular o funcionamento de um neurônio biológico.

O avanço rápido em Aprendizagem de Máquina, entretanto, só passa a existir após a primeira década no século XXI com a popularização das GPU’s e do crescimento

exponencial de informações geradas e armazenadas em grandes bancos de dados, o que possibilitou o desenvolvimento de modelos de aprendizagem mais robustos.

O entendimento de um modelo de Rede Neural deve passar, previamente, pelo neurônio artificial, ainda chamado de Perceptron. Seu funcionamento, *a priori*, é bastante simples. O Perceptron é apenas uma operação computacional que recebe “*n*” entradas e calcula uma saída, resultado de uma combinação linear desses *inputs*. Rosenblatt, inicialmente, propôs que a saída desse neurônio fosse binária; caso a soma proporcional das entradas fosse maior do que um número limite, escolhido arbitrariamente, a saída seria igual a 1, como ilustra a Equação 7.

$$saída = \begin{cases} 0, & z \leq limite \\ 1, & z > limite \end{cases} \quad (Eq. 7)$$
$$onde: z = \sum_{i=1}^n W_i x_i ,$$

em que “*x*” é a variável de entrada e “*W*” o peso atribuído a ela.

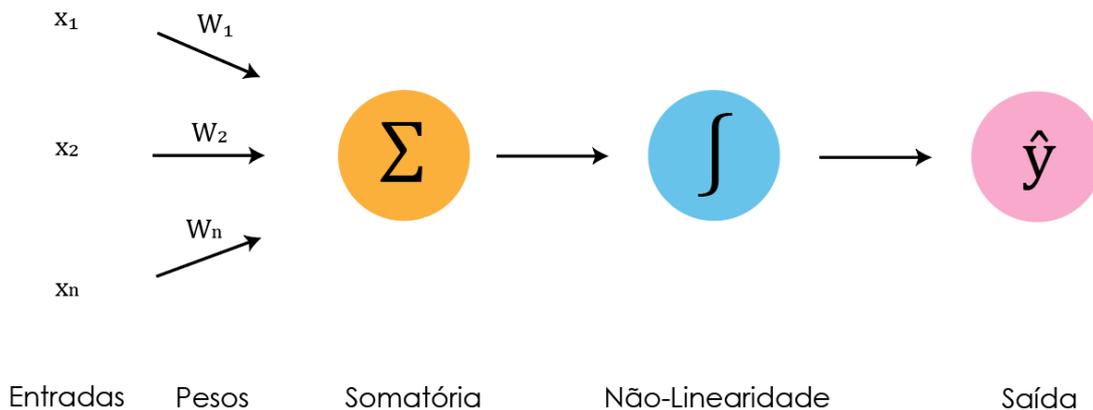
Dessa forma, o Perceptron conseguiria simular uma tomada de decisões baseando-a na relevância de cada entrada. Por exemplo, ao decidir comprar uma casa, os pesos atribuídos às variáveis “localização” e “tamanho do imóvel” seriam grandes; variáveis como “cor da fachada”, por outro lado, não teria tanta relevância e, portanto, possuiria um peso pequeno.

Hoje em dia, o neurônio artificial é modelado de uma forma um pouco diferente do que fora proposto inicialmente, como ilustra a Equação 8, em que as variáveis são definidas vetorialmente. A Figura 6, por sua vez, esquematiza a operação computacional. Note que a Equação 8 muito se assemelha à equação característica da Regressão Linear e, portanto, seu entendimento pode se beneficiar dessa analogia.

$$a = \varphi(wx + b) , \quad (Eq. 8)$$

onde “ φ ” é uma função não linear qualquer.

Figura 6: O Perceptron.

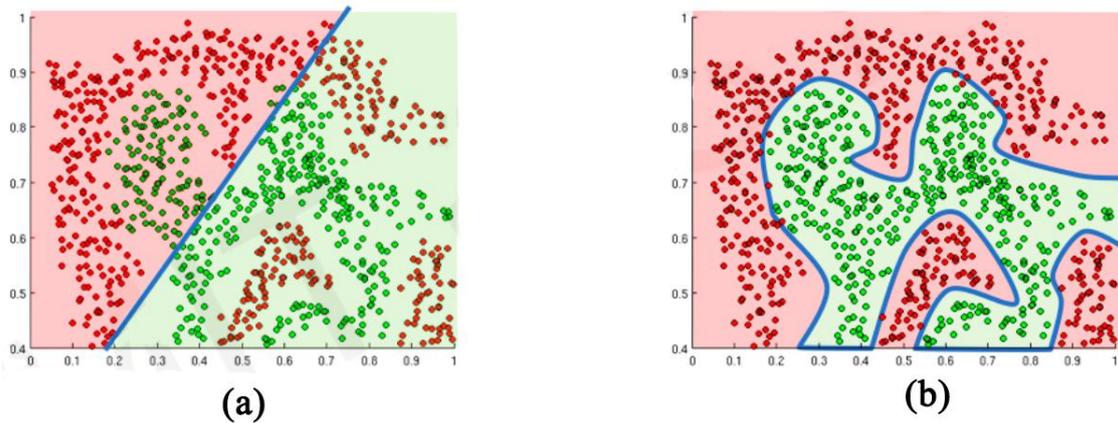


Fonte: MIT 6.S191: Introduction to Deep Learning.

A função não linear é acrescentada à modelagem do neurônio artificial uma vez que a maior parte dos problemas reais não possuem uma relação linear. Chama-se essa função de Função de Ativação uma vez que a mesma é responsável por “ativar” os neurônios que representem escolhas, ou características, importantes ao problema. Muitas funções são utilizadas como função de ativação, como a Sigmoid, Tanh e ReLu, cada uma tendo suas características e aplicações.

Como forma de entender a importância da não-linearidade nos modelos de aprendizagem, suponha que um modelo de ML seja criado com o intuito de separar os pontos verdes dos vermelhos dos dados ilustrados na Figura 7-a. Um modelo linear não seria capaz de realizar essa classificação de forma satisfatória, uma vez que os dados não possuem uma relação linear. Ao acrescentar não-linearidade ao modelo, como ilustrado na Figura 7-b, o modelo se torna complexo o suficiente para fazer a distinção necessária.

