UNIVERSIDADE FEDERAL DE ALAGOAS

Mestrado Profissional em Matemática em Rede Nacional

PROFMAT

DISSERTAÇÃO DE MESTRADO

REDES NEURAIS NO ENSINO BÁSICO

ANDRÉ OLIVEIRA MARTINS

Instituto de Matemática

Maceió, 22 de janeiro de 2021

PROFMAT

UNIVERSIDADE FEDERAL DE ALAGOAS
INSTITUTO DE MATEMÁTICA
MESTRADO EM MATEMÁTICA EM REDE NACIONAL

ANDRÉ OLIVEIRA MARTINS

REDES NEURAIS NO ENSINO BÁSICO

Maceió
2020

André Oliveira Martins

Redes Neurais no Ensino Básico

Dissertação apresentada ao Programa de Mestrado Profissional em Matemática em Rede Nacional (PROFMAT) do Instituto de Matemática da Universidade Federal de Alagoas, como requisito parcial para obtenção do grau Mestre em Matemática.

Orientador: Prof. Dr. Márcio Henrique Batista da Silva

Maceió

2020

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecária: Taciana Sousa dos Santos – CRB-4 – 2062

M386r Martins, André Oliveira.
Redes neurais no ensino básico / André Oliveira Martins. – 2020.
126 f. : il., figs. e tabs. color.

Orientador: Márcio Henrique Batista da Silva.
Dissertação (Mestrado Profissional em Matemática) – Universidade Federal de Alagoas. Instituto de Matemática. Mestrado Profissional em Matemática em Rede Nacional. Maceió, 2021.

Bibliografia: f. 115-116.
Apêndices: f. 117-126.

1. Matemática. 2. Redes neurais artificiais. 3. Ensino básico. 4. Python (Linguagem de programação de computador). I. Título.

CDU: 51: 004.8

Folha de Aprovação

ANDRÉ OLIVEIRA MARTINS

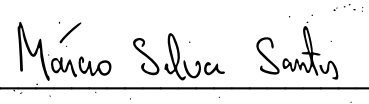
Redes Neurais no Ensino Básico

Dissertação submetida ao corpo docente do Programa de Mestrado Profissional em Matemática em Rede Nacional (PROFMAT) do Instituto de Matemática da Universidade Federal de Alagoas e aprovada em 22 de janeiro de 2021.

Prof. Dr. Márcio Henrique Batista da Silva – UFAL (Orientador)

Banca Examinadora:

Profa. Dra. Gabriela Albuquerque Wanderley – UFPB (Examinadora Externa)



Prof. Dr. Márcio Silva Santos – UFPB (Examinador Externo)

AGRADECIMENTOS

Agradeço Primeiramente, a Deus;

Aos meus pais, Laurentino Martins e Cremilda Gonzaga, que me ensinaram o real valor da família e os sentimentos de luta e coragem que hoje carrego e me conduzem em cada decisão;

As minhas irmãs Vanessa Martins e Valéria Martins pelo apoio, carinho e palavras de motivação;

A todos os colegas do curso, em particular Denise, Gabriel e Vitor pela amizade e apoio durante curso.

Agradeço ao corpo docente do PROFMAT da Universidade Federal de Alagoas em especial ao Prof. Dr. Márcio Henrique Batista da Silva pela dedicação, paciência e prontidão durante a realização deste trabalho;

Aos membros da Banca Examinadora, por aceitarem o convite de avaliar o resultado desta caminhada;

A todos, os meus sinceros agradecimentos.

"If you change the way you look at things, the things you look at change."

Wayne Dyer

RESUMO

Neste trabalho, propomos uma formação continuada para professores de matemática focada em Redes Neurais Artificiais, uma área de 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 Básico; Python.

ABSTRACT

In this work, we propose a training and developement of skills for mathematics teachers focus on Artificial Neural Networks, an area of artificial intelligence, which has attracted many scholars in the recent years. Introduce work presents how the artificial neural networks can be used in secondary education, 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 with codes in Python language.

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

SUMÁRIO

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 a realidade atual de seus alunos, tanto social como profissional.

O Novo Ensino Médio apresenta mudanças importantes para o ensino, sendo inseridas nas escolas a partir de 2021 e espera-se que as modificações estejam em prática até 2022. O Novo Ensino Médio Traz como propostas 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.

Percebe-se que tem ocorrido muitas transformações na educação, visto que o modelo tradicional de ensino já não se ajusta à sociedade atual que tem uma demanda diferente exigida principalmente pelo mercado de trabalho, colocando assim em destaque a necessidade de inserir novas práticas pedagógicas.

Nos dias atuais, as pessoas entram em contato com os mais diversos tipos tecnologias, desde muito cedo, antes mesmo de iniciar o processo de alfabetização, desse modo muitos alunos já chegam à escola bem familiarizados e tem grande facilidade de assimilar conhecimentos inovadores.

A tecnologia se insere cada vez mais rápido aproximando-se das diversas realidades e propõe mudanças para a sociedade moderna. Assim, o ensino e aprendizagem, tem ganhado redefinições que possibilitam o acesso ao conhecimento e informações de forma mais ampla.

A inteligência artificial está cada vez mais presente nas mais diversas áreas de atuação como segurança, saúde, além de estar presente no nosso dia a dia nas atividades financeiras, profissionais e nos mais diversos aparelhos domésticos como: TVs, media centers, câmeras de segurança, smartphones e computadores conectados à internet. Com a educação não tem sido diferente. Grandes mudanças já se inseriram nas escolas para proporcionar um ensino aprendizagem mais significativo. Contudo há ainda uma necessidade de buscar alternativas pedagógicas, para aumentar o uso desta tecnologia.

Desse modo, o presente trabalho tem por objetivo facilitar o acesso a um conhecimento mais amplo e significativo, utilizando sistemas de Inteligência Artificial como redes neurais e

deep learning. A relevância da temática se dá devido a necessidade da implantação da tecnologia no ambiente educacional. Desenvolvendo assim uma nova prática pedagógica nas escolas.

1 NOÇÕES BÁSICAS DE PYTHON

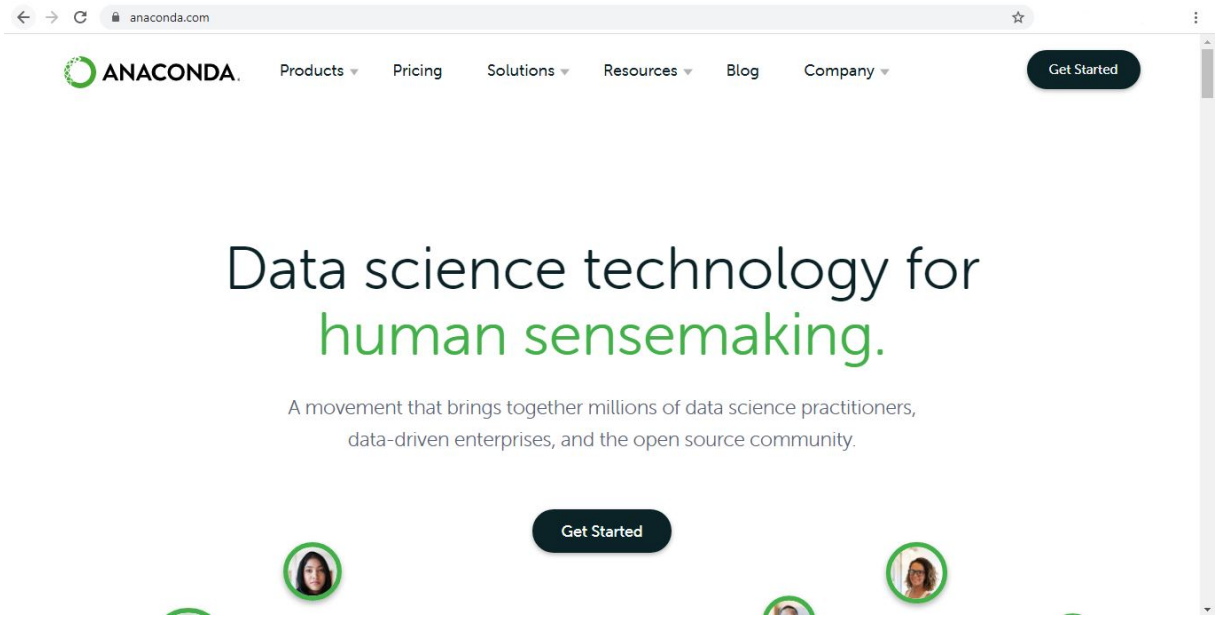
O Python é uma linguagem de programação muito intuitiva e de fácil aprendizado. Tal linguagem possui uma sintaxe concisa e clara, além de contar com muitas bibliotecas para análise de dados, tornando-a assim a linguagem mais adequada para machine learning.

1.1 Instalando o Anaconda

O Anaconda é um pacote, de código aberto, que agrega todas as ferramentas para análise de dados, em Python, em um único arquivo. Estão presentes no Anaconda o próprio Python, o Jupyter Notebook, a IDE Spyder e todas as bibliotecas que utilizaremos aqui, como o Numpy, MatPlotlib e Scikit-learn. Por sua praticidade, ao trazer tudo que precisamos em único arquivo, vamos instalar e usar o Anaconda, ao invés de instalar cada uma dessas ferramentas por vez.

O Anaconda está disponível para download gratuito no site https://www.anaconda.com/ , veja a imagem abaixo da página inicial do site.

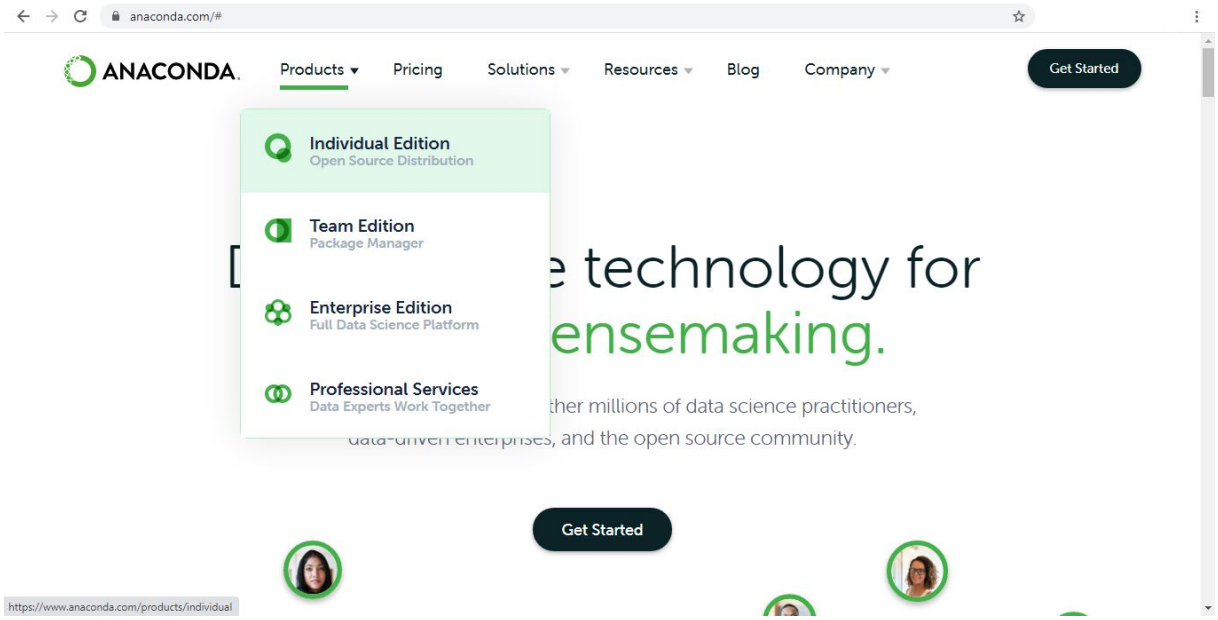
Figura 1 – Página inicial do site https://www.anaconda.com



Fonte: https://www.anaconda.com . Acesso: AGO, 2020.

Para fazer o download devemos clicar na guia *Products* e em seguida em *Individual edition*, veja a imagem abaixo.

Figura 2 – Caminho para download do Anaconda



Fonte: https://www.anaconda.com . Acesso: AGO, 2020.

Ao clicar em tal item, abre-se uma nova página e nesta encontraremos as opções de download e devemos baixar a versão adequada ao sistema operacional do computador em uso. Vai ser baixado a versão atual do Anaconda que traz o Python na versão 3.8.

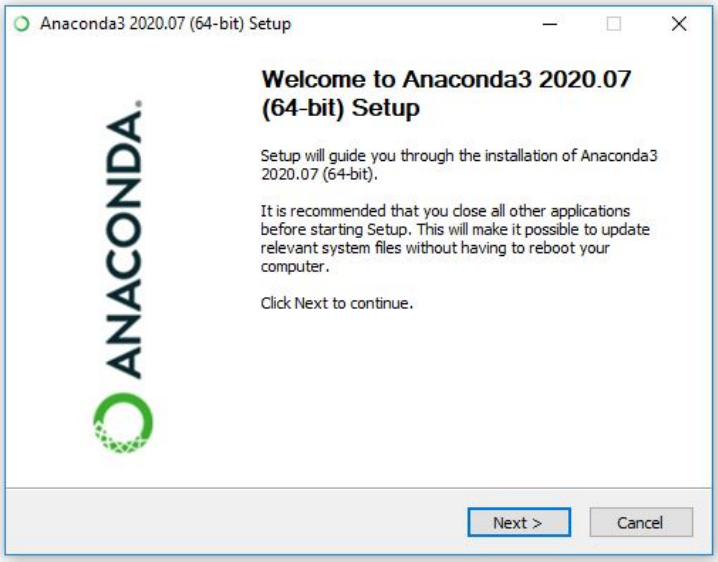
Figura 3 – Página de download do Anaconda

Fonte: https://www.anaconda.com . Acesso: AGO, 2020.

Após o download, clique duas vezes com o botão esquerdo do mouse no arquivo baixado para iniciar o processo de instalação. A primeira tela de instalação traz algumas informações,

clique em *Next*.

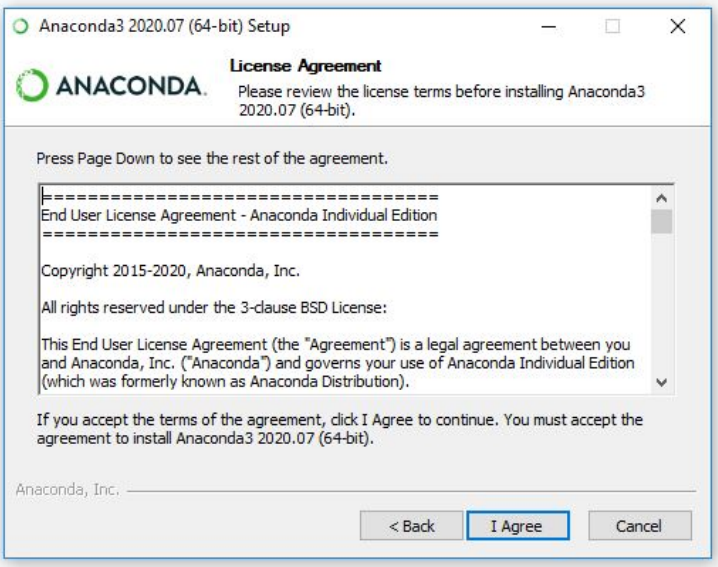
Figura 4 – Tela de instalação 1



Fonte: Instalação do Anaconda.

Em seguida, será mostrado os termos de licença, clique em *I agree* para aceitar.

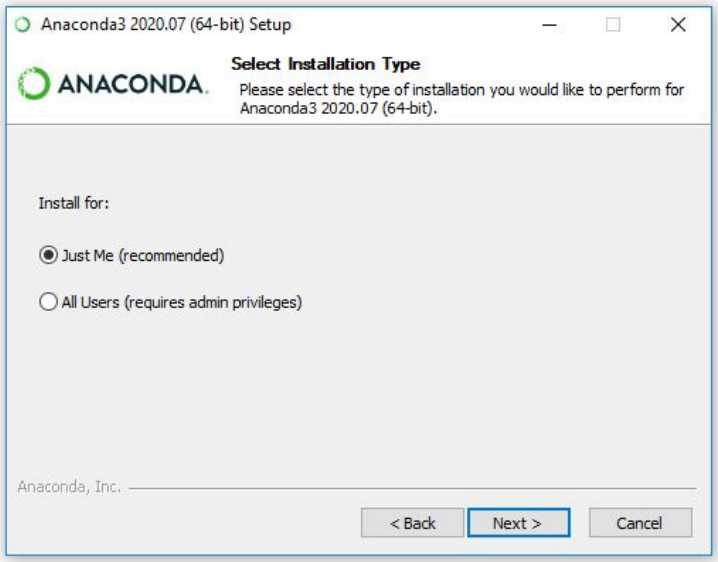
Figura 5 – Tela de instalação 2



Fonte: Instalação do Anaconda.

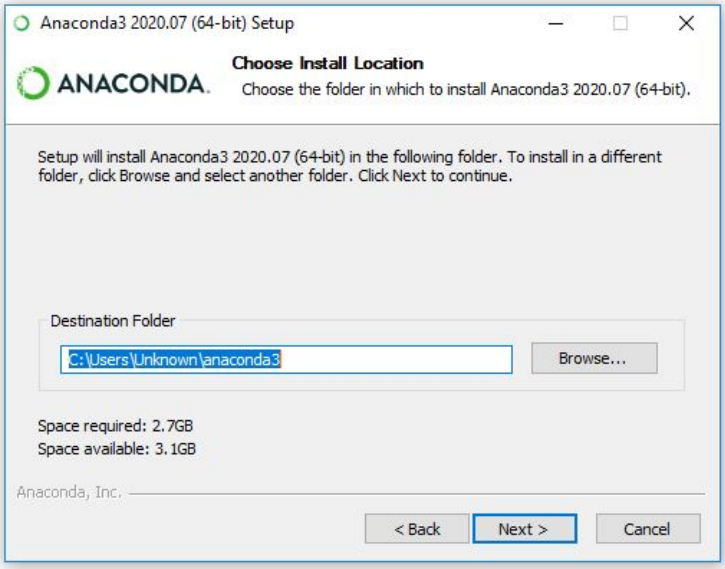
Para concluir a instalação basta seguir as telas abaixo.

Figura 6 – Tela de instalação 3



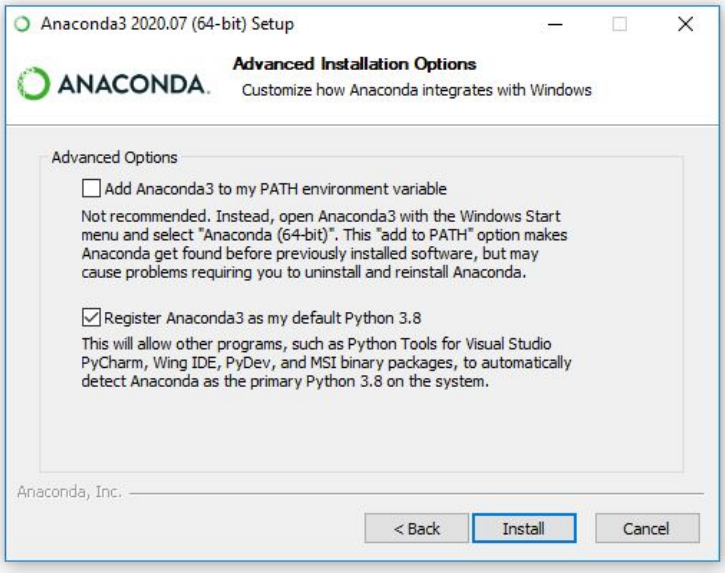
Fonte: Instalação do Anaconda.