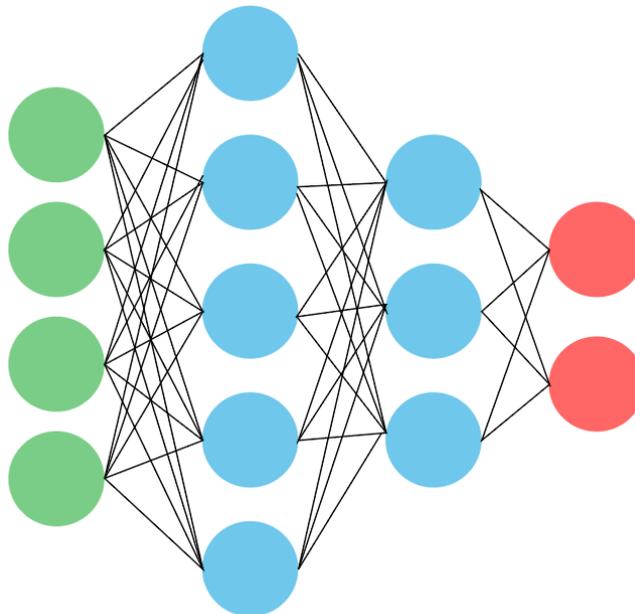
Figura 7: Importância da Função de Ativação



Fonte: MIT 6.S191: Introduction to Deep Learning

Tendo modelado um neurônio artificial, pode-se organizá-los em diversas arquiteturas formando redes: as Redes Neurais. A arquitetura de Rede Neural mais comum é conhecida como *Feedforward Neural Network*, ou *Multilayer Perceptron*. Nela, os neurônios são organizados em camadas e a informação é propagada de uma camada para outra, até que, ao final da rede, haja uma predição. A Figura 8, em que as circunferências representam os neurônios e as linhas os seus respectivos pesos, ilustra um exemplo de Rede Neural com 4 entradas, 2 saídas e 4 camadas.

Figura 8: Rede *Feedforward*.



Fonte autoral.

Na ilustração da Figura 8, a camada formada pelos neurônios verdes é a camada de entrada, que admite os *inputs* do modelo, a camada vermelha é a camada da saída, onde são feitas as previsões, e as camadas azuis são chamadas de *hidden layers* (camadas escondidas), onde cada neurônio é responsável por uma “característica” inerente à previsão.

Pode-se utilizar o modelo didático da Figura 8 como forma de ilustrar o funcionamento de uma Rede Neural *Feedforward*. Primeiramente, todos os pesos e *biases* do modelo são inicializados. As entradas do modelo, então, chegam por meio da camada de entrada e são transferidas junto aos seus respectivos pesos para cada neurônio da primeira *hidden layer*. Cada neurônio desse realiza uma operação segundo a Equação 8 e seus resultados, juntamente de seus pesos, são passados para cada neurônio da camada seguinte até o final da rede, onde há a previsão.

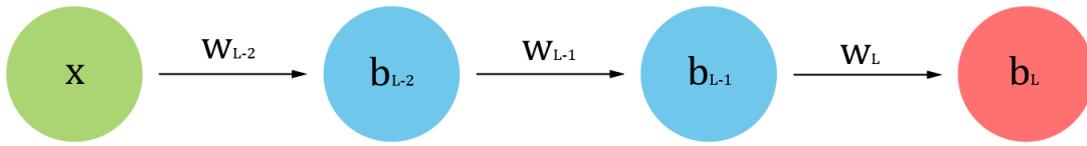
Feita a previsão, calcula-se uma função de custo para determinar o quão eficiente é o modelo e se atualiza todos os pesos e *biases* por meio da Descida de Gradiente, ou de um algoritmo de otimização similar. O processo é, então, repetido até a convergência do modelo, momento em que cada neurônio se torna capaz de identificar um padrão ou característica presente nos dados de entrada, permitindo, assim, que o modelo faça previsões corretas.

3.3.1. BACKPROPAGATION

Lida-se, em redes neurais, com um grande número de pesos e *biases* que precisam ser otimizados durante a fase de treinamento do modelo. Como já fora comentado, o cálculo de todos os gradientes relacionados ao problema é muito custoso do ponto de vista computacional. Como forma de mitigar o grande tempo de processamento, utiliza-se um algoritmo chamado *Backpropagation*.

Como veículo de entendimento do algoritmo, leve em consideração uma das redes neurais mais simples que pode ser criada, ilustrada na Figura 9, contendo apenas 1 entrada, 2 *hidden layers* e 1 saída. Essa rede é definida por 3 pesos e 3 *biases* e, portanto, a função de custo, quando aplicada a ela, é dependente dessas 3 variáveis.

Figura 9: Rede para o *Backpropagation*.



Fonte autoral.

Após a etapa de *feedforward*, ou seja, após a passagem de informação pela rede, procura-se calcular a função de custo e as derivadas parciais dessa função em relação a cada um dos parâmetros da rede, como forma de entendimento quanto à variação da saída do modelo em relação à variação de cada um desses pesos e *biases*. O Algoritmo de Retropropagação, ou *Backpropagation*, determina essas derivadas por meio da Regra de Cadeia, permitindo que a informação percorra o caminho contrário, em direção à entrada, e observando como ela se comporta durante o caminho.

Considere apenas os dois últimos neurônios da rede da Figura 9. A ativação do último neurônio segue a Equação 8 e pode ser escrita como na Equação 9. Dessa forma, a função de custo para um único exemplo pode ser determinada de acordo com a Equação 10.

$$a_L = \varphi(w_L a_{L-1} + b_L)$$

$$a_L = \varphi(z_L); \forall z_L = w_L a_{L-1} + b_L \quad (\text{Eq. 9})$$

$$C_0 = (a_L - y)^2 \quad (\text{Eq. 10})$$

Pela Regra da Cadeia, a taxa de variação da função de custo em relação ao peso da última camada do modelo, “ w_L ”, pode ser escrita pela Equação 11. O cálculo dessas derivadas é fácil e permite reescrever a Equação 11 como a Equação 12. Ao estudar os termos da Equação 12, pode-se observar que a derivada parcial da função de custo em relação ao peso da última camada pode ser determinada facilmente por valores já calculados durante a etapa de *feedforward* e pela derivada da função de ativação, que pode ser calculada analiticamente na maior parte dos casos.

$$\frac{\partial C_0}{\partial w_L} = \frac{\partial z_L}{\partial w_L} \frac{\partial a_L}{\partial z_L} \frac{\partial C_0}{\partial a_L} \quad (\text{Eq. 11})$$

$$\frac{\partial C_0}{\partial w_L} = (a_{L-1}) (\varphi'(z_L)) (2(a_L - y)) \quad (\text{Eq. 12})$$

De forma análoga, pode-se estimar o comportamento da função de custo em relação ao *bias* da camada “ L ” e à ativação da camada “ $L - 1$ ”, representados pelas equações 13 e 14. Embora a formulação realizada até agora fora feita apenas para um exemplo, a função

de custo de um lote pode ser determinada pela média das funções de custo individuais, o que permite a utilização dessas equações.

$$\frac{\partial C_0}{\partial b_L} = \frac{\partial z_L}{\partial b_L} \frac{\partial a_L}{\partial z_L} \frac{\partial C_0}{\partial a_L} = (\varphi'(z_L))(2(a_L - y)) \quad (\text{Eq. 13})$$

$$\frac{\partial C_0}{\partial a_{L-1}} = \frac{\partial z_L}{\partial a_{L-1}} \frac{\partial a_L}{\partial z_L} \frac{\partial C_0}{\partial a_L} = (w_L)(\varphi'(z_L))(2(a_L - y)) \quad (\text{Eq. 14})$$

Por outro lado, a determinação de “ $\frac{\partial C_0}{\partial w_{L-1}}$ ”, assim como a variação em relação ao *bias* e a ativação do neurônio anterior, de forma análoga, pode ser expressa pela Equação 15, em que o último termo já fora calculado pela Equação 14. Dessa forma, o algoritmo de *Backpropagation* propaga os erros da última camada até a camada de entrada, determinando, sem muito esforço computacional, as derivadas parciais e, consequentemente, o vetor gradiente do modelo.

$$\frac{\partial C_0}{\partial w_{L-1}} = \frac{\partial z_{L-1}}{\partial a_{L-2}} \frac{\partial a_{L-2}}{\partial z_{L-1}} \frac{\partial C_0}{\partial a_{L-1}} = (a_{L-2})(\varphi'(z_{L-1})) \frac{\partial C_0}{\partial a_{L-1}} \quad (\text{Eq. 15})$$

A extensão do que foi apresentado para uma rede neural mais complexa, assim como a rede da Figura 8, é feita com a utilização de uma notação vetorial, levando em consideração que a derivada da função de custo em relação à ativação de um neurônio pertencente a uma camada anterior é uma contribuição das variações de todos os neurônios da camada à frente.

4. ESTUDO DE CASO: DADOS MNIST

Neste capítulo, será desenvolvido um modelo de aprendizagem de máquina baseado em redes neurais para a classificação de dígitos manuscritos. O banco de dados que será utilizado é o conjunto MNIST, criado pela *National Institute of Standards and Technology* e amplamente utilizado em cursos introdutórios à *Machine Learning*.

O conjunto MNIST contém 60 mil dados rotulados destinados ao treino do modelo e 10 mil dados destinados ao teste. Cada conjunto é composto por uma imagem de um dígito manuscrito de 0 a 9, em uma resolução de 28x28 pixels, em preto e branco.

4.1. PRÉ-PROCESSAMENTO E ANÁLISE EXPLORATÓRIA

A primeira etapa do desenvolvimento de um modelo de aprendizagem deve ser o pré-processamento. Nesta etapa, deve-se explorar o conjunto de dados e realizar modificações no mesmo de forma a facilitar a etapa de aprendizagem, sem modificar o comportamento das covariáveis.

Para tanto, é necessária a importação do conjunto de dados que será trabalhado. Como já comentado, o MNIST *dataset* é amplamente utilizado como tema introdutório à *Machine Learning* e visão computacional e, portanto, tem sua importação facilitada. Como mostrado pelo Quadro 16, primeiramente, há a importação das bibliotecas que serão utilizadas durante o desenvolvimento do modelo e, em seguida, o banco de dados é importado por meio do Keras.

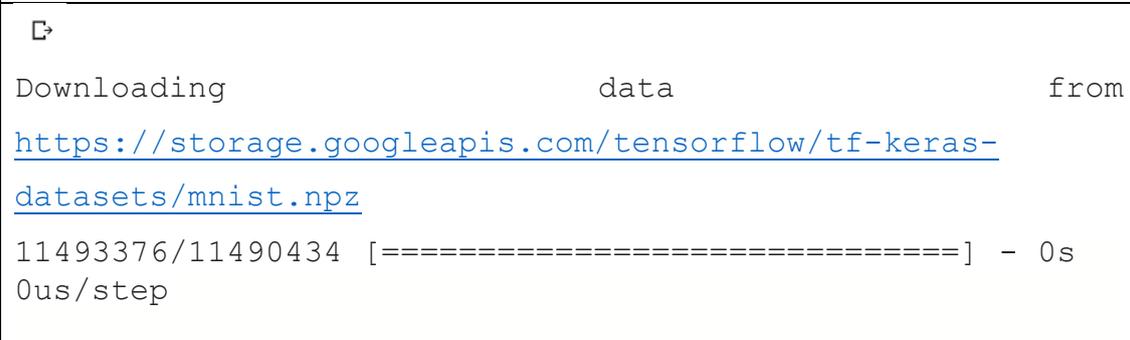
Quadro 16: Importação de Bibliotecas e do MNIST Dataset

```
%tensorflow_version 2.x
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Dropout

import numpy as np
import matplotlib.pyplot as plt

from IPython.display import display

train_set, test_set = tf.keras.datasets.mnist.load_data(path = 'mnist.npz')
train_x, train_y = train_set
test_x, test_y = test_set
```



Fonte autoral.