Figura 7 – Tela de instalação 4



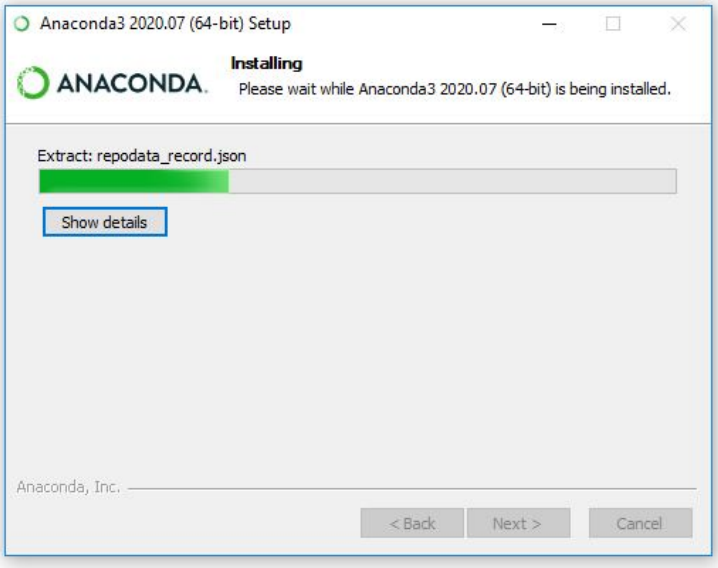
Fonte: Instalação do Anaconda.

Figura 8 – Tela de instalação 5



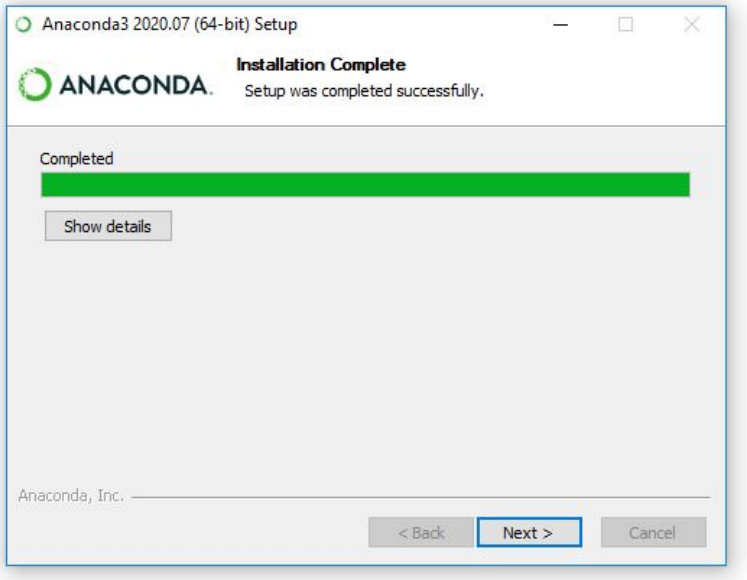
Fonte: Instalação do Anaconda.

Figura 9 – Tela de instalação 6



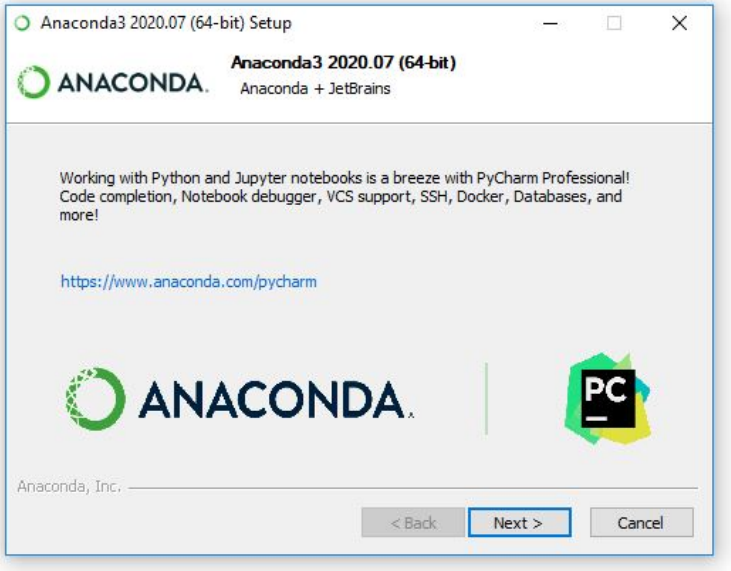
Fonte: Instalação do Anaconda.

Figura 10 – Tela de instalação 7



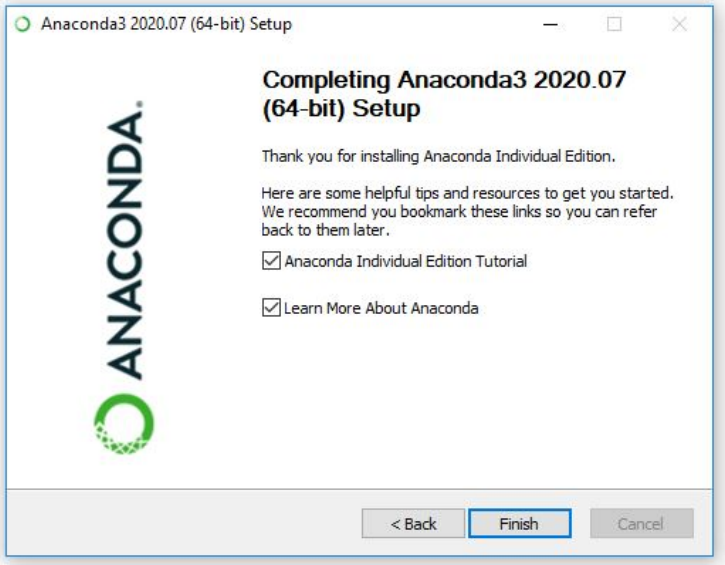
Fonte: Instalação do Anaconda.

Figura 11 – Tela de instalação 8



Fonte: Instalação do Anaconda.

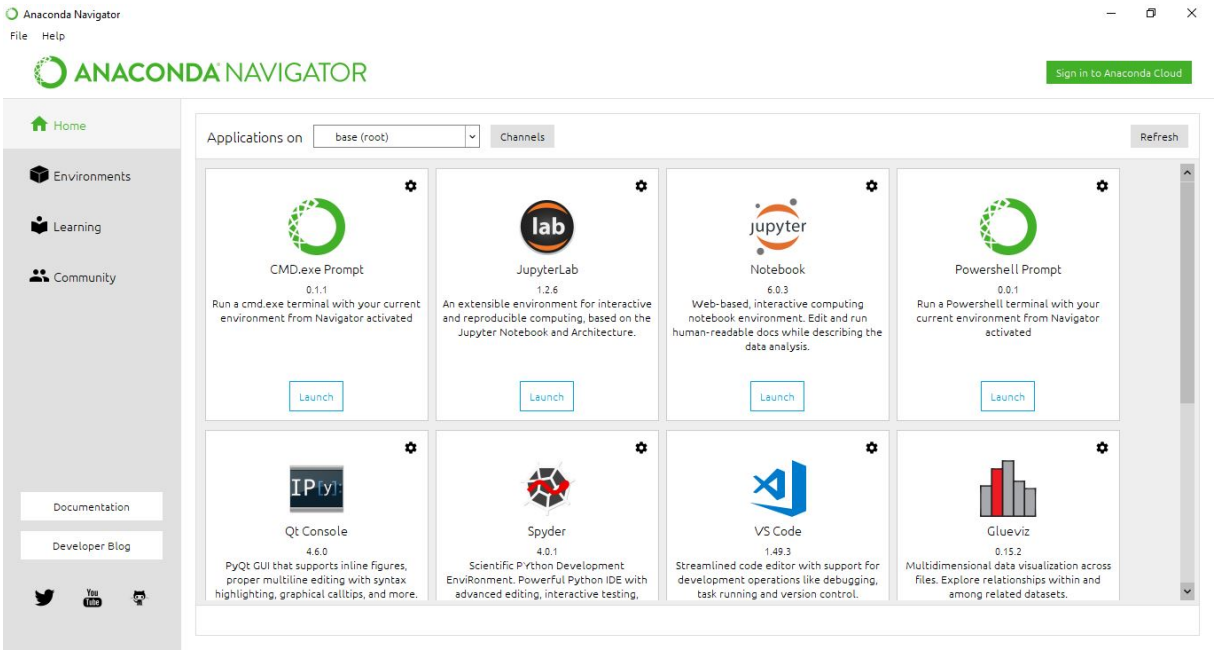
Figura 12 – Tela de instalação 9



Fonte: Instalação do Anaconda.

Com o Anaconda instalado temos agora, muitas ferramentas a nossa disposição e, entre elas, o Jupyter Notebook que é uma ferramenta que permite trabalhar com Python de forma simples e interativa. Para abrir o Jupyter Notebook, podemos procurá-lo na pasta do Anaconda ou abrir o Anaconda Navigator e clicar em *lunche* da opção Jupyter Notebook.

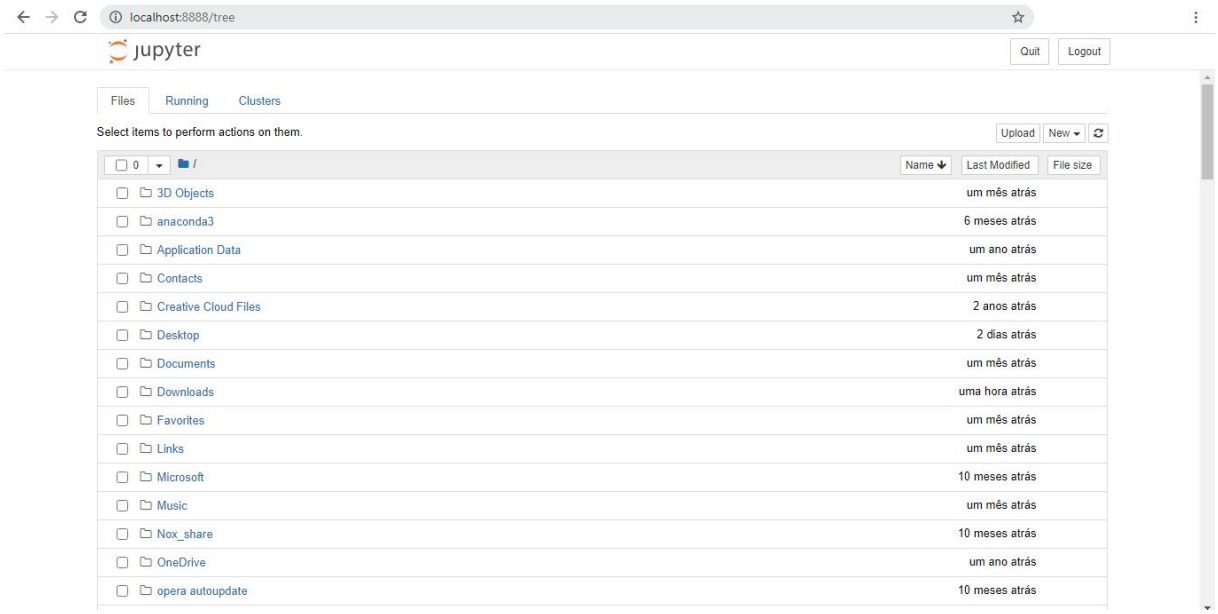
Figura 13 – Anaconda Navigator



Fonte: Elaborado pelo autor.

O Jupyter Notebook vai abrir no navegador padrão do computador, veja sua aparência na imagem abaixo.

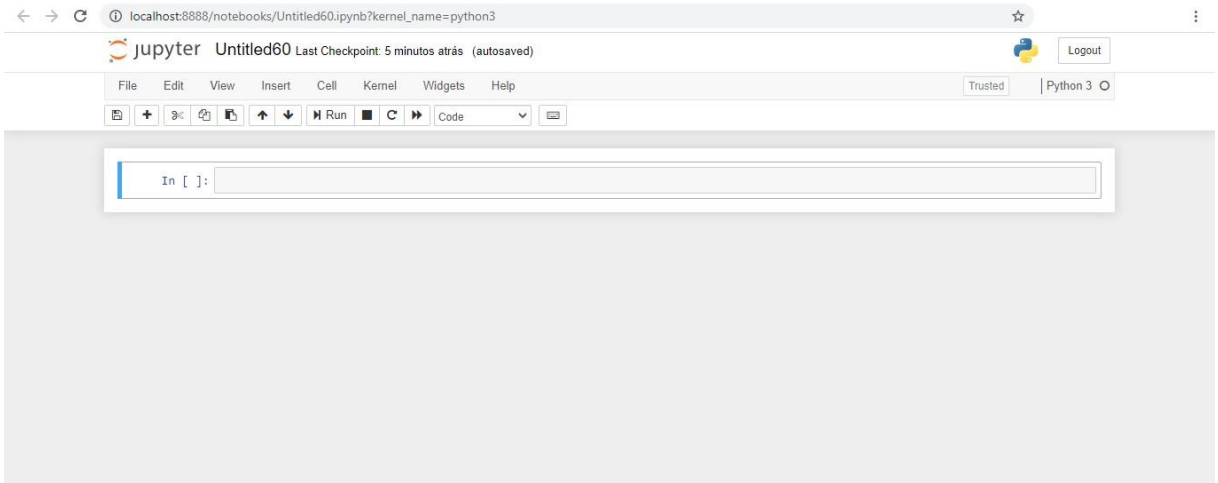
Figura 14 – Tela inicial do Jupyter Notebook



Fonte: Elaborado pelo autor.

Nesta página, podemos criar um novo arquivo Jupyter Notebook ou procurar um arquivo salvo anteriormente em uma pasta no computador. Para criar um novo arquivo clique em New e escolha Python 3. O arquivo será aberto em uma nova guia do navegador com o nome Untitled, basta clicar em cima deste nome para renomear para o nome desejado.

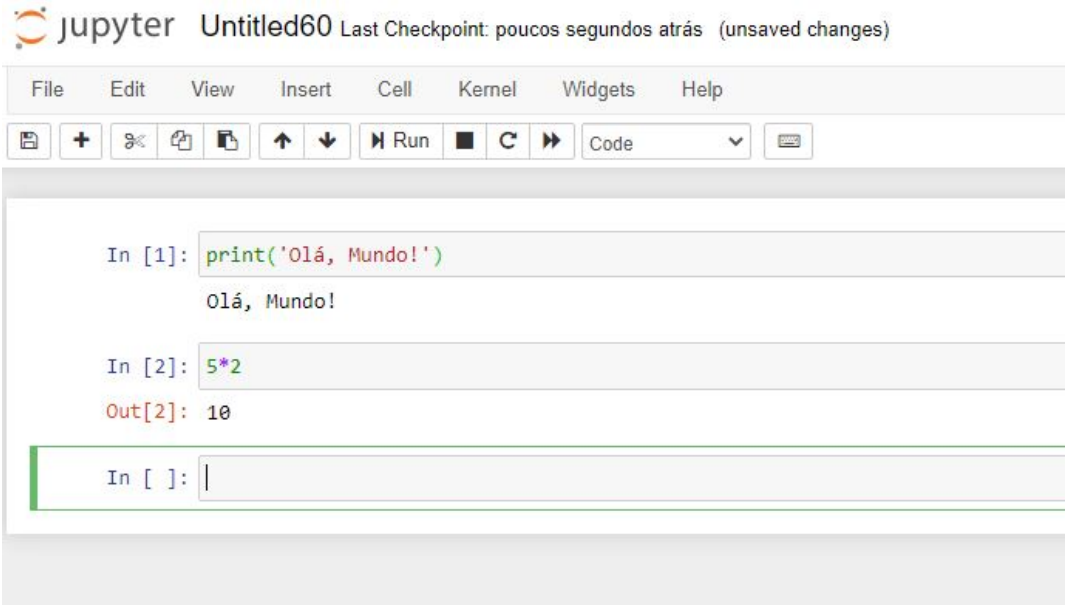
Figura 15 – Novo arquivo do Jupyter Notebook



Fonte: Elaborado pelo autor.

Perceba que temos apenas uma célula e podemos criar o primeiro código nesta célula e rodar o código clicando em *Run* ou pelo atalho *shift* + *enter*, o código será rodado e aparecerá uma nova célula. Podemos também, adicionar células clicando e +.

Figura 16 – Rodando código em célula do Jupyter Notebook



Fonte: Elaborado pelo autor.

1.2 O Interpretador Python

O Interpretador, instalado junto com o Python, é um programa com a capacidade de lê o código-fonte e traduzi-lo para linguagem do dispositivo. Os exemplos tratados aqui que misturam códigos e saídas serão formatados como no interpretador Python, apesar de terem sido feitos no Jupyter Notebook. Para ficar mais legível, as linhas de código serão antecedidas de >>> ou ... para blocos indentados e as saídas não serão precedidas de nenhum símbolo.

1.3 Variáveis

Variáveis são formas de armazenar dados para usá-los posteriormente. Para definir uma variável no Python devemos digitar o seu nome, o sinal de igualdade e o valor a ser armazenado. Os principais tipos de variáveis:

Numéricas: Quando armazena números inteiros (int) ou pontos flutuantes (float), isto é, números não inteiros.

Exemplo 1.1.

```
>>> a = 5
>>> b = 3.5
```

String: É uma sequência de caracteres que deve ser delimitada por aspas simples ('...') ou aspas duplas ("...") possibilitando a utilização de textos no Python.

Exemplo 1.2.

```
>>> str = 'O Python é uma linguagem de programação curva de aprendizado rá
    pido'
>>> a = 'A laranja é rica em vitamina C'
```

Podemos concatenar strings com o operado (+) e repeti-las com o operador $(*)$.

Exemplo 1.3.

```
>>> a = 'Eu'
>>> b = 'Tu'
>>> 3*(a + b)
    'EuTuEuTuEuTu'
```

Podemos também indexar (subscrever) e fatiar strings utilizando índices que da esquerda para a direita se inicia do 0 e da direita para esquerda inicia de -1.

Exemplo 1.4.

```
>>> palavra = 'Matemática'
>>> palavra[0]
    'M'
>>> palavra[-1]
    'a'
>>> palavra[0:3]
    'Mat'
```

```
>>> palavra[2:6]
    'temá'
>>> palavra[:6]
    'Matemá'
```

As strings do Python são imutáveis, isto é, atribuir a uma posição indexada resulta em um erro.

Exemplo 1.5.

```
>>> palavra = 'Matemática'
>>> palavra[0] = 'P'
Traceback (most recent call last):

  File "<ipython-input-36-bf5cc9f1185f>", line 1, in <module>
    palavra[0] = 'h'

TypeError: 'str' object does not support item assignment
```

Desse modo, em caso de necessidade de uma string diferente temos que criar uma nova.

A função *len()* retorna o comprimento da string.

Exemplo 1.6.

```
>>> palavra = 'Matemática'
>>> len(palavra)
    10
```

Variáveis devem ser iniciadas por letras minúsculas

O sinal (=) é usado para atribuir um valor a uma variável, por exemplo se criarmos a variável a e atribuirmos o valor 10, isto é, a = 10 Obs.: Dentro do Python a igualdade matemática é representada por dois sinais de igualdade, isto é, ==.

1.4 Operações matemáticas no Python

O interpretador do Python funciona como uma calculadora e, desse modo, podemos utilizar as operações matemáticas de maneira simples. Utilizamos, em Python, os operadores $+$, $-$, $*$ e $/$ de maneira usual e os parênteses $(())$ para agrupar as expressões.

Exemplo 1.7.

```
1
2 >>> 2 + 3
3     5
4 >>> (40-4*5)/2
5     10.0
6 >>> 15 / 7
7     2.142857142857143 # Divisão sempre retorna um ponto flutuante.
8 >>> 5**2
9     25
```

Obs.: Comentários em Python são introduzidos pelo caractere # e se estende até o final da linha física.

Utilizando o caractere (/) teremos sempre como retorno um ponto flutuante (float), para fazer uma divisão e receber um inteiro devemos utilizar o operador (//) e para calcular o resto de uma divisão devemos usar o (%).

Exemplo 1.8.

```
1 >>> 8 // 5
2     1
3 >>> 8 % 5
4     3
```

No Python para calcular potências devemos usar o operador (**).

Exemplo 1.9.

```
>>> 3 ** 2
    9
```

Para obter como resultado um número inteiro na divisão devemos operar com números inteiros, caso queiramos como resultado ponto flutuantes, devemos operar com pontos flutuantes.

1.5 Funções

1.5.1 input

A função input é destinada para interagir com o usuário, isto é, solicita dados que podem ter o formato numérico ou de string. O programa para e espera a digitação da informação seguida do ENTER.