Não é necessário, nesse momento, entender cada importação. O entendimento de cada biblioteca virá com a utilização das mesmas. O carregamento do banco de dados, por sua vez, vem de “tf.keras.datasets.mnist.load_data(path = ‘mnist.npz’)”; sua forma de utilização é sugerida pela própria documentação do Keras. Note, entretanto, que o retorno dessa função é dado por um conjunto de treino (“train_set”) e um conjunto de teste (“test_set”), separados, posteriormente, por “train_x”, “train_y”, “test_x”, “test_y”, exemplos e rótulos contidos no banco de dados..

Pode-se verificar o formato desses dois subconjuntos com a utilização do método “.shape” do Numpy, como ilustrado no Quadro 17. Feito isso, observa-se que o conjunto

de treino possui 60000 exemplos contendo 28 linhas e 28 colunas, enquanto os dados de teste são organizados por 10000 exemplos, 28 por 28, linhas e colunas referentes à resolução da imagem. Enquanto os exemplos são imagens com a resolução já citada, os respectivos rótulos são valores que variam entre 0 e 9.

Quadro 17: Formato do conjunto.

```
display(train_x.shape, train_y.shape, test_x.shape, test_y.shape)
```

```
↳
```

```
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
```

Fonte autoral.

A separação em subconjuntos de teste e treino é muito importante e comum no desenvolvimento de modelos de aprendizagem. Durante a etapa de otimização, o modelo pode se ajustar às nuances dos dados em que está tendo contato, fazendo com que a rede não tenha um poder de generalização. Esse evento é chamado de *overfitting* e, muitas vezes, está relacionado com o excesso de complexidade do modelo, ou com um tempo exagerado de treinamento. Quando, por outro lado, o modelo não consegue se adequar a nenhum conjunto de dados, fala-se em *underfitting*.

Para evitar os problemas de *underfitting*, costuma-se realizar preparações estatísticas nos dados de treino ou, até mesmo, aumentar o conjunto de dados. O problema de *overfitting*, por outro lado, é evitado com a parada antecipada do treinamento e com regularizações. Para tanto, se mostra necessária uma separação entre os dados de treino e teste para que, dessa forma, tenha-se um parâmetro de comparação no que tange a generalização do modelo.

Durante a otimização do modelo, é comum o ajuste de hiperparâmetros e a realização de tratamentos estatísticos com o intuito de aproximar a predição do modelo com os dados reais. Dessa forma, cria-se a possibilidade de um *overfitting* em relação ao conjunto de teste e, portanto, mostra-se necessária a criação de um terceiro subconjunto que permita uma validação intermediária: os dados de validação. A partir da importação dos dados MNIST, o Quadro 18 ilustra a criação desse subconjunto.

Quadro 18: Criação dos dados de validação.

```
# Criação do validation set
val_x, val_y = train_x[-10000:], train_y[-10000:]
train_x, train_y = train_x[:-10000], train_y[:-10000]
```

```
display(train_x.shape, train_y.shape, val_x.shape, val_y.shape)
```

↳

```
(50000, 28, 28)
(50000,)
(10000, 28, 28)
(10000,)
```

Fonte autoral.

Separado os conjuntos a serem trabalhados, é importante que os dados sejam explorados para um melhor conhecimento dos mesmos. Em problemas de regressão, a etapa de análise exploratória é muito importante e serve como meio de adquirir *insights* para a resolução do problema. No problema posto por esse capítulo, pode-se utilizar a função “`imshow()`” do “`matplotlib.pyplot`” para a visualização das imagens dos dígitos manuscritos, como ilustra o Quadro 19.

Quadro 19: Ilustração das imagens do MNIST.

```
def show_data(examples, targets):
    rand_num = np.random.randint(0, examples.shape[0] - 20)

    plt.figure(figsize = (13,13))

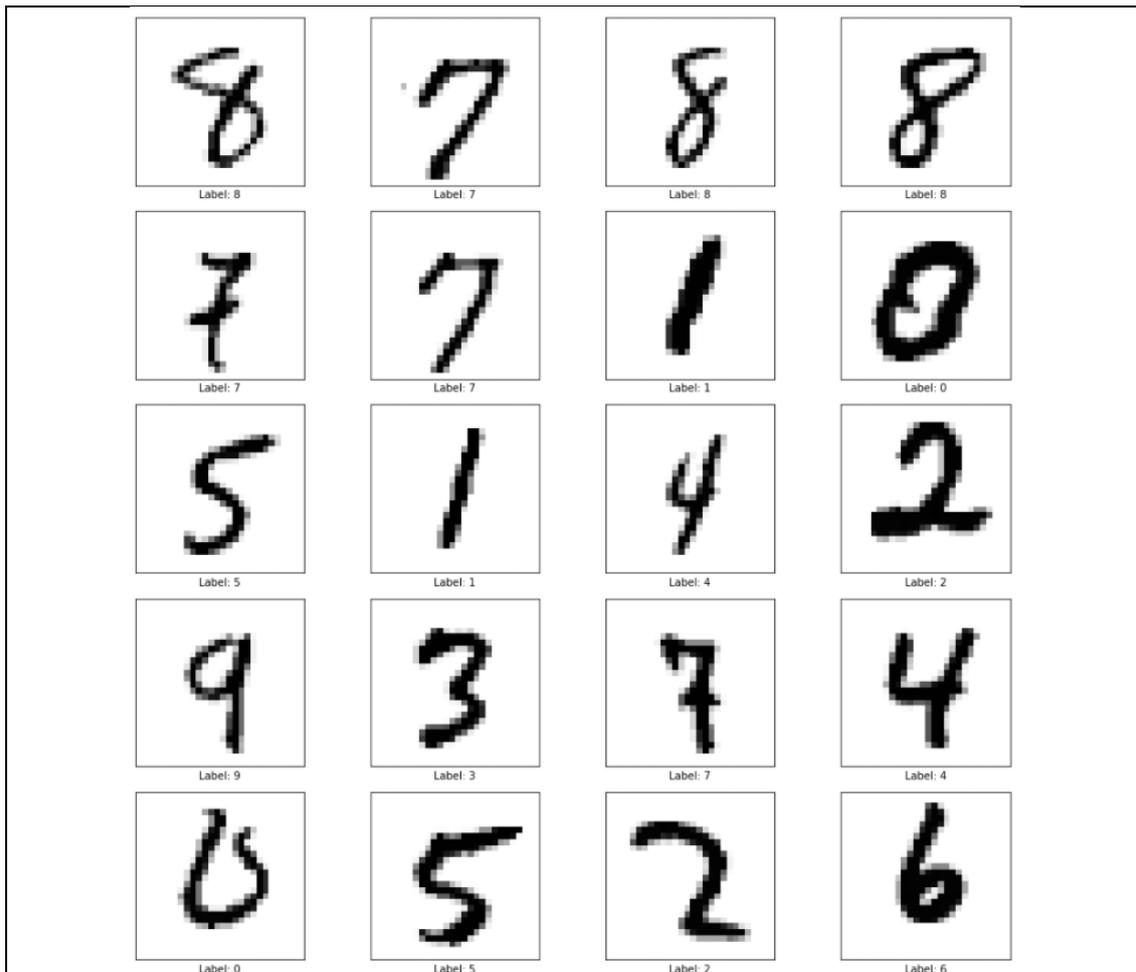
    for i in range(20):
        plt.subplot(5, 4, i+1)
        plt.imshow(examples[rand_num + i], cmap = 'Greys')
        plt.xlabel(f"Label: {targets[rand_num + i]}")

        plt.xticks([])
        plt.yticks([])

    plt.tight_layout()
    plt.show()

show_data(train_x, train_y)
```

↳



Fonte autoral.

A função criada “show_data” ilustra 20 exemplos contidos em um banco de dados qualquer a partir de um número gerado aleatoriamente, representado pela variável “rand_num”, em que o valor da função “np.random.randint()”, que retorna um valor entre 0 e “números de exemplos - 20”, é atribuído a ela.

Por meio do método “.max()”, pode-se observar, ainda, que os dados numéricos contidos nos conjuntos variam de 0 a 255. Uma rede neural possui dificuldades em treinar com dados que possuam uma diferença numérica muito grande. Essa dificuldade se dá devido a etapa de *backpropagation*, em que os gradientes se tornam muito grandes e distintos. Para tanto, é normal a aplicação de uma normalização. Existem muitas técnicas relacionadas a esse assunto, mas, como ilustra o Quadro 20, será utilizada uma normalização simples para o estudo de caso abordado.

Quadro 20: Normalização dos dados.

```
print(f'Valor máximo do pixel antes da normalização: {tra
in_x.max()}')

# Normalização
```

```
max_value = train_x.max()
train_x, test_x, val_x = train_x/max_value, test_x/max_value, val_x/max_value

print(f'Valor máximo do pixel depois da normalização: {train_x.max()}')
```

↳

```
Valor máximo do pixel antes da normalização: 255
Valor máximo do pixel depois da normalização: 1.0
```

Fonte autoral.

Por último, a alimentação de uma rede neural simples é dada por um vetor de entrada. Entretanto, os dados MNIST são matrizes que representam as imagens dos dígitos escritos à mão. É necessário, então, que haja uma manipulação nos dados de entrada, mudando o formato “28x28” para “784x1”. O Quadro 21 ilustra tal processamento, pautado na utilização do método “.reshape”, que muda o formato de um *array*.

Quadro 21: Formatação dos dados de entrada.

```
# Redução da dimensão do tensor
def dimension_reduct(tensor):
    num_pixels = 28*28

    return tensor.reshape((tensor.shape[0], num_pixels))

train_x = dimension_reduct(train_x)
test_x = dimension_reduct(test_x)
val_x = dimension_reduct(val_x)

display(train_x.shape, test_x.shape, val_x.shape)
```

↳

```
(50000, 784)
(10000, 784)
(10000, 784)
```

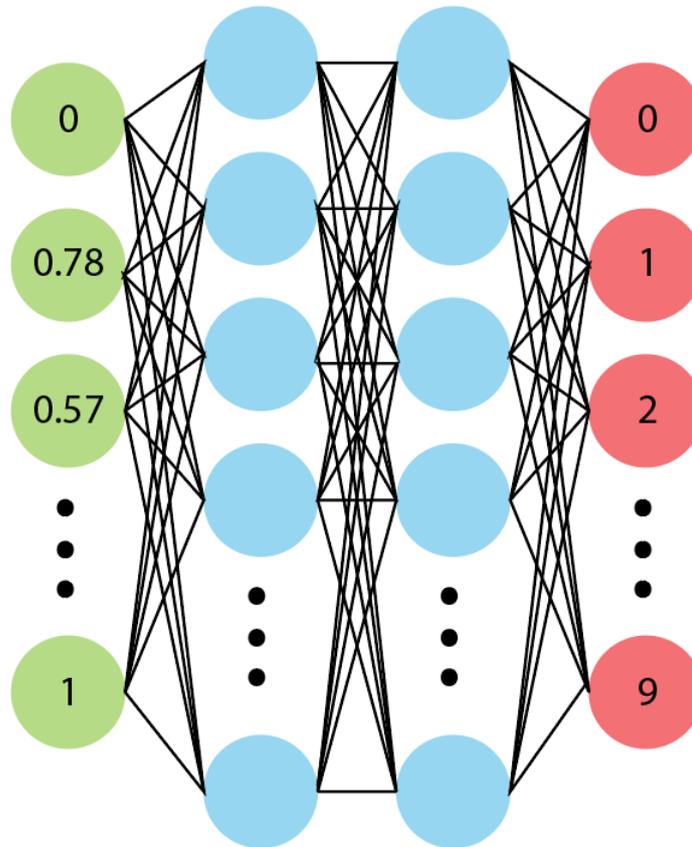
Fonte autoral.

4.2. CRIAÇÃO E TREINAMENTO DO MODELO

A criação de um modelo de aprendizagem baseado em redes neurais deve levar em consideração muitos fatores, tais como número de camadas, número de neurônios em cada camada, funções de ativações, taxa de aprendizagem, tamanho dos lotes, número de épocas etc. Mesmo para redes mais simples, a determinação desses hiperparâmetros é difícil e, muitas das vezes, é realizada de forma iterativa.

Para o problema posto, optou-se por criar um modelo com 2 *hidden layer*, a primeira com 200 neurônios e a segunda com 100 neurônios, como ilustra a Figura 10. Essa redução no número de neurônios é comum e busca uma maior especificação de cada neurônio no que tange a capacidade de identificar padrões nos dados.