Exemplo 1.10.

```
>>> nome = input('Qual é o seu nome?')
>>>print(nome, ", Seja bem-vindo!")
nome , Seja bem-vindo!
```

1.6 Ferramentas de controle de fluxo

1.6.1 Comandos *if*, *else* e *elif*

O comando *if*, se em português, é utilizado para executar determinada ação somente quando a condição dada for verdadeira. Em alguns casos é necessário executar ações de forma alternativa e para isso utilizamos o o comando *elif* que é a abreviação de *else if*, senão se em português, temos ainda, *else*, senão em português, usado executar determinada ação caso nenhuma das anteriores forem executadas.

Exemplo 1.11.

```
idade = int(input('Quantos anos você tem?:'))
>>>if idade < 18:
...     print('Você não é adulto.')
>>>elif idade >= 18 and idade < 65:
...     print('Você é adulto.')
>>>else:
...     print('Você é idoso')
```

1.6.2 Comandos *for* e *while*

Os comandos *for* e *while* são métodos de iteração em Python, o comando *for* em Python itera sobre os itens de qualquer sequência, na mesma ordem em que aparece na sequência, por sua vez, o comando *while* repete a instrução do seu bloco enquanto o condicional de seu cabeçalho for verdadeiro.

Exemplo 1.12.

```
>>>seq = [1, 2, 3]
>>>for item in seq:
...     print('Olá')
Olá
Olá
Olá
```

Exemplo 1.13.

```
>>>x = 1
>>>while x < 5:
...     print('Alô')
        x = x + 1
>>>else:
...     print(''End while)

Alô
```

```
9 Alô
10 Alô
11 Alô
12 End while
```

1.7 Funções

1.7.1 A função *range()*

A função *range()* gera progressões aritméticas, o que é muito útil caso se queira iterar sobre sequências numéricas. O ponto de parada fornecido não é incluído na lista. range (start, stop, step). Veja abaixo o significado dos parâmetros:

1. start é o valor do parâmetro inicial, se não for fornecido será 0;
2. stop é o valor do parâmetro final;
3. step é o valor do parâmetro incremento, se não for fornecido será 1.

Exemplo 1.14.

```
1 >>>range(0, 20, 3)
2 >>>b = list(range(0, 20, 3))
3 >>>print(b)
4     [0, 3, 6, 9, 12, 15, 18]
```

Exemplo 1.15.

```
1 >>>range(0, 50,3)
2 >>>a = list(range(0, 50,3))
3 >>>print(a)
4     [0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48]
```

Exemplo 1.16.

```
>>>a = range(1, 10)
>>>for i in range(1, 10):
...     print(i)
    1
    2
    3
    4
    5
    6
    7
    8
    9
```

1.7.2 A função *zip()*

A função zip() cria uma lista de tuplas, de tal modo que, a i-ésima tupla é formada pelos i-ésimos elementos de cada um de seus argumentos.

Exemplo 1.17.

```
>>>x = [1, 2, 3]
>>>y = ['a', 'b', 'c']
>>>for t in zip(x, y):
...     print(t)
(1, 'a')
(2, 'b')
(3, 'c')
```

Para visualizar as tuplas geradas pela função *zip()*, em forma de lista, devemos usar a função interna *list*.

Exemplo 1.18.

```
>>>x = [1, 2, 3]
>>>y = ['a', 'b', 'c']
>>>list(zip(x, y))
    [(1, 'a'), (2, 'b'), (3, 'c')]
```

1.8 Definindo funções

Para definir uma função devemos iniciar pelo comando *def*, seguido do nome da função e da lista de parâmetros formais entre parênteses. Os comandos que formam o corpo da função devem ser indentados e começam na linha seguinte e, por fim, a instrução *return* retorna um valor da função e finaliza a execução. Veja os exemplos abaixo.

Exemplo 1.19.

```
>>>def fib(n): # A função que imprime a sequência dos números de Fibonacci menores que n.
...     a, b = 0, 1 #Definimos o valor inicial para a e b.
...     while a < n:
...         print(a, end=' ')#Imprime o termo 'a' da sequência enquanto a for menor que n. o comando end=' ' insere um espaço entre os termos da sequência.
...         a, b = b, a+b # Itera fazendo a=b e b=a+b de forma recursiva.
...
# Agora chamaremos a função como definida
>>>fib(200)
    0 1 1 2 3 5 8 13 21 34 55 89 144
```

Exemplo 1.20.

```
>>>def media(a,b):
...     return (a+b)/2
>>>media(5,6)
    5.5
```

1.9 Funções lambda

As funções lambda reduzem o tamanho do código tornando o processo de programação mais prático.

Exemplo 1.21.

```
>>> a = lambda x: x**4
>>> a(3)
    81
```

1.10 Listas

Listas são estruturas de dados compostas usadas para agrupar itens. As listas *(list)* tem seus elementos separados por vírgula, entre colchetes.

Exemplo 1.22.

```
>>> primos = [2, 3, 5, 7, 11, 11, 13]
>>> primos
    [2, 3, 5, 7, 11, 11, 13]
```

De modo análogo as strings, as listas podem ser fatiadas e concatenadas retornando uma nova lista contendo os elementos solicitados.

Exemplo 1.23.

```
>>> primos = [2, 3, 5, 7, 11, 11, 13]
>>> primos[1:5]
    [3, 5, 7, 11]
>>> primos + [17, 19, 23]
    [2, 3, 5, 7, 11, 11, 13, 17, 19, 23]
```

As listas diferem das strings, pois são mutáveis, isto é, há possibilidade de alteração de elementos individuais de uma lista.

Exemplo 1.24.

```
>>> primos = [2, 3, 5, 7, 11, 11, 13]
>>> primos[2] = 29
>>> primos
    [2, 3, 29, 7, 11, 11, 13]
```

Podemos também adicionar novos elementos no final da lista, para isso, basta usar o método *.append()*

Exemplo 1.25.

```
>>> primos = [2, 3, 5, 7, 11, 11, 13]
>>> primos.append(17)
>>> primos
    [2, 3, 5, 7, 11, 11, 13, 17]
```

Podemos criar listas contendo outras listas.

Exemplo 1.26.

```
>>> primos = [2, 3, 5, 7, 11, 11, 13]
>>> letras = ['a', 'b', 'c']
>>> lista = [primos, 'casa', 'carro', letras]
>>> lista
    [[2, 3, 5, 7, 11, 11, 13], 'casa', 'carro', ['a', 'b', 'c']]
```

1.11 Tuplas

As Tuplas *tuple* são sequências imutáveis, diferentemente da lista não há possibilidade de adicionar ou remover elementos. Uma tupla tem a forma de uma sequência de dados separados por vírgulas e podem vir ou não entre parênteses.

Exemplo 1.27.

```
>>> t = 'casa', 10, 'olá!'
>>> t[2]#Observe que a contagem dos elementos de uma tupla começa do 0 da esquerda para a direita.
'olá!'
>>> t
('casa', 10, 'olá!')
t[1] = 4#Note que se ternarmos modificar um elemento de uma tupla teremos um erro.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

1.12 Dicionários

Os dicionários possuem uma estrutura similar as listas, onde cada um de seus elementos são uma combinação de chaves e valores. Em Python, os dicionários são criados utilizando chaves.

Exemplo 1.28.

```
>>> alimentos = {'Feijão': 5.80, 'Arroz': 3.50, 'Tomate': 3.30, 'Banana': 2.40 }
>>> alimentos['Feijão']
    5.8
```

1.13 Valores e operadores booleanos

No Python, uma variável pode assumir dois valores booleanos *True* (verdadeiro) ou *False*. Pode ser muito útil para comparação de variáveis.

Exemplo 1.29.

```
>>> a = 2
>>> b = 3
>>> a > b
    False
```

O Python suporta três operadores booleanos básicos que são *or*, *and* e *not* que são equivalentes respectivamente a disjunção $(\vee)$, conjunção $(\wedge)$ e a negação $(\neg)$. Cada um dos operadores booleanos segue as regras estabelecidas pela sua tabela verdade.

1.13.1 *Or*

O operador *or* só resulta *false* se seus dois operadores são *false*. O valor lógico do operador *or* é definido pela tabela verdade:

Tabela 1 – Tabela verdade do *or*

p	q	p *or* q
V	V	V
V	F	V
F	V	V
F	F	F

Fonte: Elaborado pelo autor.

Exemplo 1.30.

```
>>> a = False
>>> b = False
>>> a or b
    False
```

1.13.2 *And*

O operador *end* só resulta *True* se seus dois operadores forem *True*. O valor lógico do operador *and* é definido pela tabela:

Tabela 2 – Tabela verdade do *and*

p	q	$p\ and\ q$
V	V	V
V	F	F
F	V	F
F	F	F

Fonte: Elaborado pelo autor.

Exemplo 1.31.

```
>>> a = True
>>> b = True
>>> a e b
    True
```

1.13.3 *Not*

O operador *not* é utilizado para alterar o seu valor lógico de uma sentença, dando ideia contrária. Segue abaixo a tabela do operador *not*.

Tabela 3 – Tabela verdade do *not*

p	$not\ p$
V	F
F	V

Fonte: Elaborado pelo autor.

Exemplo 1.32.

```
>>> a = True
>>> not a
    False
```

Os Operadores padrões de comparação em Python são < (menor que), > (maior que), == (igual), <= (menor ou igual), >= (maior ou igual) e !=(diferente).

No Python temos a indentação como principal característica, usada para agrupar os comandos em blocos.

1.14 Classes

Podemos pensar uma classe como sendo um construtor de um objeto ou projeto para criação de objetos. Variáveis dentro de uma classe são chamados de atributos, já funções, quando dentro de classes, são chamadas de métodos.

Exemplo 1.33.

```
>>> class Exemplo:
...     def __init__(self, x, y):
...         self.a = x
...         self.b = y
...     def seq(self, n):
...         while self.a < n:
...             print(self.a, end=' ')
...             self.a, self.b = self.b, self.a + self.b
...         print()
>>> z = Exemplo(0, 1)
>>> z.seq(200)
    0 1 1 2 3 5 8 13 21 34 55 89 144
```

O método *__init__* define os valores iniciais da classe, que podem ser usados por outros métodos.

1.15 Comandos importantes

- *break* é um comando que tem como função sair de imediato do laço de repetição mais interno, seja *for* ou *while*.
- *pass* é um comando que não faz nada e é usado quando a sintaxe exige e não é necessária nenhuma ação ou, temporariamente, quando se está decidindo ou desenvolvendo o código para ocupar seu lugar.

1.16 Bibliotecas científicas do Python

1.16.1 NumPy

O NumPy é uma biblioteca científica do Python feita para trabalhar com álgebra linear. A classe array do Numpy é chamada de *ndarray* e oferece mais funcionalidades que a biblioteca array padrão do Python que trabalha apenas com matrizes unidimensionais.

Exemplo 1.34.

```
1 >>> import numpy as np
```

1.16.2 Pandas

O Pandas é uma ferramenta construída em linguagem Python para manipulação e análise de dados em código aberto. Devemos importá-lo usando o código abaixo

Exemplo 1.35.

```
1 >>> import pandas as pd
```

1.16.3 Matplotlib

É uma biblioteca muito utilizada em machine learning para visualização de dados através de plotagem dos mais variados tipos de gráficos. Para utilizamos essa biblioteca precisamos

importar com o camando *import* da biblioteca *matplotlib* a parte de plotar gráficos *pyplot* utilizando o comando como no exemplo abaixo

Exemplo 1.36.

```
1 >>> import matplotlib.pyplot as plt
```

2 MATEMÁTICA PARA REDES NEURAIS

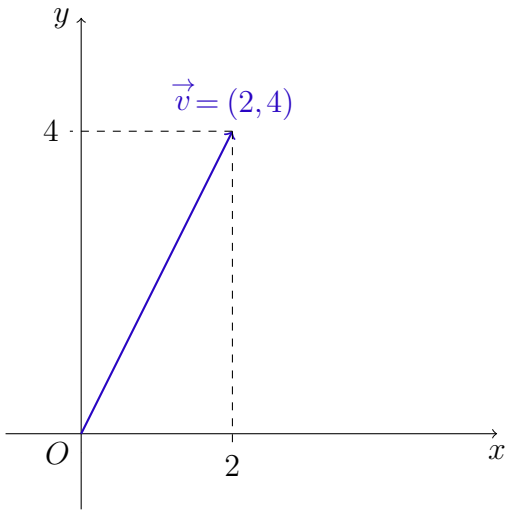
As redes neurais são modelos computacionais que modelam o cérebro humano, tais modelos são baseados em cálculos matemáticos, onde se usa fortemente conhecimentos de álgebra linear, como operações com matrizes, cálculo com funções de várias variáveis e máximos e mínimos. Assim, neste capítulo, veremos os principais conceitos matemáticos utilizados na construção de uma rede neural.

2.1 Vetores

Um vetor $\vec{v}$, do ponto de vista geométrico, é o conjunto de segmentos orientados munidos de mesmo **módulo**, **direção** e **sentido**.

- **Módulo**: É o comprimento do vetor $\vec{v}$ denotado por $\left\|\vec{v}\right\|$;
- **Direção**: Definimos a direção, como sendo a reta suporte que contém seus representantes, isto é, dois vetores $\vec{u}$ e $\vec{v}$ têm a mesma direção se possuírem representantes na mesma reta suporte;
- **Sentido**: Podemos definir sentido, em segmentos orientados com mesma direção, desse modo, se $\vec{u}=\vec{AB}$ e $\vec{v}=\vec{BA}$ dizemos que $\vec{u}$ e $\vec{v}$ tem sentidos contrários.

Figura 17 – Representação de um vetor no plano cartesiano



Fonte: Elaborada pelo autor.

Trataremos, neste trabalho, vetores como um ponto em algum espaço de dimensão finita. Nesse ponto de vista, vetores são excelentes ferramentas para representar dados numéricos.

Podemos realizar, com vetores, operações de adição, subtração e produto por um número real. Dados os vetores $\vec{u} = (u_1, u_2, u_3)$ e $\vec{v} = (v_1, v_2, v_3)$

- $\vec{u} + \vec{v} = (u_1 + v_1, u_2 + v_2, u_3 + v_3)$
- $\vec{u} - \vec{v} = (u_1 - v_1, u_2 - v_2, u_3 - v_3)$
- $k\,\vec{u} = k\,(u_1, u_2, u_3) = (ku_1, ku_2, ku_3)$

Exemplo 2.1. Sejam $\vec{u} = (1,3,2)$, $\vec{v} = (2,0,1)$ e k=5,

- $\vec{u} + \vec{v} = (1+2, 3+0, 2+1) = (3,3,3)$
- $\vec{u} - \vec{v} = (1-2, 3-0, 2-1) = (-1,3,1)$
- $5\,\vec{u} = 5\,(1,3,2) = (5\cdot 1, 5\cdot 3, 5\cdot 2) = (5,15,10)$

2.2 Matrizes

Definição 2.1. Uma matriz A do tipo $m \times n$ é uma lista de números a_{ij}, onde $1 \leq i \leq m$ e $1 \leq j \leq n$, dispostos em m linhas e n colunas, de forma que o elemento a_{ij} está localizado no cruzamento da i-ésima linha com a j-ésima coluna.

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m1} & \dots & a_{mn} \end{bmatrix}$$

Podemos criar uma matriz em Python usando a biblioteca NumPy de acordo com o código abaixo.