Figura 10: Morfologia.



Fonte autoral.

Pode-se observar da Figura 10 que a camada de entrada é composta de 784 neurônios, que representam os pixels das imagens, que variam entre 0 e 1. A camada de saída, por sua vez, é composta por 10 neurônios que representam os dígitos entre 0 e 9.

Para a construção do modelo de aprendizagem, esse texto opta pela utilização da API Keras. Mais especificamente, a estrutura “Sequential”, contida nessa aplicação. Essa estrutura lida muito bem com redes neurais completamente conectadas, ou seja, com redes em que as camadas passam suas ativações diretamente para as camadas seguintes.

O código em Python é ilustrado pelo Quadro 22. Primeiro, é criada uma função “create_model” com o intuito de retornar o modelo de aprendizagem proposto. Atribui-se à variável “model” a estrutura “Sequential” e, posteriormente, adiciona-se camada por camada por meio do método “.add”. Camadas completamente conectadas são criadas pela

estrutura “Dense”, que recebe como parâmetro o número de neurônios pertencentes a essa camada e a função de ativação característica aos neurônios, dentre muitos outros parâmetros que não será abordado por esse texto.

Quadro 22: Criação do modelo.

```
def create_model(eta = 0.01):
    model = Sequential()
    model.add(Dense(units = 200, activation = 'relu', input
_shape = (784,))) # camada de entrada e primeira hidden l
ayer
    model.add(Dense(units = 100, activation = 'relu')) # se
gunda hidden layer
    model.add(Dense(units = 10, activation = 'softmax')) #
saída

    optimizer = tf.keras.optimizers.SGD(eta)
    model.compile(optimizer = optimizer,
                  loss = 'sparse_categorical_crossentropy',
                  metrics = ['acc'])

    return model
```

Fonte autoral.

Observe que, ainda do Quadro 22, não foi criado uma camada de entrada. Isso ocorre porque a estrutura “Dense” permite a criação dessa camada de forma indireta, adicionando à primeira *hidden layer* o parâmetro “input_shape”. Esse parâmetro recebe como entrada uma tupla ou lista que represente a formato de entrada do modelo que está sendo criado.

Escolheram-se as funções de ativações ReLu e Softmax para as 2 *hidden layers* e para a camada de saída, respectivamente. A função Unidade Linear Retificada (ReLU), definida pela Equação 16, é uma escolha comum de não linearidade e retorna 0 para valores negativos e possui uma relação linear para valores positivos.

$$ReLU(z) = \max(0, z) \quad (\text{Eq. 16})$$

Para a camada de saída, por outro lado, não se espera um valor numérico qualquer. É desejado que, ao final do modelo, seja apontado um rótulo específico dentre os 10 existentes. Para tanto, utiliza-se a função Softmax, definida pela Equação 17, que retorna a probabilidade de cada neurônio contido na camada de saída ser o rótulo verdadeiro. Dessa forma, para uma rede treinada, uma imagem contendo o dígito manuscrito 2 retornaria um valor grande no neurônio que representa o rótulo 2 e valores pequenos nos demais neurônios, indicando que há uma grande possibilidade daquela imagem representar o numeral em questão.

$$S(z) = \frac{e^{(z_i)}}{\sum_j e^{z_j}} \quad (\text{Eq. 17})$$

Após a definição do modelo, é necessário compilá-lo. Para tanto, utiliza-se o método “.compile”, informando qual o algoritmo de otimização (*optimizer*), a função de custo (*loss*) e as métricas (*metrics*) que serão utilizadas. Para esse primeiro modelo, opta-se pela utilização da Descida de Gradiente estocástica, que pode ser acessada por “tf.keras.optimizers.SGD”, que recebe como parâmetro a taxa de aprendizagem, e pela Acuracidade, que retorna a porcentagem de acertos, como otimizador e métrica, respectivamente.

A função de custo, diferentemente da MSE para problemas de regressão, deve levar em consideração as probabilidades atingidas ao final do modelo. Para tanto, utiliza-se a *Sparse Categorical Crossentropy*, definida pela equação 18, uma escolha comum para tratar problemas de multivariáveis.

$$SCCE(z) = -\frac{1}{N} \sum_{i=1}^N y_i \log(\hat{y}_i) \quad (\text{Eq. 17})$$

Feito isso, pode-se criar o modelo proposto. Após a criação, é comum verificar se a construção do modelo foi correta. Para tanto, pode-se utilizar o método “.summary” para verificar as camadas criadas, as saídas de cada camada e o número de parâmetros existentes. O Quadro 23 ilustra o processo.

Quadro 23: Verificação do modelo.

<pre>Model = create_model() model.summary()</pre>		
		
<pre>Model: "sequential"</pre>		
Layer (type)	Output Shape	Param #
dense_22 (Dense)	(None, 200)	157000
dense_23 (Dense)	(None, 100)	20100
dense_24 (Dense)	(None, 10)	1010
<pre>Total params: 178,110 Trainable params: 178,110 Non-trainable params: 0</pre>		

Fonte autoral.

Feito o modelo, é necessária a criação de uma nova função para a execução da etapa de treinamento, ilustrada no Quadro 24. Essa etapa é regida, principalmente, pelo método “fit” que recebe como parâmetros os dados de treino, validação, a quantidade de lotes e o número de épocas, entre muitos outros, e executa as etapas de *feedforward* e

backpropagation no modelo criado. Ainda nessa função, salva-se todo o registro de treinamento na variável “history” para, posteriormente, plotar os gráficos da função de custo e da acuracidade ao longo do tempo, no final do treinamento.

Quadro 24: Treinamento.

```
def train_model(model, train_x, train_y, val_x, val_y,
batch_size, epochs):
    history = model.fit(train_x,
                        train_y,
                        batch_size = batch_size,
                        epochs = epochs,
                        validation_data = (val_x, val_y))

    # plot graphs
    fig = plt.figure(figsize = (20,10))

    plt.subplot(1,2,1)
    plt.plot(history.history['loss'], label = 'Treino')
    plt.plot(history.history['val_loss'], label = 'Validação
o')
    plt.xlabel('Épocas', size = 15)
    plt.ylabel('Custo', size = 15)
    plt.xticks(size = 15)
    plt.yticks(size = 15)
    plt.legend(prop = {'size': 15})

    plt.subplot(1,2,2)
    plt.plot(history.history['acc'], label = 'Treino')
    plt.plot(history.history['val_acc'], label = 'Validação
')
    plt.xlabel('Épocas', size = 15)
    plt.ylabel('Acuracidade', size = 15)
    plt.xticks(size = 15)
    plt.yticks(size = 15)
    plt.legend(prop = {'size': 15})

    plt.tight_layout()
    plt.show()
```

Fonte autoral.