```
1 >>> a = np.array([[1, 2, 3], [4, 5, 6]])# Cria um array com 2 linha e 3
      colunas.
2 >>> a
3 ... array([[1, 2, 3],
4         [4, 5, 6]])
```

Vamos tratar as linhas e as colunas de uma matriz como vetores linha e coluna respectivamente.

2.2.1 Soma de matrizes

Definição 2.2. Sejam A e B duas matrizes do mesmo tipo $m \times n$, a soma $A+B$ é definida como a soma elemento a elemento.

$$A+B = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m1} & \dots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m1} & \dots & b_{mn} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \dots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & \dots & a_{2n}+b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}+b_{m1} & a_{m1}+b_{m1} & \dots & a_{mn}+b_{m1} \end{bmatrix}$$

Exemplo 2.2. Sendo $A = \begin{bmatrix} 2 & 1 & 4 \\ 4 & -2 & 0 \\ 6 & 0 & 1 \end{bmatrix}$ e $B = \begin{bmatrix} 7 & 0 & 5 \\ 1 & 0 & 1 \\ 3 & 8 & 2 \end{bmatrix}$ matrizes do mesmo tipo 3×3 a soma $A+B$

$$A+B = \begin{bmatrix} 9 & 1 & 9 \\ 5 & -2 & 1 \\ 9 & 8 & 3 \end{bmatrix}$$

2.2.2 Multiplicação de matrizes por um número

Seja A uma matriz $m \times n$ e k um número, o produto de k por M é definido como multiplicação k por cada elemento da matriz A.

$$kA = k\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m1} & \dots & a_{mn} \end{bmatrix} = \begin{bmatrix} ka_{11} & ka_{12} & \dots & ka_{1n} \\ ka_{21} & ka_{22} & \dots & ka_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ ka_{m1} & ka_{m1} & \dots & ka_{mn} \end{bmatrix}$$

Exemplo 2.3. Seja $A = \begin{bmatrix} 2 & 1 \\ 4 & -2 \\ 6 & 0 \end{bmatrix}$ o produto $3A$ é

$$3A = \begin{bmatrix} 6 & 3 \\ 12 & -6 \\ 18 & 0 \end{bmatrix}$$

2.2.3 Multiplicação de matrizes

Considere duas matrizes $A = (a_{ij})_{m \times n}$ e $B = (b_{jk})_{n \times p}$. O produto das matrizes A e B é uma matriz C tal que

$$c_{ik} = a_{i1}b_{1k} + a_{i2}b_{2k} + \ldots + a_{in}b_{nk} = \sum_{j=1}^{n} a_{ij}b_{jk}$$

Onde $1 \leq i \leq m$ e $1 \leq k \leq p$.

Observe que o produto usual de matrizes AB só existe se o número de colunas de A for igual ao número de linhas de B. Desse modo, a matriz AB é uma matriz com o mesmo número de linhas de A e mesmo número de colunas de B.

Exemplo 2.4. Dadas as matrizes $A = \begin{bmatrix} 0 & 1 & 4 \\ 4 & 0 & 1 \end{bmatrix}$ e $B = \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix}$ O produto de A por B, existe, pois as matrizes são respectivamente dos tipos 2×3 e 3×1, fazendo $C = AB$

$$AB = \begin{bmatrix} 0 \cdot 1 + 1 \cdot 0 + 4 \cdot 2 \\ 4 \cdot 1 + 0 \cdot 0 + 1 \cdot 2 \end{bmatrix} = \begin{bmatrix} 8 \\ 6 \end{bmatrix}$$

2.2.4 O produto Hadamard

O produto Hadamard de matrizes do mesmo tipo, também conhecido como produto de Schur, é muito utilizado em diferentes áreas em diversas aplicações. Dadas duas matrizes $A = (a_{ij})$ e $B = (b_{ij})$, do mesmo tipo $m \times n$, o produto Hadamard de A por B é definido por

$$A \odot B = a_{ij} b_{ij}$$

Exemplo 2.5. Dadas as matrizes $A = \begin{bmatrix} 2 & 4 & 0 \\ 1 & 1 & 5 \end{bmatrix}$ e $B = \begin{bmatrix} 0 & 2 & 0 \\ 1 & 3 & 4 \end{bmatrix}$ O produto Hadamard $A \odot B$, de A por B, existe, pois, as matrizes são do mesmo tipo 2×3.

$$A \odot B = \begin{bmatrix} 2 \cdot 0 & 4 \cdot 2 & 0 \cdot 0 \\ 1 \cdot 1 & 1 \cdot 3 & 5 \cdot 4 \end{bmatrix} = \begin{bmatrix} 0 & 8 & 0 \\ 1 & 3 & 20 \end{bmatrix}$$

2.3 Funções reais de uma variável real

Definição 2.3. Dado o conjunto $X \subset \mathbb{R}$ uma função de X em $\mathbb{R}$, isto é, $f : X \to \mathbb{R}$ é chamada de função real de uma variável real.

Exemplo 2.6. A função $f : \mathbb{R} \to \mathbb{R}$ definida por $f(x) = x^2 + 2x - 1$ é uma função real de uma variável real.

Definição 2.4. Sejam f uma função e $a \in D_f$ um ponto de seu domínio. Quando existir o limite

$$\lim_{x \to a} \frac{f(x) - f(a)}{x - a} = L \neq \pm\infty$$

fazendo $h = x - a$ temos $x = a + h$ teremos outra expressão da derivada

$$\lim_{h \to 0} \frac{f(a+h) - f(a)}{h} = L \neq \pm\infty$$

diremos que L é a derivada de f em a e escrevemos:

$$f'(a) = L$$

A próximas proposições trazem as regras de derivação que permitem o cálculo de derivadas sem o uso da definição.

Proposição 2.1. *Sejam f e g funções deriváveis em a e k uma constante. Então:*

(a) $\frac{dy}{dx}(k) = 0$;

(b) $(kf)'(a) = kf'(a)$;

(c) $(f+g)'(a) = f'(a) + g'(a)$;

(d) $(fg)'(a) = f'(a)\,g(a) + f(a)\,g'(a)$;

(e) $\left(\frac{f}{g}\right)'(a) = \frac{f'(a)g(a) - f(a)g'(a)}{[g(a)]^2}$.

Demonstração: Vamos demonstrar apenas o item (d), os outros itens são análogos. Seja $h(x) = f(x)g(x)$, da definição de derivada temos

$$\begin{aligned}
h'(a) &= \lim_{x\to a} \frac{h(x) - h(a)}{x - a} \\
&= \lim_{x\to a} \frac{f(x)\,g(x) - f(a)\,g(a)}{x - a} \\
&= \lim_{x\to a} \frac{f(x)\,g(x) - f(a)\,g(x) + f(a)\,g(x) - f(a)\,g(a)}{x - a} \\
&= \lim_{x\to a} \left[\frac{f(x) - f(a)}{x - a} g(x) + f(a)\frac{g(x) - g(a)}{x - a}\right] \\
&= f'(a)g(a) + f(a)g'(a)
\end{aligned}$$

Proposição 2.2. *Se $f(x) = x^n$ onde n é um inteiro positivo, então*

$$f'(x) = nx^{n-1}$$

Demonstração: *Utilizaremos a fórmula abaixo*

$$x^n - a^n = (x - a)(x^{n-a} + x^{n-a}a + \cdots xa^{n-2} + a^{n-1})$$

que pode ser verificada aplicando indução sobre n. Pela definição da derivada temos

$$\begin{aligned}
f'(a) &= \lim_{x\to a} \frac{f(x) - f(a)}{x - a} = \lim_{x\to a} \frac{x^n - a^n}{x - a} \\
&= \lim_{x\to a} (x^{n-a} + x^{n-a}a + \cdots xa^{n-2} + a^{n-1}) \\
&= a^{n-a} + a^{n-a}a + \cdots aa^{n-2} + a^{n-1} \\
&= na^{n-1}
\end{aligned}$$

Proposição 2.3. *Seja g e f duas funções reais de uma variável real tais que, g é derivável em x e f é derivável em $g(x)$. Então a função composta $h = f \circ g$ definida por $h(x) = f(g(x))$ é derivável em x e $h'(x)$ é dada por*

$$h'(x) = f'(g(x))g'(x)$$

Exemplo 2.7. Seja $h(x) = \left(x^2+1\right)^2$. Se fizermos $f(u) = u^2$ e $g(x) = x^2+1$, então $f'(u) = 2u$ e $g'(x) = 2x$ segue pela regra da cadeia que

$$h'(x) = 2(x^2+1)2x = 4x(x^2+1) = 4x^3+4x$$

2.4 Funções vetoriais

Uma função vetorial é uma função que tem como domínio o conjunto dos números reais e como imagem um conjunto de vetores.

Definição 2.5. Dados os conjuntos $I \subset \mathbb{R}$ e $\mathbb{R}^3$ uma função vetorial de $I \subset \mathbb{R}$ com valores em $\mathbb{R}^3$ é uma aplicação $\sigma : I \to \mathbb{R}^3$ tal que

$$\sigma(t) = (x(t), y(t), z(t)), \;\; t \in I, \;\; onde,$$

$x(t)$,$y(t)$ e $z(t)$ são funções reais definidas em I.

Dizemos que a função $\sigma(t)$ é contínua em I se $\sigma(t)$ é contínua para todo $t \in I$.

Quando $\sigma(t)$ é contínua em I, O vetor $\sigma(t) = (x(t), y(t), z(t))$ descreve uma curva C em $\mathbb{R}^3$. Para cada $t \in I$ obtemos um ponto $P = (x, y, z) \in C$, onde $x = x(t)$, $y = y(t)$ e $z = z(t)$.

A equação é chamada de parametrização da curva C e as equações acima são chamadas de equações paramétricas da curva C e a variável t é chamada de parâmetro.

Exemplo 2.8. Seja r uma reta que passa pelo ponto $P_0 = (1,0,2)$ e paralela ao vetor não nulo $\overrightarrow{u} = (1,1,1)$. Sendo $P = (x,y,z) \in r$, então $PP_0 = t\,\overrightarrow{u}$, isto é,

$$(x,y,z) = (1,0,2) + t\,(1,1,1)$$

Segue que,

$$\sigma(t) = (1+t, t, 2+t), \;\; t \in \mathbb{R}$$

é uma parametrização da reta r. Portanto, as equações paramétricas da reta r são:

$$x = 1+t,\ \ y = t\ \ e\ \ z = 2+t,\ \ t \in \mathbb{R}$$

Vamos estabelecer, na proposição abaixo, a regra da cadeia para funções vetoriais.

Proposição 2.4. *Seja $\sigma(u)$ uma função vetorial no intervalo I e u uma função real diferenciável de uma variável real t, com imagem no intervalo I, então*

$$\frac{d}{dt}\sigma(u(t)) = \frac{\sigma}{du}(u(t))\frac{du}{dt}(t)$$

2.5 Funções de várias variáveis

Definição 2.6. Dados os conjuntos $D \subset \mathbb{R}^n$ e $\mathbb{R}$. Uma função $f : D \subset \mathbb{R}^n \to \mathbb{R}$ é uma função real de n variáveis associa a cada n-upla $(x_1, x_2, \ldots, x_n) \in D \subset \mathbb{R}^n$ a um único número real $y = f(x_1, x_2, \ldots, x_n)$.

Exemplo 2.9. A função $z = f(x, y) = \frac{x^4}{x^2+y^2}$ é uma função de duas variáveis, onde seu domínio é o $\mathbb{R}^2$ menos os pontos tais que $x^2 + y^2 = 0$, isto é, o ponto $(0, 0)$.

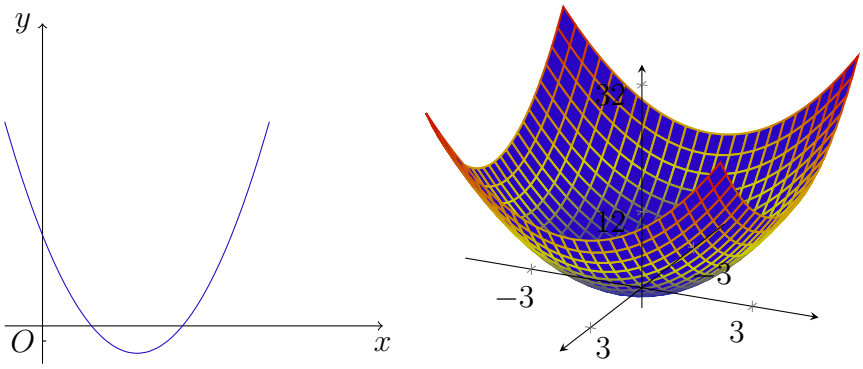
Definição 2.7. Sendo $f : D \subset \mathbb{R}^n \to \mathbb{R}$ uma função de n variáveis. O gráfico de f, denotado por G_f, é definido como um subconjunto de $\mathbb{R}^{n+1}$, isto é, os pontos do gráfico de f são da forma $(x_1, x_2, \ldots, x_n, z)$, onde $z = f(x_1, x_2, \ldots, x_n)$ com $(x_1, x_2, \ldots, x_n) \in D$, Assim,

$$G_f = \Big\{(x_1, x_2, \ldots, x_n, f(x_1, x_2, \ldots, x_n)) \in \mathbb{R}^{n+1} | (x_1, x_2, \ldots, x_n) \in D\Big\}$$

Desse modo, podemos visualizar somente os gráficos de funções de uma e de duas variáveis, que são subconjuntos do $\mathbb{R}^2$ e do $\mathbb{R}^3$ respectivamente, funções de três ou mais variáveis não são visualizáveis pois são subconjuntos de quatro ou mais dimensões.

Exemplo 2.10. Veja abaixo gráficos de funções de uma e de duas variáveis.

Figura 18 – Gráficos de uma e de duas variáveis



Fonte: Elaborada pelo autor.

2.5.1 Derivadas parciais

Definição 2.8. Seja $f(x,y)$ uma função de duas variáveis definido em um conjunto I da forma $(a_1,b_1)\times(a_2,b_2)$ e (x_0,y_0) um ponto de I. A derivada parcial de f com relação a x no ponto (x_0,y_0) é definida por

$$\frac{\partial f}{\partial x}=\lim_{\Delta x\to 0}\frac{f(x_0+\Delta x,y_0)-f(x_0,y_0)}{\Delta x},$$

se o limite existir.

De modo análogo, a derivada parcial de f com relação a y no ponto (x_0,y_0) é definida por

$$\frac{\partial f}{\partial y}=\lim_{\Delta y\to 0}\frac{f(x_0,y_0+\Delta y)-f(x_0,y_0)}{\Delta y},$$

se o limite existir.

A definição das derivadas parciais para funções de n variáveis onde $n\geq 3$ é idêntico ao caso de duas variáveis.

Exemplo 2.11. Seja $f(x,y)=3x_2+y_3$. Calcule $\frac{\partial f}{\partial x}$ e $\frac{\partial f}{\partial y}$.

Solução: Para calcular $\frac{\partial f}{\partial x}$ devemos considerar y como uma constante e derivar f com relação a x do mesmo modo como em funções de apenas uma variável, desse modo,

$$\frac{\partial f}{\partial x}=6x$$

Analogamente, consideramos x constante e derivando com relação a y, temos

$$\frac{\partial f}{\partial y} = 3y_2$$

A próxima proposição traz a regra da cadeia para funções de duas variáveis.

Proposição 2.5. *Seja $z = f(x,y)$ uma função derivável nas variáveis x e y, onde $x = g(t)$ e $y = h(t)$ são funções deriváveis de t. Então z é uma função diferenciável de t e*

$$\frac{\partial z}{\partial t} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$

Definição 2.9. Seja f uma função de n variáveis $(x_1, x_2, \ldots, x_n)$ que possui derivadas parciais no ponto $P_0 = (x_1^0, x_2^0, \ldots, x_n^0)$. O vetor gradiente de f em P_0, é o vetor

$$\nabla f(P_0) = \left(\frac{\partial f}{\partial x_1}(P_0), \ldots, \frac{\partial f}{\partial x_n}(P_0)\right)$$

.

2.6 Máximos e mínimos

Definição 2.10. Sendo $a \in X$ e dado um número real $r > 0$. A bola aberta de centro a e raio r é o conjunto $B(a,r)$ dos pontos cuja distância ao ponto a é menor que r. isto é,

$$B(a,r) = \{x \in X, d(x,a) < r\}$$

Definição 2.11. Seja $z = f(x,y)$ uma função real definida em um conjunto $D \subset \mathbb{R}^2$ e $(x_0, y_0) \in D$. Chamamos de valor mínimo relativo de f (resp. valor máximo relativo de f) se existir uma bola aberta $B_r(x_0, y_0) \subset D$ de modo que

$$f(x,y) \geq f(x_0,y_0) \;\; (resp. f(x,y) \geq f(x_0,y_0))$$

,

Para todo (x,y) em B.

A proposição abaixo estabelece uma condição necessária para a existência de máximos e mínimos relativos.

Proposição 2.6. *Sejam $z = f(x, y)$ uma função definida em $D \subset R^2$ cujo interior é não vazio, e (x_0, y_0) pertence ao interior de D. Se $\frac{\partial f}{\partial x}(x_0, y_0)$ e $\frac{\partial f}{\partial y}(x_0, y_0)$ existem e $f(x, y)$ tem um máximo ou um mínimo relativo em (x_0, y_0) então*

$$\frac{\partial f}{\partial x}(x_0, y_0) = 0 \ \ e \ \ \frac{\partial f}{\partial y}(x_0, y_0) = 0$$

.

Definição 2.12. Um ponto (x_0, y_0) pertencente ao interior do domínio de $f(x, y)$ é chamado ponto crítico de f se $\nabla f(x_0, y_0)$ não existe ou $\nabla f(x_0, y_0) = (0, 0)$

Definição 2.13. Um ponto (x_0, y_0) pertencente ao interior do domínio de $f(x, y)$ é chamado ponto crítico de f se $\nabla f(x_0, y_0)$ não existe ou $\nabla f(x_0, y_0) = (0, 0)$

A natureza de um ponto crítico poderá ser analisada pelo teste da segunda derivada de acordo com a proposição abaixo.

Proposição 2.7. *Seja $f(x, y)$ uma função de classe C^2 dentro da bola aberta $B_r(x_0, y_0)$. Suponha que (x_0, y_0) é um ponto crítico de $f(x, y)$, isto é, $\nabla f(x_0, y_0) = (0, 0)$. Fazendo*

$$A = \frac{\partial^2 f}{\partial x^2}(x_0, y_0), \ \ B = \frac{\partial^2 f}{\partial x \partial y}(x_0, y_0) \ \ e \ \ C = \frac{\partial^2 f}{\partial y^2}(x_0, y_0).$$

Se

(i) $B^2 - AC < 0$ *e* $A < 0$*, então f tem um valor máximo relativo em (x_0, y_0);*

(ii) $B^2 - AC < 0$ *e* $A > 0$*, então f tem um valor mínimo relativo em (x_0, y_0);*

(iii) $B^2 - AC >$*, então f tem um ponto de cela em (x_0, y_0).*

3 NOÇÕES DE APRENDIZADO DE MÁQUINA

O aprendizado de máquina é um ramo da inteligência artificial que trabalha com modelos capazes de reconhecer padrões a partir dos dados apresentados. Este tipo de inteligência tem uso imprescindível em tarefas computacionais em que algoritmos explícitos não são viáveis.

3.1 Inteligência artificial

A inteligência é a capacidade de tomar a melhor decisão possível, dada a informação disponível, com a capacidade de se adaptar a novas situações. A inteligência artificial é, por sua vez, a capacidade das máquinas de simular os seres humanos, isto é, resolver problemas e tomar decisões inteligentes.

3.2 Aprendizado de máquina

Também chamado de modelo preditivo ou mineração de dados é um campo da ciência da computação que cria modelos que evoluem, isto é, aprendem a partir dos dados que absorvem e através do reconhecimento de padrões.

Definição 3.1. Um algoritmo é uma sequência finita de instruções, onde cada uma delas são executadas em um tempo finito, com as quais podemos transformar a entrada na saída desejada.

Exemplo 3.1. Em uma soma, a entrada são os dois valores que desejamos somar e sua respectiva saída é o resultado da soma.

3.3 Categorias de aprendizado de máquina

Existem vários tipos de aprendizado de máquina, daí surge a necessidade de categorizá-los tomando como base suas características. Por exemplo, se são ou não treinados com supervisão humana, se podem ou não aprender rapidamente, se comparam dados novos com os dados conhecidos ou se encontram padrões nos dados de treinamento e criam um modelo preditivo.

3.3.1 Aprendizado de máquina supervisionado

O aprendizado de máquina supervisionado ocorre quando os dados usados para treinar o algoritmo incluem a solução desejada, classe ou rótulo ("label"), isto é, tem um conjunto de dados rotulados que traz a resposta certa do problema.

Exemplo 3.2. O filtro de spam é um exemplo aprendizado de máquina supervisionado, pois ele é treinado com um conjunto de e-mails, onde uma parte são classificados como spam e a outra parte como não spam, com o objetivo que o filtro aprenda a classificar novos e-mails.

Podemos realizar dois tipos de tarefas de aprendizado supervisionado:

- **Classificação:** Quando a variável a ser predita é qualitativa, neste caso, o objetivo é fazer a previsão de classes.
- **Regressão:** Ocorre quando a variável a ser predita é quantitativa.

3.3.2 Aprendizado de máquina não supervisionado

No aprendizado de máquina não supervisionado, a máquina recebe um conjunto de entradas, mas nenhum conjunto de saídas correspondente. Nesse caso, cabe à máquina encontrar padrões de semelhança e diferenças entre os dados e, com esses padrões, gerar novas saídas corretas. Não existe rótulo ("label") o algoritmo aprende sem uma resposta certa.

Exemplo 3.3. A identificação dos perfis de clientes que compram em uma loja, onde o objetivo é agrupá-los em várias categorias de acordo com semelhanças e diferenças.

3.3.3 Aprendizado de máquina semi-supervisionado

Em aprendizado de máquina, podemos trabalhar com conjuntos de dados onde existem dados com rótulo e uma grande quantidade de dados sem rótulo, nesse caso, dizemos que se trata de aprendizado de máquina semi-supervisionado. A maioria dos algoritmos de aprendizado semi-supervionado são uma combinação dos algoritmos de aprendizado supervisionando com os de aprendizado não supervisionado.

Exemplo 3.4. Um bom exemplo disso é a identificação de fotos em redes sociais. O algoritmo identifica a mesma pessoa em várias fotos (não supervisionado) e depois só precisa de um rótulo(supervisionado).

3.3.4 Aprendizado de máquina por reforço

Em Aprendizado de máquina por reforço há interação com o ambiente dinâmico, com o objetivo de executar ações e obter, em troca, feedbacks em termos de premiações e punições. O aprendizado ao longo do tempo tem com o objetivo realizar as ações com as melhores estratégias. Um bom exemplo, onde se usa aprendizado por reforço, é em robôs para que aprendam a andar.

3.3.5 Aprendizado online ou em lote

No aprendizado online o sistema aprende de modo incremental a partir de um fluxo de dados recebido, enquanto no aprendizado em lote o sistema não é capaz de aprender de forma incremental, isto é, é treinado com a quantidade de dados disponível e geralmente feito offline, o sistema depois de treinado roda sem aprender mais nada.

3.3.6 Aprendizado baseado em instâncias

O sistema aprende ao memorizar os exemplos e, posteriormente, generaliza para novos casos analisando a similaridade.

3.3.7 Aprendizado baseado em modelo

No aprendizado baseado em modelo constrói-se um modelo a partir de um conjunto de exemplos o qual é usado para fazer previsões.

3.4 Sobreajuste e sub-ajuste

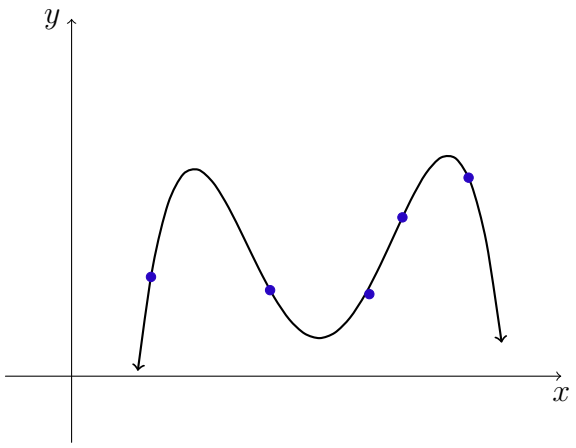
Um projeto de aprendizado de máquina pode não funcionar bem, isto é, pode não ter o desempenho esperado. Isso pode acontecer devido a dois fatores, um deles é que o algoritmo pode ser ruim, outro é que os dados podem ser ruins. Quando se faz um treinamento com o

conjunto de dados ruim, isto é, que não é representativo, provavelmente, não haverá previsões satisfatórias.

3.5 Sobreajuste

É um modelo que funciona bem com os dados de testes, mas não tem bom desempenho com dados novos, isto é, não generaliza muito bem. O sobreajuste acontece normalmente devido a quantidade ou a qualidade do conjunto de dados de treinamento. Para evitar o sobreajuste pode ser necessário usar um modelo mais simples, com menos parâmetros, coletar mais dados ou reduzir os ruídos dos dados de treinamento.

Figura 19 – Gráfico de um modelo com sobreajuste



Fonte: Elaborada pelo autor.

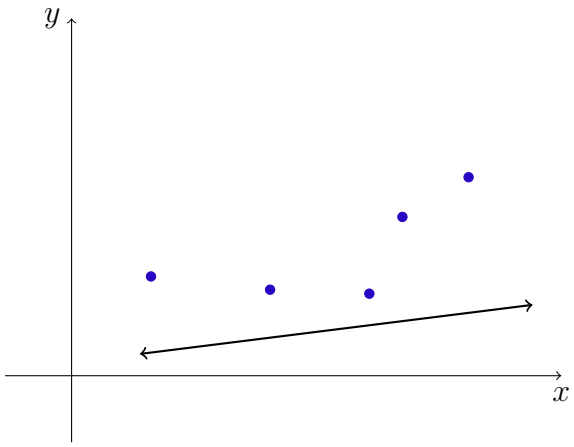
3.6 Sub-ajuste

Não funciona bem nem com os dados de treino nem com os dados novos. Normalmente ocorre quando o modelo é muito simples para um aprendizado complexo em relação a conjunto de dados, daí a tendência de imprecisão de suas predições. Nesse caso, para solucionar o problema de sub-ajuste se torna necessário escolher um modelo melhor, com mais parâmetros.

Para tentar evitar problemas como sobreajuste e sub-ajuste podemos dividir o conjunto de dados da seguinte forma: dois terços dos dados serão usados para treinar o modelo e o um terço restante usado para testar o modelo. É comum dividir o conjunto de forma que 80% dos dados sejam utilizados para treinar o modelo e 20% para testá-lo.

Exemplo 3.5. O gráfico abaixo representa um treinamento que teve um sub-ajuste.

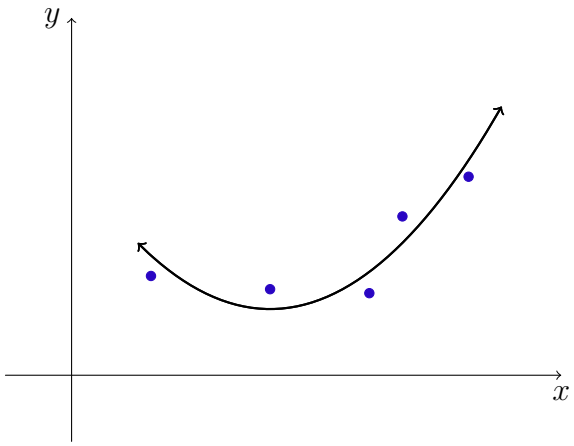
Figura 20 – Gráfico de um modelo com sub-ajuste



Fonte: Elaborada pelo autor.

Exemplo 3.6. O gráfico abaixo representa um treinamento que faz boas previsões.

Figura 21 – Gráfico de um modelo com um bom treinamento



Fonte: Elaborada pelo autor.

4 REDES NEURAIS

As redes neurais artificiais são modelos matemáticos que apresentam um grande poder de processamento e que se estabelece na estrutura fisiológica do cérebro humano com objetivo de reproduzir suas funções. Se faz necessário, antes de mais nada, buscar compreender como as redes neurais funcionam, conhecendo os estudos que proporcionaram o surgimento dessa área.

Diferentemente do que se pensa hoje em dia, a teoria das redes neurais não é nova. 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.

Surgiram diversos estudos que impulsionaram a origem de vários modelos de redes neurais que são aplicados atualmente. Porém, a aplicação das técnicas de redes neurais apontava para algumas limitações, impossibilitando assim, a utilização na solução de alguns problemas.

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.

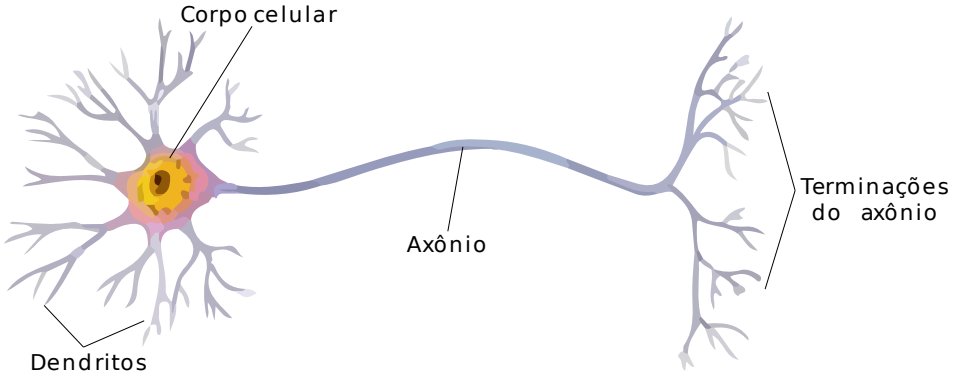
A partir dos anos 2000, as redes neurais voltaram ao centro das atenções devido ao crescimento do poder de processamento dos computadores além da quantidade de dados disponíveis, o que facilitou a implementação e o treinamento dessas redes e consequentemente avanço do Deep Learning.

As Redes Neurais Artificiais ou apenas redes neurais, como foi dito, são modelos de inteligência artificial inspirado no funcionamento do cérebro humano, elas 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. Este 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.

4.0.1 Neurônio biológico

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

Figura 22 – Neurônio biológico



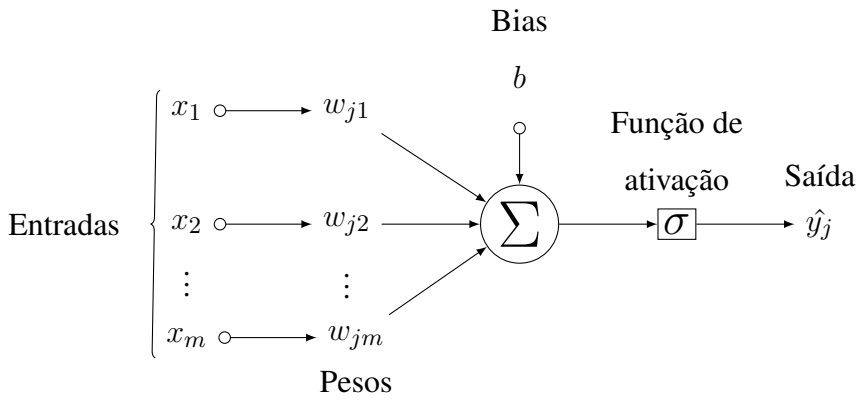
Fonte: https://metodosupera.com.br/neuronios-glossario-do-cerebro/. Acesso: OUT, 2020.

Os dendritos tem 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 chamada de sinapse, desse modo, o sinal do neurônio flui da esquerda para a direita transmitindo estímulos elétricos através de sinapses.

4.0.2 Neurônio artificial

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 23 – Neurônio artificial



Fonte: https://tex.stackexchange.com/questions/132444/diagram-of-an-artificial-neural-network. Acesso: OUT, 2020.

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. Desse modo, o integrador representado pele letra Σ realiza a soma das entradas ponderadas por esses pesos, isto é, $u_j = \sum_{i=1}^{m} w_{ji}x_i$. Note que a posição dos índices dos pesos w_{ji} são invertidas, o primeiro índice (j) se refere ao neurônio e o segundo índice (i) se refere a entrada ou ao neurônio da camada anterior, no caso de camada oculta ou camada de saída, isto é, indica de onde vem o sinal de ativação. Adicionando o termo de bias obtemos o campo local induzido $v_j = u_j + b_j$. A saída da rede é obtida aplicando a função de ativação σ em v_j. Podemos simplificar esses cálculos usando representação matricial como segue.

$$u_j = \sum_{i=1}^{m} w_{ji}x_i = w_{j1}x_1 + w_{j2}x_2 + \ldots + w_{jm}x_m = \mathbf{x}^T\mathbf{w}$$

$$v_j = u_j + b_j$$

Portanto,

$$\hat{y_j} = \sigma(v_j)$$

onde

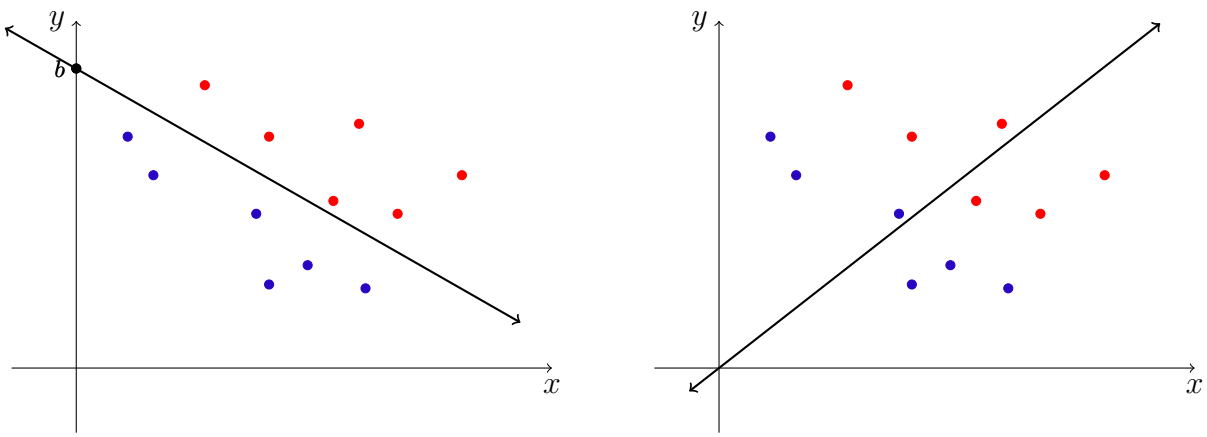
- $\mathbf{x} = [x_1, x_2, \ldots, x_m]^T$ é o vetor onde suas componentes são os sinais de entrada;
- $\mathbf{w} = [w_{j1}, x_{j2}, \ldots, x_{jm}]^T$ é o vetor onde suas componentes são os pesos sinápticos de ligação das unidades de entrada com neurônio j;
- u_j é o sinal de saída do integrador linear devido aos sinais de entrada;
- b_j é termo de bias, que é um parâmetro externo do neurônio
- v_j é chamado de campo local induzido;
- $\sigma(\bullet)$ é uma função de ativação;
- $\hat{y_j}$ é o sinal de saída do neurônio.

4.0.3 O termo bias

O termo bias é acrescentado na equação do campo local induzido $v = w_{ji}x_i + b_j$ que é a 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 se a reta para pela origem, acima ou abaixo da origem, desse modo, o bias aumenta a liberdade de posicionamento da reta. A reta que separa as duas classes de um conjunto de dados binários é chamada reta de decisão.

Exemplo 4.1. Veja os gráficos abaixo, se não existir um bias diferente de 0 a reta não conseguirá separar as classes, pois só poderá rotacionar em torno da origem durante o treinamento.

Figura 24 – A importância do termo bias



Fonte: Elaborada pelo autor.

Podemos trabalhar, equivalentemente, com o termo bias embutido, isto é, adicionando a entrada

$$x_0 = 1$$

e o seu respectivo peso é

$$w_{j0} = b_j$$

Desse modo, o vetor de entrada terá como primeira componente 1, $\mathbf{x} = [1, x_1, x_2, \ldots, x_m]^T$ e o vetor de pesos terá como primeira componente o termo de bias b_j, $\mathbf{w} = [b_j, w_{j1}, x_{j2}, \ldots, x_{jm}]^T$. Assim podemos reescrever a equação do campo local induzido da seguinte forma.

$$v_j = \sum_{i=0}^{m} w_{ji} x_i.$$

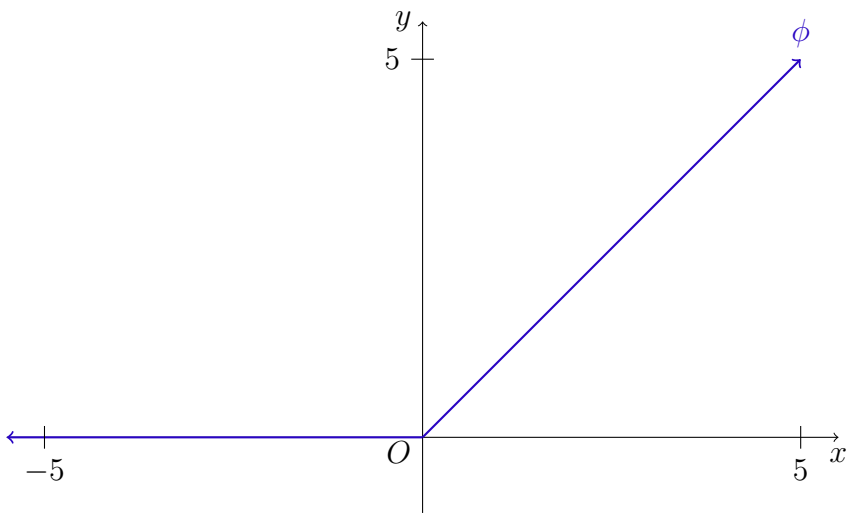
4.0.4 Funções de ativação

As funções de ativação são transformações não-lineares, feitas ao longo do sinal de entrada e enviada para a próxima camada de neurônios como uma nova entrada, em caso de camada oculta. Essas funções definem a saída de um neurônio em um intervalo, isto é, restringem a amplitude de um neurônio, normalmente saída de um neurônio é representado como um intervalo normalmente $[0, 1]$ ou $[-1, 1]$. As principais funções de ativação são:

- ***Função ReLU:*** A função de ativação linear retificada (Rectified Linear Unit - ReLU) é uma função linear por partes que se tornou a mais popular função de ativação na atualidade, sendo padrão em algumas redes neuras devido a facilidade do treinamento e do desempenho alcançado.

$$\phi(u) = \begin{cases} u, & \text{se} \quad u \geq 0 \\ 0, & \text{se} \quad u < 0 \end{cases}$$

Figura 25 – Gráfico da função de ativação ReLu



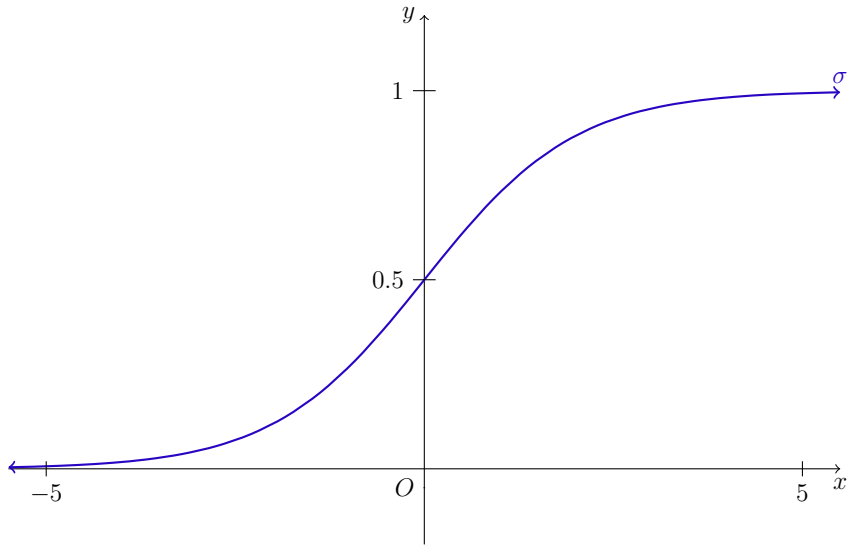
Fonte: Elaborada pelo autor.

- ***Função Sigmoide:*** A função sigmoide definida por

$$\sigma(u) = \frac{1}{1 + exp(-au)}.$$

Veja o gráfico abaixo da função sigmóid para $a = 1$.

Figura 26 – Gráfico da função de ativação sigmoid



Fonte: Elaborada pelo autor.

Onde a é o seu parâmetro de inclinação. A função sigmoide tem propriedades importantes que a tornam uma das funções de ativação mais utilizada em construções de redes neurais,

como ser estritamente crescente, exibir um balanceamento adequado entre comportamento linear e não linear e ser uma função diferenciável.

Podemos definir a função sigmoid, em python, da seguinte maneira:

```
1 >>> def sigmoid(z):
2 ...     return 1.0/(1.0+np.exp(-z))
3 >>>sigmoid(10)
4 ...0.9999546021312976
```

Conhecendo as principais funções de ativação, devemos escolher a que se adequa melhor ao problema a ser trabalhado, não há uma regra específica para sua escolha, entretanto, dependendo da complexidade do problema:

- A função Sigmoid é mais utilizada em problemas de classificação;
- A função ReLU é uma função de ativação muito usada, em diversos casos, más so deve esta em neurônios de camadas ocultas;
- De modo geral, podemos usar inicialmente a função ReLU, caso não tenha os resultados esperados, testamos outras funções de ativação.

As redes neurais são compostas por nós ou unidades, os neurônios, conectados por ligações direcionadas e tendo duas características básicas:

- **Arquitetura:** É a forma que rede está disposta, isto é, o número de camadas e como se dar a conexão entre os neurônios;
- **Aprendizado:** Trata das regras de ajuste dos pesos sinápticos da rede.

A ligação entre as unidades i e j tem por finalidade propagar a ativação do neurônio a_i com o neurônio h_j, onde cada ligação tem um peso w_{ji} que determina a força e o sinal dessa conexão. Para cada neurônio j é calculado a soma deu suas entradas ponderadas, pelos pesos w_{ji}.

$$s_j = \sum_{i=1}^{m} w_{ji} xi$$

Que é somado ao termo bias, em seguida aplica-se a função de ativação σ para obter a saída.

$$a_j = \sigma(s_j + b_j) = \sigma(\sum_{i=1}^{m} w_{ij} xi + b_j)$$

A forma de conexão dos neurônios é de grande importância para o tipo de aprendizado que se quer trabalhar e para isso existe duas opções relevantes quando se fala em conexão de neurônios entre as camadas da rede:

- **Redes com alimentação para frente :** Também chamada Redes Feedforward (FNN), são redes sem realimentação, isto é, o sinal tem uma única direção percorrendo a rede da camada de entrada (input layer) para a camada de saída (output layer) sem conexão entre neurônios da mesma camada.
- **Redes recorrentes(Com retroalimentação):** Diferentemente de redes Feedforward, as redes recorrentes incluem conexões ponderadas entre neurônios de uma mesma camada ou recebem entradas de camadas posteriores, ou ainda, podem ter sua própria saída como entrada. Essas conexões são chamadas de conexões de retroalimentação ou feedback.

Vamos tratar, neste trabalho, somente redes com alimentação para frente. As redes feedforward são dispostas em camadas, de forma que cada unidade recebe entrada somente de unidades da camada imediatamente anterior. Podemos classificar este tipo de rede neural em dois subtipos. O primeiro são as redes de camada única ou rede perceptron, em que todos os neurônios da camada de entrada se conectam diretamente com os neurônios da camada de saída, o segundo tipo, as redes multicamadas são as que tem uma ou mais camadas ocultas, o que implica que as entradas não são conectadas com as saídas da rede.

Quando uma RNA contém uma profunda pilha de camadas ocultas ela é chamada de *deep neural network(DNN)*. A área de estudo de Deep Learning estuda as DNNs e mais geralmente modelos que contém pilhas profundas de cálculos.

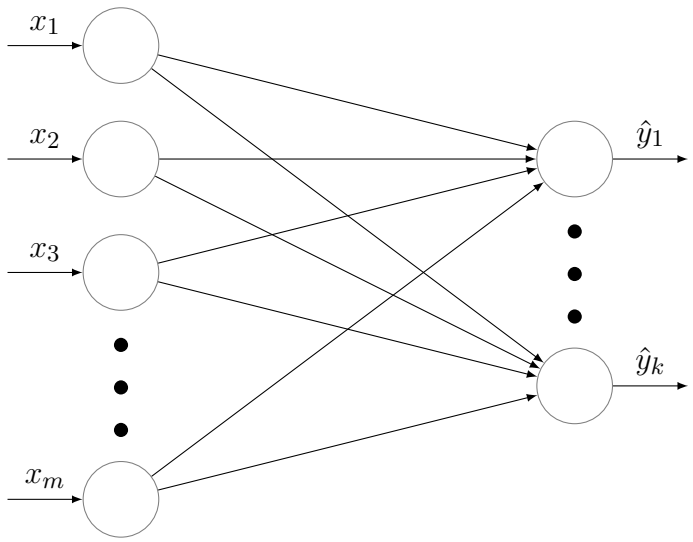
4.0.5 Perceptrons

As redes neurais com uma única camada e com alimentação para frete, chamadas também de rede perceptron, foram as primeiras redes neurais artificiais, desenvolvidas por Rosenblatt

(1958). Essas redes têm como característica principal que todas as entradas são conectadas diretamente com as saídas da rede neural. Os perceptrons construídos com apenas um neurônio na camada de saída são limitados a classificação de problemas binários, isto é, quando temos apenas duas classes. Aumentando a quantidade de neurônios da camada de saída podemos realizar classificações com mais de duas classes, desde que, sejam todas linearmente separáveis, caso contrário o perceptrons não terão resultados satisfatórios.

Exemplo 4.2. Diagrama de uma rede perceptron com m inputs (entradas) e k outputs (saídas).

Figura 27 – Rede perceptron de camada única



Fonte: https://tex.stackexchange.com/questions/423745/draw-a-filled-neural-network-diagram-with-tikz . Acesso: OUT, 2020.

4.1 Treinamento de um perceptron de camada única

Para treinar uma rede preceptron de camada única vamos utilizar 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:

- $\mathbf{x}^{(j)} = \left[x_1^{(j)}, x_2^{(j)}, \ldots, x_m^{(j)}\right]^T$ é um vetor m-dimensional;
- $\mathbf{y}^{(j)} = \left[y_1^{(j)}, y_2^{(j)}, \ldots, y_k^{(j)}\right]^T$ 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 $\mathbf{x}^{(j)} = \left[1, x_1^{(j)}, x_2^{(j)}, \ldots, x_m^{(j)}\right]^T$ e o vetor de pesos é da forma $\mathbf{w}^{(j)} = \left[w_{i0}^{(j)} = b_i, w_{i1}^{(j)}, w_{i2}^{(j)}, \ldots, w_{im}^{(j)}\right]^T$.

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 $\hat{y}_i^{(j)}$, dado em função de w_{ir}, é a predição da rede para a j-ésimo exemplo do conjunto de treinamento, é definido por

$$\hat{y}_i^{(j)} = \sum_{r=0}^{m} w_{ir}^{(j)} x_r^{(j)}. \tag{3}$$

Desse modo a soma $E(n)$ dos erros de todos os neurônios da camada de saída da rede, ao passarmos o j-ésimo exemplo do conjunto de teste é

$$E^{(j)} = \frac{1}{2} \sum_{i=1}^{k} \epsilon_i^{2(j)} \tag{4}$$

$$E^{(j)} = \frac{1}{2} \sum_{i=1}^{k} \left(y_i^{(j)} - \hat{y}_i^{(j)}\right)^2. \tag{5}$$

Onde k é o número de neurônios da camada de saída. Sendo n o número de amostras do conjunto de treinamento podemos definir o erro total E_T da rede após a apresentação de todo o conjunto de treinamento como:

$$E_T = \frac{1}{n} \sum_{j=1}^{n} E^{(j)} \tag{6}$$

Ou ainda,

$$E_T = \frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{k}\left(y_i^{(j)} - \hat{y}_i^{(j)}\right)^2. \tag{7}$$

Onde E_T, a função de custo da rede, é uma função de várias variáveis dadas pelos pesos sinápticos e os níveis de bias introduzidos dentro dos vetores de pesos. A atualização dos pesos ocorre a cada apresentação de um exemplo do conjunto de treinamento a rede, um a um, até a apresentação completa do todo o conjunto de treinamento, que recebe o nome de uma época.

Utilizando a regra da cadeia, podemos derivar a função de custo $E^{(j)}$ com relação aos pesos sináptico $w_{ir}^{(j)}$, assim

$$\frac{\partial E^{(j)}}{\partial w_i^{(j)}} = \frac{\partial E^{(j)}}{\partial \epsilon_i^{(j)}}\frac{\partial \epsilon_i^{(j)}}{\partial \hat{y}_i^{(j)}}\frac{\partial \hat{y}_i^{(j)}}{\partial w_{ir}^{(j)}}. \tag{8}$$

Vamos determinar agora cada uma das derivadas do lado direito da equação (8) Derivando ambos os lados da equação (4) com relação ϵ_i, temos:

$$\begin{aligned}\frac{\partial E^{(j)}}{\partial \epsilon_i^{(j)}} &= \frac{1}{2}\frac{\partial}{\partial \epsilon_i^{(j)}}\left(\sum_{i=1}^{k}(\epsilon_i^{(j)})^2\right)\\ &= \frac{1}{2}\left(\frac{\partial}{\partial \epsilon_i^{(j)}}(\epsilon_1^{(j)})^2 + \frac{\partial}{\partial \epsilon_i^{(j)}}(\epsilon_2^{(j)})^2 + \cdots + \frac{\partial}{\partial \epsilon_i^{(j)}}(\epsilon_i^{(j)})^2 + \cdots + \frac{\partial}{\partial \epsilon_i^{(j)}}(\epsilon_k^{(j)})^2\right)\\ &= \epsilon_i^{(j)}.\end{aligned} \tag{9}$$

Derivando ambos os lados da equação (2) obtemos:

$$\begin{aligned}\frac{\partial \epsilon_i^{(j)}}{\partial \hat{y}_i^{(j)}} &= \frac{\partial}{\partial \hat{y}_i^{(j)}}\left(y_i^{(j)} - \hat{y}_i^{(j)}\right)\\ &= \frac{\partial y_i^{(j)}}{\partial \hat{y}_i^{(j)}} - \frac{\partial \hat{y}_i^{(j)}}{\partial \hat{y}_i^{(j)}}\\ &= -1.\end{aligned} \tag{10}$$

Por fim derivando a equação (3), com relação a w_{ir}, obtemos

$$\begin{aligned}\frac{\partial \hat{y}_i^{(j)}}{\partial w_{ir}^{(j)}} &= \frac{\partial}{\partial w_{ir}^{(j)}}\left(\sum_{r=0}^{m} w_{ir}^{(j)} x_r^{(j)}\right)\\ &= x_r^{(j)}.\end{aligned} \tag{11}$$

Substituindo de (9) a (11) em (8) obtemos:

$$\begin{aligned}\frac{\partial E}{\partial w_{ir}^{(j)}} &= \epsilon_i^{(j)} \cdot (-1) \cdot x_r \\ &= -\epsilon_i^{(j)} x_r^{(j)}.\end{aligned} \tag{12}$$

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

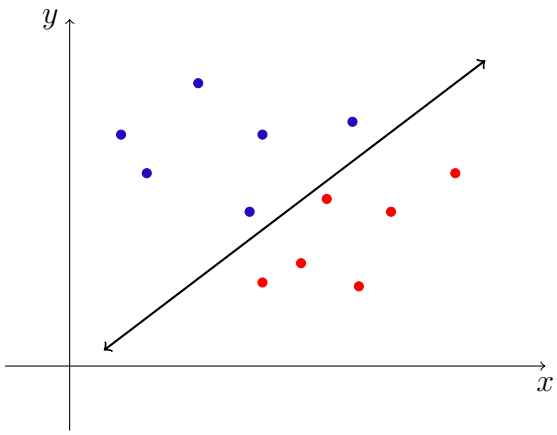
$$\begin{aligned}\Delta w_{ir}^{(j)} &= -\eta \frac{\partial E^{(j)}}{\partial w_{ir}^{(j)}} \\ &= \eta \epsilon_i^{(j)} x_r^{(j)}.\end{aligned} \tag{13}$$

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

$$\begin{aligned}w_{ir}^{(j+1)} &= w_{ir}^{(j)} + \Delta w_{ir}^{(j)} \\ &= w_{ir}^{(j)} + \eta \epsilon_i^{(j)} x_r^{(j)}.\end{aligned} \tag{14}$$

As redes perceptrons, redes de uma única camada, são limitadas e só podem ser utilizadas em classificação de objetos linearmente separáveis. Para entender melhor, considere um conjunto de dados com duas classes que podem ser representados no plano, pois bem, essas classes são linearmente separáveis se existir uma reta que separe os objetos de uma classe dos objetos da outra classe. veja a figura abaixo.

Figura 28 – Classes linearmente separáveis



Fonte: Elaborada pelo autor.

Para resolver problemas não-lineares seja regressão ou classificação é necessário a introdução de camadas ocultas. É consenso que o uso de apenas uma camada oculta é suficiente para resolver a maioria de dos problemas, visto que, redes com apenas uma camada oculta são capazes de aproximar qualquer função não-linear.

4.2 Construção de um perceptron de camada única código à código em Python

Vamos apresentar, agora, os códigos, em Python, de um Perceptron de camada única para classificação binária. Para isso vamos utilizar o conjunto MNIST, que contém 60000 imagens de dígitos manuscritos.

No código abaixo importamos as bibliotecas e o módulos que serão utilizados para construção do perceptron.

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 import pickle #Módulo usado para serializar objetos para que possam ser
      salvos em um arquivo e carregados em um programa mais tarde.
5 import gzip #Módulo com função de compactar e descompactar arquivos.
6 import random #Este módulo implementa geradores de números pseudoaleatórios
      .
```