Criadas as funções de modelagem e treinamento, pode-se treinar, efetivamente, o modelo de aprendizagem, como ilustra o Quadro 25. Para uma taxa de aprendizagem de 0.01, 32 lotes e 20 épocas, obtém-se 98,45% de acuracidade para os dados de treino e 97,36% para os dados de validação.

Quadro 25: Treinamento do modelo.

```
batch_size = 32
```

```
epochs = 20
```

```
train_model(model, train_x, train_y, val_x, val_y,  
batch_size, epochs)
```

↳

```
Epoch 1/20
```

```
1563/1563 [=====] - 4s 2ms/step  
- loss: 0.6717 - acc: 0.8233 - val_loss: 0.3180 -  
val_acc: 0.9136
```

```
Epoch 2/20
```

```
1563/1563 [=====] - 3s 2ms/step  
- loss: 0.3064 - acc: 0.9135 - val_loss: 0.2544 -  
val_acc: 0.9284
```

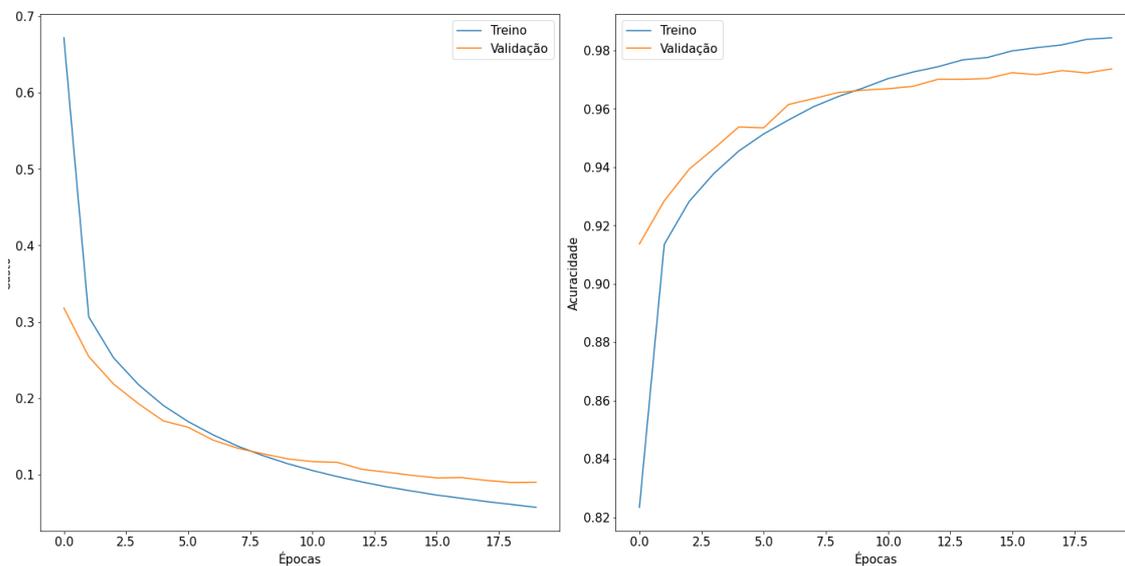
```
...
```

```
Epoch 19/20
```

```
1563/1563 [=====] - 4s 2ms/step  
- loss: 0.0609 - acc: 0.9837 - val_loss: 0.0897 -  
val_acc: 0.9722
```

```
Epoch 20/20
```

```
1563/1563 [=====] - 4s 2ms/step  
- loss: 0.0572 - acc: 0.9843 - val_loss: 0.0901 -  
val_acc: 0.9736
```

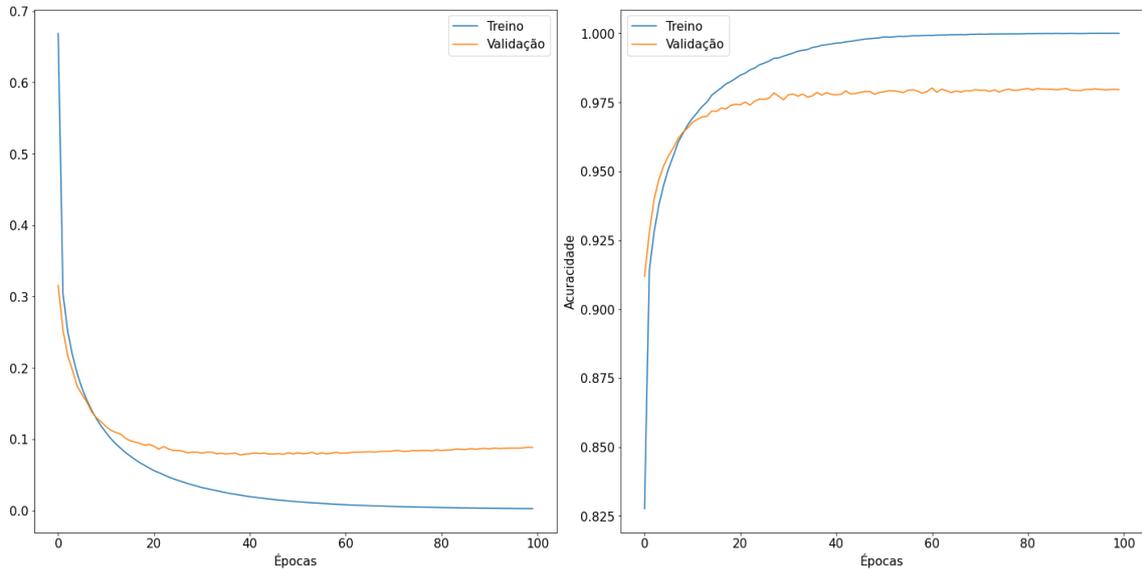


Fonte autoral.

Pode-se, então, realizar mudanças nos hiperparâmetros com o intuito de melhorar os resultados obtidos. Ao observar os gráficos gerados ao final do treinamento, pode-se observar que durante as 20 épocas, o modelo não alcançou a convergência, que seria representada por uma linha quase horizontal no final do gráfico. Aumentando-se o número de épocas para 100, obtém-se uma acuracidade de 97,96% nos dados de validação

e gráficos muito mais suaves, como ilustra a Figura 11. Todavia, a acuracidade dos dados de treino fora de 100%, o que sugere um *overfitting*. Logo, deve-se atentar ao treinamento excessivo.

Figura 11: Otimização 1 do modelo.



Fonte autoral.

Uma forma de evitar o *overfitting* é a aplicação de regularização. Um tipo de regularização comum às redes neurais é o *dropout*, que anula de forma aleatória, e a cada iteração, uma porcentagem dos neurônios, reduzindo, assim, a complexidade do modelo e favorecendo a robustez do mesmo, uma vez que o treinamento se dá por morfologias distintas a cada iteração, evitando que certos neurônios se coadaptem ao aprendizado de neurônios anteriores.

Ao acrescentar camadas *dropout* após cada *hidden layer*, que 50% dos neurônios de cada camada comentada, obtém-se uma acuracidade de 97,76% e 97,92% para os dados de treino e validação, respectivamente. A ausência de *overfitting* pode ser verificada, também, pelos gráficos da função de custo e acuracidade, ilustrados pela Figura 12. A modificação da morfologia do modelo é mostrado no Quadro 26.

Quadro 25: Treinamento do modelo.

```
def create_model(eta = 0.01):  
    model = Sequential()  
    model.add(Dense(units = 200, activation = 'relu', input  
_shape = (784,))) # camada de entrada e primeira hidden l  
ayer  
    model.add(Dropout(0.5))
```

```

model.add(Dense(units = 100, activation = 'relu')) # se
gunda hidden layer
model.add(Dropout(0.5))
model.add(Dense(units = 10, activation = 'softmax')) #
saída

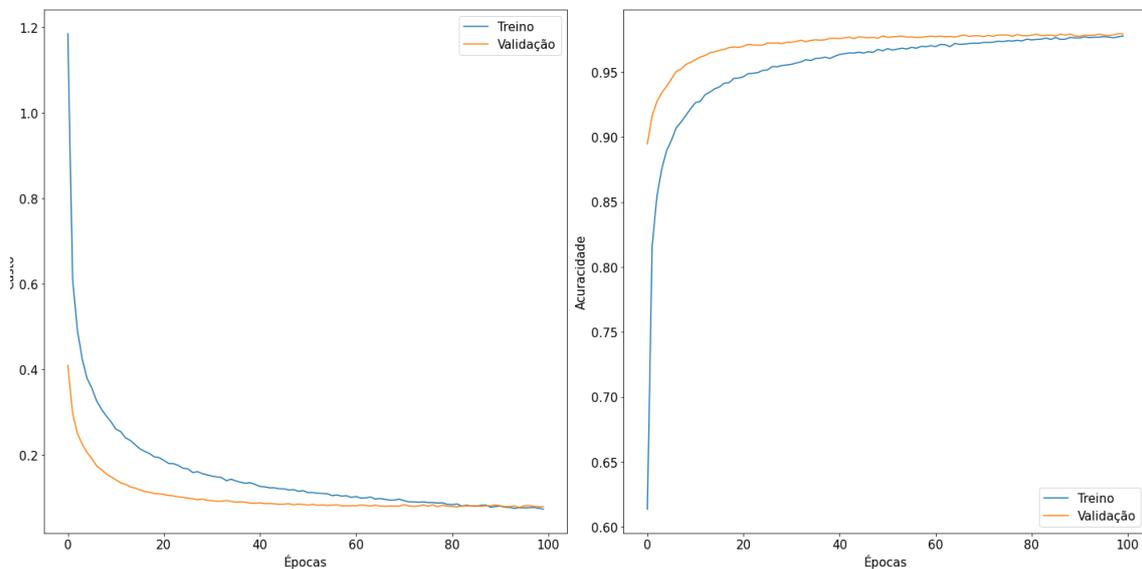
optimizer = tf.keras.optimizers.SGD(eta)
model.compile(optimizer = optimizer,
              loss = 'sparse_categorical_crossentropy',
              metrics = ['acc'])

return model

```

Fonte autoral.

Figura 12: Otimização do modelo.



Fonte autoral.

Como forma de otimização, pode-se, também, modificar a morfologia do modelo, tornando-o mais robusto, aumentar o tamanho do lote e, dessa forma, aumentar os exemplos levados em consideração durante uma iteração, utilizar otimizadores adaptativos, como ADAM ou ADAGRAD, para diminuir o tempo de treinamento, entre outras estratégias. Todavia, cada estratégia dessa possui suas peculiaridades e, portanto, seu uso deve ser observado com parcimônia.

Pode-se, ainda, testar o modelo com os dados de teste. Para isso, utiliza-se o método “.evaluate”. Para o problema posto, obteve-se uma acuracidade de 97,96% nestes dados de teste. O código em Python para isso é ilustrado pelo Quadro 26. Pode-se, ainda, utilizar o modelo para predizer um valor a partir de uma imagem sem rótulo por meio do método “.predict”. A Figura 13 mostra a predição de 20 imagens contidas no banco de dados de teste.

Quadro 26: Aplicação do modelo nos dados de teste.

```
model.evaluate(test_x, test_y)
↳
313/313 [=====] - 1s 2ms/step - loss:
0.0788 - acc: 0.9796
```

Fonte autoral.

Figura 13: Predições.



Fonte autoral.

5. REFERÊNCIAS BIBLIOGRÁFICAS

GÉRON, A. **Hands-On Machine Learning with Scikit-Learn and TensorFlow.**

NIELSEN, M. *Neural Networks and Deep Learning.* Determination Press, 2015.

California, EUA: O'Reilly Media, 2017.

PETERS, Tim. The zen of python. In: **Pro Python.** Apress, 2010. p. 301-302.