Os códigos abaixo, carregam o arquivo mnist.pkl.gz, que contém o conjunto MNIST, que está na mesma pasta onde se localiza o arquivo do Jupyter Notebook da rede perceptron, em seguida faz o tratamento dos dados para que estejam prontos para serem inseridos na rede.

```
def load_data():
    f = gzip.open('mnist.pkl.gz', 'rb')# Abre o arquivo compactado, o modo padrão de argumento 'rb'.
    training_data, validation_data, test_data = pickle.load(f, encoding="latin1")# O método pickle.load ler os os dados f.Ao definir a codificação para latin1 permite importar os dados diretamente.
    f.close()#fecha o arquivo f.
    return (training_data, validation_data, test_data)

#Importa o conjunto de dados carregado na função load_data
training_data, validation_data, test_data = load_data()

#Converte os array de rótulos de modo que se o rótulo é 5 modifica para 1, caso contrário 0.
Y = (training_data[1] == 5).astype(np.int)
y_valid_binary = (validation_data[1] == 5).astype(np.int)
y_test_binary = (test_data[1] == 5).astype(np.int)

#Renomeia e separa os conjuntos de dados dos respectivos rótulos.
x_train_binary = training_data[0]
x_valid_binary = validation_data[0]
x_test_binary = test_data[0]

#Converte os array que representam as imagens de 28x28 pixels para um uma arrey de uma linha de 784 pixels
Z = [np.reshape(x, (784)) for x in x_train_binary]
validation_inputs = [np.reshape(x, (784)) for x in x_valid_binary]
test_inputs = [np.reshape(x, (784)) for x in x_test_binary]

X = np.array(Z) #Cria um array de array de 784 pixel
y = Y #Renomeia o conjunto que contém os rótulos  do conjunto de treinamento.
```

Podemos verificar, com o código abaixo, se a forma dos dados de entrada se está de

acordo com a camada de entrada rede que terá 784 neurônios.

```
#Mostra a forma dos dados de treino.
X.shape
(50000, 784)
```

Abaixo, construímos a rede perceptron através do construtor de objetos *class*.

```
class Perceptron(object):

    def __init__(self, sizes):#O método(função) __init__ é chamado de construtor e é utilizado para armazenar variáveis globais que serão utilizados por outros métodos. O par metro sizes é um objeto do tipo lista que determina o número de neurônio das camadas da rede.
        self.sizes = sizes
        self.biases = np.zeros(1)#Define os bias iniciais
        self.weights = np.zeros(784) #Define os pesos iniciais
        self.epocas = epocas
        self.rate = rate

    def fit(self, X, y):#Treina a rede.

        self.errors = []  #Cria um conjunto vazio, onde será armazenado o número de classificações erradas.

        for i in range(self.epocas):#Note que neste código x += n -> x = x + n.
            err = 0
            for xi, target in zip(X, y):
                delta_w = self.rate * (target - self.predict(xi))#\Delta wri(j) =\eta \episilon x_{i}
                self.weights += delta_w * xi #w_{j+1} = w_{j} + \Delta w x_{i}
                self.biases += delta_w # b_{j+1}=b_{j} +  \Delta w
                err += int(delta_w != 0.0) #!= Diferente
            self.errors.append(err)#O método .append insere os erros no conjunto self.erros.
        return self

```

```
24    def net_input(self, X):#Calcula o campo de ativação da rede dado o
      conjunto de entrada X.
25
26        return np.dot(X, self.weights) + self.biases
27
28    def predict(self, X): #Aplica o campo de ativação na função de ativação
      degrau, de modo que se a saída for 1 temos um 5, por outro lado, se a sa
      ída for 0 tem que a rede prever que não é um 5.
29
30        return np.where(self.net_input(X) >= 0.0, 1, 0)
```

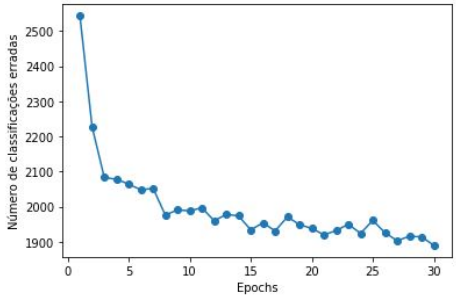
Fonte: <http://www.bogotobogo.com/python/scikit-learn/Perceptron_Model_with_Iris_DataSet.php>.

Nesse ponto, com a rede já construída, podemos treiná-la, para isso, antes, devemos definir a taxa de aprendizado e o número de épocas de treinamento.

```
1 rate = 0.001 #Taxa de aprendizado
2 epocas = 30
3 rede = Perceptron([784, 1])#sizes=[784, 1] define que a camada de entrada
      tem 784 neurônio e camada de saída tem apenas 1 neurônio.
4 rede.fit(X, y)
5 rede.fit(X, y)#Treina a rede
```

Podemos verificar se o treinamento foi satisfatório com o código abaixo.

```
1 plt.plot(range(1, len(rede.errors) + 1), rede.errors, marker='o')
2 plt.xlabel('Epochs')
3 plt.ylabel('Número de classificações erradas')
4 plt.show()
```




```
5
```

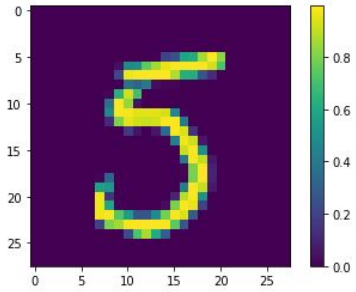
Vamos testar o desempenho do perceptron com o conjunto de testes, para isso, vamos definir a função *evaluate*, que vai contar o número de previsões corretas do preceptron.

```
def evaluate(test_inputs, y_test_binary):#O método evaluate retorna o nú
    mero de entradas de teste para as quais a rede neural produz o resultado
     correto. A saída da rede, neste caso, é o índice do neurônio da camada
    de saída com a maior ativação.

        test_results = [(rede.predict(x), y) for x, y in zip(test_inputs,
    y_test_binary)]#O código np.argmax calcula o índice com maior argumento(
    ativação) da saída gerada pelo método feedforward para o sinal de
    entrada x.
        return sum(int(x == y) for x, y in test_results)# Retorna o número
    de tuplas em test_results onde x=y, isto é, que a previsão foi correta.
t = evaluate(test_inputs, y_test_binary)#Executa a função no conjunto de
    testes
t
9620
len(test_inputs)#Número de elementos do conjunto test_inputs
```

Note que o perceptron acertou 9620 imagens de um total de 10000, isso representa uma taxa de precisão de 96,2%.

Podemos fazer a previsão de uma única imagem do conjunto de testes e visualizar se a previsão foi correta, veja a sequência de códigos abaixo.

```
rede.predict(test_inputs[509])
array([1])

plt.figure()
plt.imshow(test_inputs[509].reshape(28,28))#Usamos o método .reshape para
    modificar a imagem para o formato original para que seja possível a
    visualização.
plt.colorbar()
plt.grid(False)
plt.show()
```



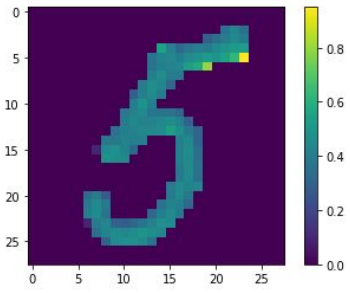
```
9
```

Vamos testar o Perceptron com uma imagem que não pertence ao conjunto de dados MNIST, nessa parte, podemos escrever o dígito em uma folha escanear para fazer o teste, é importante destacar que é necessário retirar o fundo da imagem escaneada para que a imagem não tenha ruídos.

```
1 from PIL import Image#PIL é a biblioteca para trabalhar com imagens no
      Python. O modulo Image do PIL traz varia funções úteis para imagens.
2 from matplotlib import image #O módulo image do matplotlib é usado para
      operações básicas de carregamento, redimensionamento e exibição de
      imagens
3
4 image = Image.open('img5.png')#Carrega a imagem img5(sem fundo) presente na
       mesma pasta do arquivo Jupyter Notebook da rede neural.
5
6 img_resize = image.resize((28,28))#redimensiona a imagem para 28x28.
7
8 image_cinza = img_resize.convert(mode="L")#Converte a imagem para escale de
       cinza.
9
10 from numpy import asarray #O asarray é uma função NumPy usada para
      converter as entradas de um array.
11
12 data = asarray(image_cinza)#Converte a image_cinza em um array.
13
14 data1 = data / 100 #A divisão ocorre para que os valores que representam os
       pixels variem de 0 a 1.
15
16 img = np.reshape(data1, (784))#Redimensiona a imagem para ficar de acordo
      com a camada de entrada da rede.
```

Agora, a imagem está de acordo com a camada de entrada do Perceptron, podemos fazer a previsão.

```
1 rede.predict(img)
2 array([1])
3
4 plt.figure()
5 plt.imshow(data1)
6 plt.colorbar()
7 plt.grid(False)
8 plt.show()
```

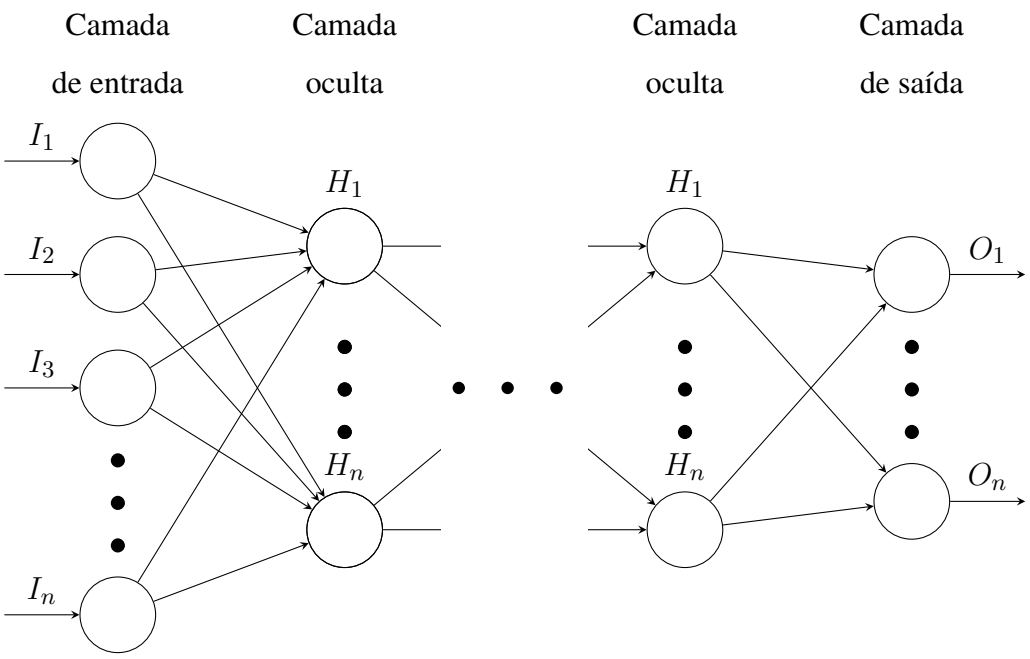



```
9
```

4.3 Redes multicamada com alimentação para frente

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 29 – Redes multicamada com alimentação para frente



Fonte: https://tex.stackexchange.com/questions/343462/n-level-deep-networks-using-tikz. Acesso: OUT, 2020.

Uma rede perceptron de múltiplas camadas tem as seguintes características:

- Uma função de ativação não-linear e diferenciável;
- Uma ou mais camadas ocultas, responsáveis pela realização de tarefas complexas como extração de padrões significativos dos dados de entrada;
- Alto grau de conectividade.

É muito importante, ao construir uma rede neural artificial, a decisão quanto ao número de camadas a serem utilizadas. Como foi dito na seção anterior a utilização de apenas uma camada oculta já capacita a rede para resolver uma grande parte dos problemas não-lineares. O uso de poucas camadas ocultas pode levar a rede ao underfitting (sub-ajuste), isto é, não funcionará bem nem com os dados de treino nem com os dados de testes. Por outro lado, se rede tiver muitas camadas ocultas ocorrerá overfitting (sobreajute), isto é, a rede vai se ajustar aos dados de treinamento e não fara boas generalizações. Desse modo, devemos sempre escolher o menor número de camadas que soluciona o problema, isso deve ser obtido testando e modificando os paramentos da rede durante o treinamento.

Podemos pensar as redes neurais como uma função $h_{w,b}(x)$ parametrizadas pelos pesos w_{ji} e pelos bias b_j, desse modo podemos expressar a saída como uma função real de várias variáveis dadas pelos pesos e pelos bias. A partir daí temos que determinar os pesos e o bias de modo que o erro seja mínimo e para isso podemos utilizar derivadas parciais e o método de gradiente descendente.

Cada neurônio da camada de saída está associado a uma das classes a serem preditas. Desse modo, podemos representar o valor de saída, para uma entrada específica, pelo vetor $\hat{y} = [\hat{y_1}, \hat{y_2}, \ldots, \hat{y_m}]^T$ onde m é o número de neurônios da camada de saída e também o número de classes do conjunto de dados, desse modo, a componente com maior ativação é a classe predita pela rede. Os rótulos do problema podem também ser representados por vetores m-dimensionais, onde a componente da classe correta vale 1 e as demais componentes vale 0.

4.4 Treinamento de redes neurais multicamada

Nesta seção, vamos apresentar a matemática por trás do treinamento de uma rede neural profunda, utilizando o algoritmo backpropagation que fornece um método computacional eficiente para o treinamento de redes neurais multicamadas. O algoritmo é baseado no gradiente descendente e, para utilizá-lo, é preciso que a função de custo seja contínua e diferenciável, para garantir esses requisitos utilizaremos uma função de ativação contínua e diferenciável.

Até 1986, muitos pesquisadores tentaram encontrar uma maneira de treinar MLPs (Multilayer perceptron), sem sucesso. Até que David Rumelhart, Geoffrey Hinton e Ronald Williams publicaram um artigo que introduziu o algoritmo de treinamento ***backpropagation*** usado até hoje.

O Algoritmo backpropagation é composto por duas fases que interagem:

- ***forward*:** É a fase para frente, também chamada de propagação, onde o conjunto de treinamento é apresentado a rede, passando pela primeira camada, sendo ponderado pelos respectivos pesos e aplicado a função de ativação em cada neurônio produzindo um valor de saída, que se tornam as entradas dos neurônios da camada seguinte e esse processo ocorre camada a camada até a camada de saída produzir o valor de saída para a rede, que é comparado com os respectivos rótulos, que são os valores desejáveis, através de uma função de custo que mede o erro cometido pela rede ao fazer a previsão.

- ***backward*** É a fase para trás, também chamado de retropropagação, que utiliza o erro encontrado na fase forward para ajustar os pesos e os bias. O ajuste é realizado da camada de saída para a camada de entrada. Os pesos e bias são ajustados para que a resposta da rede se aproxime da resposta correta (rótulo).

Considere o conjunto T de treinamento com k classes e n amostras

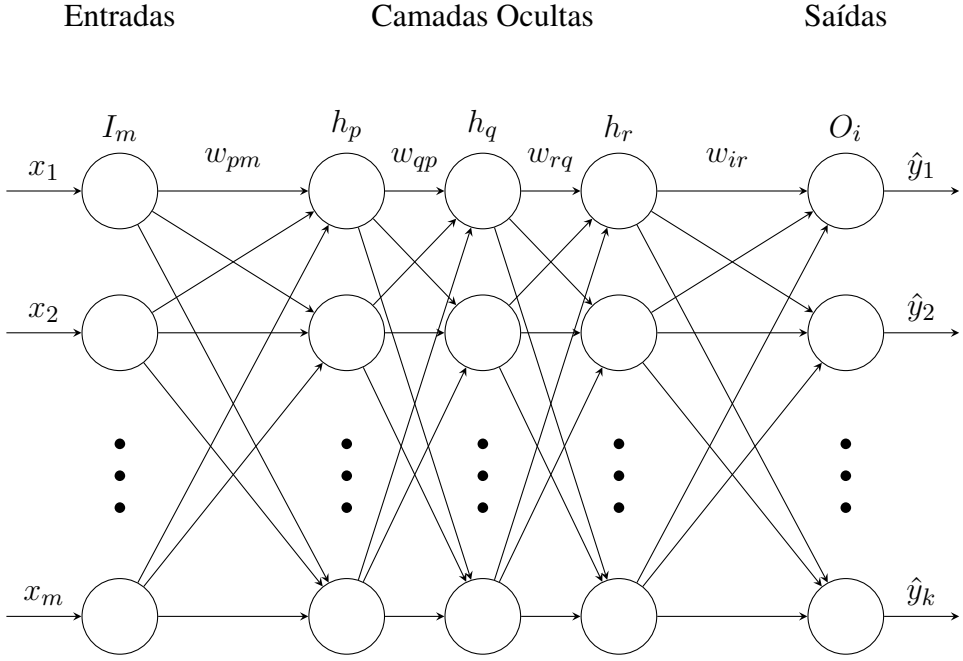
$$T = \left\{ \left(\mathbf{x}^{(1)}, \mathbf{y}^{(1)} \right), \left(\mathbf{x}^{(2)}, \mathbf{y}^{(2)} \right), \ldots, \left(\mathbf{x}^{(n)}, \mathbf{y}^{(n)} \right) \right\} \tag{15}$$

Onde:

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

Vamos construir e treinar uma rede com a camada de entrada *(input layer)*, com m entradas, três camadas ocultas (hidden layer) e a camada de saída com k neurônios.

Figura 30 – Rede neural artificial



Fonte: Elaborada pelo autor.

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 h_r e O_i respectivamente e os termos de viés, 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 neurônio 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 de todos os neurônios da rede. Assim, por exemplo, para o p-ésimo neurônio da camada h_p, um vetor de entrada é da forma $\mathbf{x}^{(j)} = \left[1, x_1^{(j)}, x_2^{(j)}, \ldots, x_m^{(j)}\right]^T$ e o vetor de pesos é da forma $\mathbf{w}^{(j)} = \left[w_{p0}^{(j)} = b_p, w_{p1}^{(j)}, w_{p2}^{(j)}, \ldots, w_{pm}^{(j)}\right]^T$.

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

$$\epsilon_i^{(j)} = y_i^{(j)} - \hat{y}_i^{(j)}. \tag{16}$$

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

$$E^{(j)} = \frac{1}{2}\sum_{i=1}^{k} {\epsilon_i^{(j)}}^2 \tag{17}$$

$$E^{(j)} = \frac{1}{2}\sum_{i=1}^{k} \left(y_i^{(j)} - \hat{y}_i^{(j)}\right)^2. \tag{18}$$

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

$$E_T = \frac{1}{n}\sum_{j=1}^{n} E^{(j)}, \tag{19}$$

ou ainda,

$$E_T = \frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{k}\left(y_i^{(j)} - \hat{y}_i^{(j)}\right)^2, \tag{20}$$

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 a rede, um a um, até a apresentação completa de todo o conjunto de treinamento que recebe o nome de época. Por simplificação de notação, omitiremos o índice (j) que indica a apresentação do j-ésimo exemplo do conjunto de treinamento a rede, por exemplo, utilizaremos $\hat{y}_i$ em vez de $\hat{y}_i^{(j)}$. Deixado claro que, todos os cálculos abaixo se referem a apresentação do j-ésimo exemplo do conjunto de treinamento, note que

$$\hat{y}_i = \xi\left(u_{O_i}\left(w_{ir}\right)\right), \tag{21}$$

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

$$u_{O_i}(w_{ir}) = \sum_{r=0}^{R} w_{ir}h_r. \tag{22}$$

Substituindo (21) em (18) temos

$$E = \sum_{i=1}^{k}\left(y_i - \xi\left(u_{O_i}\left(w_{ir}\right)\right)\right)^2. \tag{23}$$

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

$$\frac{\partial E}{\partial w_{ir}} = \frac{\partial E}{\partial \epsilon_i}\frac{\partial \epsilon_i}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial u_{O_i}}\frac{\partial u_{O_i}}{\partial w_{ir}}. \tag{24}$$

Vamos determinar agora cada uma das derivadas do lado direito da equação (24) Deri-

vando ambos os lados da equação (17) com relação ϵ_i, temos que

$$\begin{aligned}\frac{\partial E}{\partial \epsilon_i} &= \frac{\partial}{\partial \epsilon_i}\left(\sum_{i=1}^{k}(\epsilon_i)^2\right) \\ &= \frac{\partial}{\partial \epsilon_i}(\epsilon_1)^2 + \frac{\partial}{\partial \epsilon_i}(\epsilon_2)^2 + \cdots + \frac{\partial}{\partial \epsilon_i}(\epsilon_i)^2 + \cdots + \frac{\partial}{\partial \epsilon_i}(\epsilon_k)^2 \\ &= 2\epsilon_i.\end{aligned} \tag{25}$$

Derivando ambos os lados da equação (16) obtemos:

$$\begin{aligned}\frac{\partial \epsilon_i}{\partial \hat{y}_i} &= \frac{\partial}{\partial \hat{y}_i}(y_i - \hat{y}_i) \\ &= \frac{\partial y_i}{\partial \hat{y}_i} - \frac{\partial \hat{y}_i}{\partial \hat{y}_i} \\ &= -1.\end{aligned} \tag{26}$$

Derivando ambos os lados da equação (21), com relação a, obtemos:

$$\begin{aligned}\frac{\partial \hat{y}_i}{\partial u_{O_i}} &= \frac{\partial}{\partial u_{O_i}}\left(\xi\left(u_{O_i}\right)\right) \\ &= \xi'\left(u_{O_i}\right).\end{aligned} \tag{27}$$

Por fim derivando a equação (22), com relação a w_{ir}, obtemos

$$\begin{aligned}\frac{\partial u_{O_i}}{\partial w_{ir}} &= \frac{\partial}{\partial w_{ir}}\left(\sum_{r=0}^{R} w_{ir} h_r\right) \\ &= h_r.\end{aligned} \tag{28}$$

Substituindo de (25) a (28) em (24) obtemos:

$$\begin{aligned}\frac{\partial E}{\partial w_{ir}} &= 2\epsilon_i \cdot (-1) \cdot \xi'\left(u_{O_i}\right) h_r \\ &= -2\epsilon_i \cdot \xi'\left(u_{O_i}\right) h_r.\end{aligned} \tag{29}$$

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

$$\begin{aligned}\Delta w_{ir} &= -\eta \frac{\partial E}{\partial w_{ir}} \\ &= -\eta \frac{\partial E}{\partial \epsilon_i} \frac{\partial \epsilon_i}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial u_{O_i}} \frac{\partial u_{O_i}}{\partial w_{ir}} \\ &= 2\eta \epsilon_i \cdot \xi'(u_{O_i})\, h_r.\end{aligned} \tag{30}$$

Onde η é a taxa de aprendizado da rede. Observe que o sinal negativo na equação (30) é usado para inverter a direção do gradiente que que aponta para 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 (30) da seguinte forma:

$$\Delta w_{ir} = 2\eta \delta_i h_r, \tag{31}$$

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

$$\begin{aligned}\delta_i &= \frac{\partial E}{\partial u_{O_i}} \\ &= \epsilon_i \cdot \xi'(u_{O_i}).\end{aligned} \tag{32}$$

Para a correção do valor do peso w_{ir} referente ao J-ésimo exemplo do conjunto de treinamento para o $(j+1)$-ésimo utilizaremos a seguinte equação

$$\begin{aligned}w_{ir}(j+1) &= w_{ir}(j) + \Delta w_{ir}(j) \\ &= w_{ir}(j) + 2\eta \delta_i h_r.\end{aligned} \tag{33}$$

Vamos derivar a função de custo E com relação aos pesos w_{rq}. Note que para isso devemos escrever E em função de w_{rq} que é um peso pertencente a uma camada oculta e, diferente do caso anterior, a atualização deste peso depende do erro de todos os neurônios que estão a sua direita, isto é, todos os neurônios da camada de saída. Desse modo, Observe que:

$$u_{h_r} = \sum_{q=0}^{Q} w_{rq} h_q \tag{34}$$

daí,

$$h_r = \zeta\left(u_{h_r}\left(w_{rq}\right)\right). \tag{35}$$

Substituindo (35) em (21) obtemos:

$$\hat{y}_i = \xi\left(u_{O_i}\left(\zeta\left(u_{h_r}\left(w_{rq}\right)\right)\right)\right). \tag{36}$$

Substituindo (36) em (23):

$$E = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - \xi\left(u_{O_i}\left(\zeta\left(u_{h_r}\left(w_{rq}\right)\right)\right)\right)\right)^2. \tag{37}$$

Derivando E em relação a w_{rq} obtemos:

$$\begin{aligned}\frac{\partial E}{\partial w_{rq}} &= \frac{\partial}{\partial w_{rq}}\left(\frac{1}{2}\sum_{i=1}^{n}\left(y_i - \xi\left(u_{O_i}\left(\zeta\left(u_{h_r}\left(w_{rq}\right)\right)\right)\right)\right)^2\right) \\ &= \sum_{i=1}^{n}\epsilon_i\frac{\partial \epsilon_i}{\partial w_{rq}} \\ &= \sum_{i=1}^{n}\epsilon_i\frac{\partial \epsilon_i}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial u_{O_i}}\frac{\partial u_{O_i}}{\partial h_r}\frac{\partial h_r}{\partial u_{h_r}}\frac{\partial u_{h_r}}{\partial w_{rq}}.\end{aligned} \tag{38}$$

Observe que já calculamos algumas destas derivadas vamos calcular as que faltam. Derivando a equação (22) em relação a h_r temos:

$$\begin{aligned}\frac{\partial u_{O_i}}{\partial w_{ir}} &= \frac{\partial}{\partial w_{ir}}\left(\sum_{r=0}^{R} w_{ir}h_r\right) \\ &= w_{ir}.\end{aligned} \tag{39}$$

A derivada parcial de h_r, com relação ao campo de ativação u_{h_r}, é obtida diferenciando ambos os lados da equação (35):

$$\frac{\partial h_r}{\partial u_{h_r}} = \zeta'\left(u_{h_r}\right). \tag{40}$$

Para determinar $\frac{\partial u_{h_r}}{\partial w_{rq}}$ devemos derivar a equação (34), desse modo:

$$\begin{aligned}\frac{\partial u_{h_r}}{\partial w_{rq}} &= \frac{\partial}{\partial w_{rq}}\left(\sum_{q=0}^{Q} w_{rq}h_q\right)\\ &= h_q.\end{aligned} \tag{41}$$

Substituindo de (25) a (27) e de (39) a (41) em (38) obtemos:

$$\begin{aligned}\frac{\partial E}{\partial w_{rq}} &= \sum_{i=1}^{n} \epsilon_i \cdot (-1) \cdot \xi'\left(u_{O_i}\right) w_{ir} \zeta'\left(u_{h_r}\right) h_q\\ &= -\sum_{i=1}^{n} \delta_i w_{ir} \zeta'\left(u_{h_r}\right) h_q\\ &= -\zeta'\left(u_{h_r}\right) h_q \sum_{i=1}^{n} \delta_i w_{ir}.\end{aligned} \tag{42}$$

Desse modo, a variação do peso w_{rq} é definida como

$$\begin{aligned}\Delta w_{rq} &= -\eta \frac{\partial E}{\partial w_{rq}}\\ &= \eta \zeta'\left(u_{h_r}\right) h_q \sum_{i=1}^{n} \delta_i w_{ir}\\ &= \eta \delta_r h_q,\end{aligned} \tag{43}$$

onde $\delta_r = \zeta'\left(u_{h_r}\right) \sum_{i=1}^{n} \delta_i w_{ir}$. Assim a correção dos pesos w_{rq} com referência ao j-ésimo exemplo do conjunto de treinamento é feita de acordo com a equação abaixo.

$$\begin{aligned}w_{rq}(j+1) &= w_{rq}(j) + \Delta w_{rq}(j)\\ &= w_{rq}(j) + \eta \delta_r h_q.\end{aligned} \tag{44}$$

O próximo passo é derivar o erro com relação aos pesos w_{qp}, de modo análogo aos casos anteriores devemos escrever a função do custo E em função dos pesos w_{qp}

Assim temos que:

$$u_{h_q}\left(w_{qp}\right) = \sum_{p=0}^{P} w_{qp} h_p, \tag{45}$$

daí segue que:

$$h_q = \phi\left(u_{h_q}(w_{qp})\right). \tag{46}$$

Substituindo (46) em (36)

$$\hat{y}_i = \xi\left(u_{O_i}\left(\zeta\left(u_{h_r}\left(\phi\left(u_{h_q}(w_{qp})\right)\right)\right)\right)\right). \tag{47}$$

Substituindo (47) em (18):

$$E = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - \xi\left(u_{O_i}\left(\zeta\left(u_{h_r}\left(\phi\left(u_{h_q}(w_{qp})\right)\right)\right)\right)\right)\right)^2. \tag{48}$$

Derivando E em relação a w_{qp} obtemos:

$$\begin{aligned}\frac{\partial E}{\partial w_{qp}} &= \frac{\partial}{\partial w_{qp}}\left(\frac{1}{2}\sum_{i=1}^{n}\left(y_i - \xi\left(u_{O_i}\left(\zeta\left(u_{h_r}\left(\phi\left(u_{h_q}(w_{qp})\right)\right)\right)\right)\right)\right)^2\right)\\ &= \sum_{i=1}^{n}\epsilon_i\frac{\partial \epsilon_i}{\partial w_{qp}}\\ &= \sum_{i=1}^{n}\epsilon_i\frac{\partial \epsilon_i}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial w_{qp}}.\end{aligned} \tag{49}$$

Vamos determinar $\frac{\partial \hat{y}_i}{\partial w_{qp}}$. Para isso vamos derivar a equação (47) com relação a w_{qp}

$$\begin{aligned}\frac{\partial \hat{y}_i}{\partial w_{qp}} &= \xi'\left(u_{O_i}\right)\frac{\partial u_{O_i}}{\partial w_{qp}}\\ &= \xi'\left(u_{O_i}\right)\sum_{r=0}^{R} w_{ir}\zeta'\left(u_{h_r}\right)\frac{\partial u_{h_r}}{\partial w_{qp}}\\ &= \xi'\left(u_{O_i}\right)\sum_{r=0}^{R} w_{ir}\zeta'\left(u_{h_r}\right)w_{rq}\phi'\left(u_{h_q}\right)h_p.\end{aligned} \tag{50}$$

A derivada parcial de h_q com relação ao campo de ativação u_{h_q} é obtida diferenciando ambos os lados da equação (46) obtemos:

$$\frac{\partial h_q}{\partial u_{h_q}} = \phi'\left(u_{h_q}\right). \tag{51}$$

Para determinar $\frac{\partial u_{h_q}}{\partial w_{qp}}$ devemos derivar a equação (45), desse modo:

$$\begin{aligned}\frac{\partial u_{h_q}}{\partial w_{qp}} &= \frac{\partial}{\partial w_{qp}}\left(\sum_{p=0}^{P} w_{qp}h_p\right)\\ &= h_p.\end{aligned} \tag{52}$$

Substituindo de (25) a (27), de (39) a (41) e de (50) a (52) em (49) obtemos:

$$\begin{aligned}\frac{\partial E}{\partial w_{qp}} &= \sum_{i=1}^{n}\epsilon_i\cdot(-1)\cdot\xi'(u_{O_i})\sum_{r=0}^{R} w_{ir}\zeta'(u_{h_r})\,w_{rq}\phi'\left(u_{h_q}\right)h_p\\ &= -\phi'\left(u_{h_q}\right)h_p\sum_{r=0}^{R}\zeta'(u_{h_r})\sum_{i=1}^{n}\delta_i w_{ir}w_{rq}\\ &= -\phi'\left(u_{h_q}\right)h_p\sum_{r=0}^{R}\delta_r w_{rq}.\end{aligned} \tag{53}$$

Desse modo, a variação do peso w_{qp} é definida como

$$\begin{aligned}\Delta w_{qp} &= -\eta\frac{\partial E}{\partial w_{qp}}\\ &= \eta\phi'\left(u_{h_q}\right)h_p\sum_{r=0}^{R}\delta_r w_{rq}\\ &= \eta\delta_q h_p.\end{aligned} \tag{54}$$

Onde $\delta_q = \phi'\left(u_{h_q}\right)\sum_{r=0}^{R}\delta_r w_{rq}$. Assim a correção dos pesos w_{qp} com referência ao j-ésimo exemplo do conjunto de treinamento é feita de acordo com a equação abaixo.

$$\begin{aligned}w_{qp}(j+1) &= w_{qp}(j)+\Delta w_{qp}(j)\\ &= w_{qp}(j)+\eta\delta_p h_p.\end{aligned} \tag{55}$$

Por fim, devemos derivar a função de custo E em relação aos pesos w_{pm}. Escrevendo o erro E em função de w_{pm}, para isso, note que

$$u_{h_p} = \sigma\left(\sum_{m=0}^{M} w_{pm}I_m\right), \tag{56}$$

daí segue que

$$h_q = \sigma\left(u_{h_p}(w_{pm})\right). \tag{57}$$

Substituindo (57) em (47)

$$\hat{y}_i = \xi\left(u_{O_i}\left(\zeta\left(u_{h_r}\left(\phi\left(u_{h_q}\left(\sigma\left(u_{h_p}(w_{pm})\right)\right)\right)\right)\right)\right)\right). \tag{58}$$

Substituindo (58) em (18)

$$E = \frac{1}{n}\sum_{i=1}^{n}\left(\xi\left(u_{O_i}\left(\zeta\left(u_{h_r}\left(\phi\left(u_{h_q}\left(\sigma\left(u_{h_p}(w_{pm})\right)\right)\right)\right)\right)\right)\right) - y_i\right)^2, \tag{59}$$

Derivando E parcialmente com relação aos pesos w_{pm} de modo análogo aos casos anteriores obtemos

$$\frac{\partial E}{\partial w_{pm}} = -\sigma'\left(u_{h_p}\right) I_m \sum_{q=0}^{Q} \delta_q w_{qp}. \tag{60}$$

Segue que a variação do peso w_{pm} é dada por

$$\begin{aligned} \Delta w_{pm} &= -\eta \frac{\partial E}{\partial w_{pm}} \\ &= \eta \sigma'\left(u_{h_p}\right) I_m \sum_{q=0}^{Q} \delta_q w_{qp} \\ &= \eta \delta_p I_m. \end{aligned} \tag{61}$$

Onde $\delta_p = \phi'\sigma'\left(u_{h_p}\right)\sum_{q=0}^{Q}\delta_q w_{qp}$. Assim a correção dos pesos w_{pm} com referência ao j-ésimo exemplo do conjunto de treinamento é feita de acordo com a equação abaixo.

$$\begin{aligned} w_{pm}(j+1) &= w_{pm}(j) + \Delta w_{pm}(j) \\ &= w_{pm}(j) + \eta \delta_p I_m. \end{aligned} \tag{62}$$

Podemos generalizar esse treinamento para redes neurais com um número qualquer de camadas ocultas. Desse modo, o algoritmo Backpropagation poderá ser sintetizado da forma abaixo.

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

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

onde $y_i^{l-1}(n)$ é o sinal de saída do neurônio i da camada $l-1$, na apresentação do $n-\acute{e}simo$ 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

$$y_j^l = \sigma(v_j^l(n)).$$

Note que $y_j^0(n) = x_j$, isto é, a $j-\acute{e}sima$ componente do vetor de entrada x(n). Se o neurônio j pertence a 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 $j-\acute{e}sima$ componente do vetor de rótulos.

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

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

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 esta equação, visto que, $w_{j0}^l = b_j^l$.

4.5 Construção de uma rede neural código à código em Python

Vamos construir uma rede neural para identificar dígitos manuscritos e utilizaremos o conjunto de dados MNIST para seu treinamento. O MNIST é formado por imagens digitalizadas de dígitos manuscritos de 28x28 pixels, em escala de cinza onde cada pixel vai ser representado por uma escala de 0 a 1, de modo que 0 representa o branco e 1 representa o preto e os valores intermediários representam tons de cinza que se escurecem a medida que se aproxima de 1. As imagens não tem ruídos, isto é, não tem o fundo. A imagem será convertida para uma sequência 28x28=784 pixels, isto é, os pixels serão alinhados, desse modo, as entradas da serão representadas por vetores de 784 dimensões onde cada uma de suas componentes representa um pixel da imagem. Já a saída desejada pode ser representada como uma função vetorial $y = f(x)$ onde x é a entrada da rede, por exemplo, se x representa um 4 temos que $y(x) = (0,0,0,0,1,0,0,0,0,0)^T$.

O conjunto MNIST poderá ser baixado no endereço https://github.com/mnielsen/neural-networks-and-deep-learning/archive/master.zip

Em python, a construção da rede neural será em forma de uma classe que chamaremos de *RedeNeural*.

```
1 >>># Imports
2 >>>import random
3 >>>import numpy as np
4
5 >>> class RedeNeural(object):
6         def __init__(self, sizes): #sizes é um objeto do tipo lista que
    contém o número de neurônio nas respectivas camadas
7             self.num_layers = len(sizes)
8             self.sizes = sizes
9             self.biases =[np.random.randn(y, 1) for y in sizes[1:]]#Gera os
     bias aleatoriamente usando a função Numpy 'np.random.randn' para gerar
    distribuições gaussianas com 0 de média e 1 de desvio padrão.
10            self.weights = [np.random.randn(y, x) for x, y in zip(sizes
    [:-1], sizes[1:])]#Gera os pesos aleatoriamente usando a função Numpy '
    np.random.randn' para gerar distribuições gaussianas com 0 de média e 1
    de desvio padrão.
11        def feedforward(self, a): #adiciona o método feedforward a classe
    Network que dada a entrada para rede retorna a saída correspondente,
```

```
    isto é, aplica a equação a'=\phi(wa+b).
12          for b, w in zip(self.biases, self.weights):
13              a = sigmoid(np.dot(w, a)+b)# A função np.dot  calcula o
    produto escalar de w e a.
14          return a
15
16      def SGD(self, training_data, epochs, mini_batch_size, eta,
    teste_data= None):
17      training_data = list(training_data)
18          n = len(trainig_data)
19
20          if test_data:
21              teste_data = list(test_data)# Cria uma lista com os
    elementos do conjunto de treinamento. O argumento test_data é opcional,
    se for fornecido o programa avaliará a rede após cada período de
    treinamento e mostrará o progresso parcial. Isto é útil para rastrear o
    progresso, mas deixa o aprendizado mais lento.
22              n_test = len(test_data)
23
24          for j in range(epochs):
25              random.shuffle(training_data)# A função random.shuffle
    randomiza os elementos do conjunto de treinamento.
26              mini_batches = [training_data[k:k+mini_batch_size] for k in
     range(0, n, mini_batch_size)] #divide o conjunto de treinamento em
    lotes menores de tamanho conveniente para que o treinamento seja mais rá
    pido e eficiente.
27
28              for mini_batch in mini_batches:
29                  self.update_mini_batch(mini_batch, eta)#eta é a taxa de
     aprendizagem, atualiza os pesos e os bias com uma única iteração da
    decida de gradiente.
30
31          if test_data:# Se o 'test_data' for acrescentado no método SGD
    vai printar o progresso do aprendizado.
32              print("Epoch {} : {} / {}".format(j,self.evaluate(test_data
    ),n_test));
33          else:#Caso não seja acrescentado o 'test_data' ao método SGD o
    progresso do treinamento não será mostrado, somente a finalização.
34              print("Epoch {} finalizada".format(j))
```

```
35
36      def update_mini_batch(self, mini_batch, eta):#Utiliza o mini_batch para
        atualizar os pesos e bias, utilizando a decida do gradiente. O
        mini_batch é uma lista de tuplas (x, y), into é, um subconjunto do
        conjunto de treinamento, eta é a taxa de aprendizado da rede.
37
38
39          nabla_b = [np.zeros(b.shape) for b in self.biases]#Cria um lista de
            arrays com entradas todas 0, com a mesma forma dos array de bias.
40          nabla_w = [np.zeros(w.shape) for w in self.weights]#Cria um lista
        de arrays com entradas todas 0, com a mesma forma dos array de pesos.
41
42          for x, y in mini_batch:
43              delta_nabla_b, delta_nabla_w = self.backprop(x, y)
44              nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]#
        nabla_b=nb+dnb, nb em nabla_b e dnb em delta_nabla_b
45              nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]#
        nabla_w=nw+dnw, nw em nabla_w e dnw em delta_nabla_w
46
47          self.weights = [w-(eta/len(mini_batch))*nw for w, nw in zip(self.
        weights, nabla_w)]
48          self.biases = [b-(eta/len(mini_batch))*nb for b, nb in zip(self.
        biases, nabla_b)]
49      def backprop(self, x, y):#O método backprop retorna a (nabla_b, nabla_w
        ) representando o gradiente para a função de custo C_x. nabla_b e
        nabla_w são listas de camadas de matrizes numpy, semelhantes a ‘self.
        biases‘ e ‘self.weights‘."""
50
51          nabla_b = [np.zeros(b.shape) for b in self.biases]
52          nabla_w = [np.zeros(w.shape) for w in self.weights]
53
54
55          activation = x #Define as ativações feitas no método feedforward
        como x.
56
57          activations = [x] #Cria a lista 'activations' para armazenar as
        ativações.
58
59          # Lista para armazenar todos os vetores z, camada por camada
```

```
60          zs = [] #Cria uma lista para armazenar os campos de ativação (z =
    wa + b) camada a camada.
61
62          for b, w in zip(self.biases, self.weights):
63              z = np.dot(w, activation)+b#Calcula o campo de ativação.
64              zs.append(z)#Adiciona o campo de ativação z calculado na linha
    imediatamente acima à lista 'zs=[]'.
65              activation = sigmoid(z)#aplica a função de ativação sigmoid ao
    campo de ativação.
66              activations.append(activation)#Adiciona as ativação calculada
    na linha acima à lista 'activations'
67
68          # Backward pass
69          delta = self.cost_derivative(activations[-1], y) * sigmoid_prime(zs
    [-1])#define delta como
70          nabla_b[-1] = delta
71          nabla_w[-1] = np.dot(delta, activations[-2].transpose())
72
73          # Aqui, l = 1 significa a última camada de neurônios, l = 2 é a
    segunda e assim por diante.
74          for l in range(2, self.num_layers):
75              z = zs[-l]
76              sp = sigmoid_prime(z)
77              delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
78              nabla_b[-l] = delta
79              nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
80          return (nabla_b, nabla_w)
81
82      def evaluate(self, test_data):#O método evaluate retorna o número de
    entradas de teste para as quais a rede neural produz o resultado correto
    . A saída da rede, neste caso, é o índice do neurônio da camada de saída
     com a maior ativação.
83
84          test_results = [(np.argmax(self.feedforward(x)), y) for (x, y) in
    test_data]#O código np.argmax calcula o índice com maior argumento(ativa
    ção) da saída gerada pelo método feedforward para o sinal de entrada x.
85          return sum(int(x == y) for (x, y) in test_results)# Retorna o nú
    mero de tuplas em test_results onde x=y, isto é, que a previsão foi
    correta.
```

```
86
87      def cost_derivative(self, output_activations, y):#Define o método
        cost_derivative como sendo as ativações de saída menos os respectivos ró
        tulos(previsão correta).
88          return (output_activations-y)
89
90  # Função de Ativação Sigmoide
91  def sigmoid(z):
92      return 1.0/(1.0+np.exp(-z))
93
94  # Função para retornar as derivadas da função Sigmoide
95  def sigmoid_prime(z):
96      return sigmoid(z)*(1-sigmoid(z))
```

Fonte: http://deeplearningbook.com.br/construindo-uma-rede-neural-com-linguagem-python/ Acesso em: 6 de set. 2020

As imagens do conjunto de dados MNIST são arrays numpy 28x28. Vamos utilizar os códigos abaixo para converter cada array 28x28 para 784x1 que é a forma adequada para a camada de entrada da rede que terá 784 neurônios.

```
1  import pickle
2  import gzip
3  import numpy as np
4
5  def load_data():
6      f = gzip.open('mnist.pkl.gz', 'rb')
7      training_data, validation_data, test_data = pickle.load(f, encoding="
       latin1")# O método pickle.load ler os dados f. Ao definir a codificação
       para latin1 permite importar os dados diretamente.
8      f.close()#fecha o arquivo f.
9      return (training_data, validation_data, test_data)
10
11 def load_data_wrapper():
12     tr_d, va_d, te_d = load_data()#Armazena os conjuntos de treinamento,
       validação e testes nas variáveis tr_d, va_d e te_d respectivamente.
13     training_inputs = [np.reshape(x, (784, 1)) for x in tr_d[0]]#Muda o
       formato das imagens conjunto de treinamento de 28x28 para 784x1,
       enfileira os pixels.
14     training_results = [vectorized_result(y) for y in tr_d[1]]#Cria o
```

```
    conjunto de vetores coluna com os rótulos do conjunto de treinamento. Ex
    .: se rótulo é 1 cria o vetor (0,1,0,0,0,0,0,0,0,0)^T.
15      training_data = zip(training_inputs, training_results)# Cria o conjunto
         training_data através da função interna zip que retorna uma lista de
        tupla onde a i-ésima tupla é formada pelos i-ésimos elementos dos
        argumentos de zip que são training_inputs e training_results.
16      validation_inputs = [np.reshape(x, (784, 1)) for x in va_d[0]]#Muda o
        formato das imagens conjunto de validação de 28x28 para 784x1, enfileira
         os pixels.
17      validation_data = zip(validation_inputs, va_d[1])# Cria o conjunto
        validation_data através da função interna zip que retorna um lista de
        tupla onde a i-ésima tupla é formada pelos i-ésimos elementos dos
        argumentos de zip que são validation_inputs e va_d[1].
18      test_inputs = [np.reshape(x, (784, 1)) for x in te_d[0]]#Muda o formato
         das imagens conjunto de testes de 28x28 para 784x1, enfileira os pixels
        .
19      test_data = zip(test_inputs, te_d[1])# Cria o conjunto test_data atravé
        s da função interna zip que retorna um lista de tupla onde a i-ésima
        tupla é formada pelos i-ésimos elementos dos argumentos de zip que são
        test_inputs e te_d[1].
20      return (training_data, validation_data, test_data) #Retorna os
        conjuntos training_data, validation_data, test_data no formato de imagem
         adequado para a rede receber em sua camada de entrada 784x1 pixels.
21
22  def vectorized_result(j):
23      e = np.zeros((10, 1))
24      e[j] = 1.0
25      return e
```

Nesse ponto, podemos escolher o número de camadas e neurônios por camada, para treinar os dados MNIST que foi carregado pelo método *load_data_wrapper()* no código anterior. Note que, também no código anterior, convertemos as imagens de 28x28 para 784x1, isto é, enfileiramos os pixels de cada imagem do conjunto de dados, desse modo, a rede terá três camada onde camada de entrada terá 784 neurônios(número de pixels da imagem) e camada de saída terá 10 neurônios(um neurônio para cada resultado possível), por sua vez a camada oculta poderemos testar quantidades de neurônios de modo a melhorar o desempenho da rede.

```
1 training_data , validation_data , test_data = load_data_wrapper()#Carrega os
    conjunto de treinamento , validação e de testes através do método '
    load_data_wrapper()' já convetidos.
2 training_data = list(training_data)#Insere os dados de treinamento em uma
    lista.
3
4 net = Network([784, 30, 10])#Define o número de camadas e neurônios por
    camada.
5 net.SGD(training_data , 10, 10, 3.0, test_data=test_data)#Treina a rede onde
     training_data , epochs=10, mini_batch_size=10, eta=3.0, test_data=
    test_data.
6 ... Epoch 0 : 9029 / 10000
7 ... Epoch 1 : 9180 / 10000
8 ... Epoch 2 : 9317 / 10000
9 ... Epoch 3 : 9331 / 10000
10 ... Epoch 4 : 9398 / 10000
11 ... Epoch 5 : 9383 / 10000
12 ... Epoch 6 : 9393 / 10000
13 ... Epoch 7 : 9407 / 10000
14 ... Epoch 8 : 9385 / 10000
15 ... Epoch 9 : 9423 / 10000
```

Observe que em 10 épocas a rede atinge $94,23\%$ de precisão com o conjunto de testes. Podemos agora, com a rede treinada, fazer predições de imagens específicas. Vamos utilizar o conjunto de validação validation_data para testar a rede com uma única imagem.

```
1 validation_data = list(validation_data)#Insere o conjunto de validação em
    uma lista.
```

Vamos definir a função *predict* para predizer uma única imagem através do método feedforward que gera a saída da rede ao ser apresentada uma imagem de entrada.

```
1 predict = net.feedforward
```

Agora podemos escolher uma imagem do conjunto de validação e testar a rede neural com uma determinada imagem.

```
img = validation_data[87][0]# O conjunto validation_data é uma lista de tuplas, onde cada tupla é da forma (x, y) sendo x uma imagem 784x1 e y o rótulo de 0 a 9. Desse modo, validation_data[86][0] é a tupla da 87 posição e o [0] indica que estamos tratando de x.
```

```
>>>predict(img)
...array([[1.64995963e-08],
          [9.36312714e-04],
          [8.92430435e-03],
          [9.31662871e-01],
          [6.93058375e-10],
          [9.21360949e-05],
          [4.43268556e-07],
          [1.90307805e-06],
          [5.59104491e-04],
          [7.47985014e-05]])
```

Para determinar qual o neurônio com maior ativação podemos a função numpy *argmax* como no código abaixo.

```
>>>np.argmax(predict(img))#Retorna a posição da maior componente da array, isto é, a predição da rede.
...3
```

Podemos verificar se predição está correta através da verificação do seu rótulo

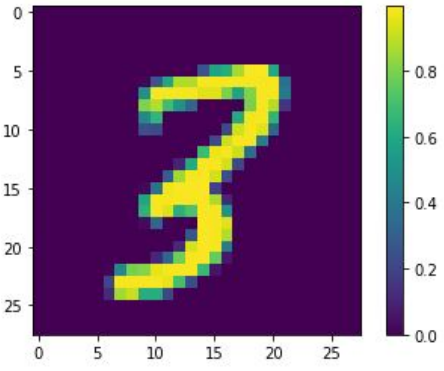
```
>>>r = validation_data[87][1]#O 87 indica a posição da tupla na lista e o 1 indica o rótulo y. Obs formato das tuplas (x, y).
>>>r
...3
```

Podemos também utilizar a biblioteca matplotlib para visualizar a imagem predita. Para isso, devemos antes, mudar o formato da imagem de acordo com o código abaixo.

```

img2 = validation_data[87][0].reshape(28, 28)# O conjunto validation_data é
    uma lista tuplas, onde cada tupla é da forma (x, y) sendo x uma imagem
    784x1 e y o rótulo de 0 a 9. Desse modo, validation_data[87][0] é a
    tupla da 87   posição e o [0] indica que estamos tratando de x. A função
     reshape muda o formato da imagem de 784x1 para 28x28 para permitir a
    visualização da imagem.
```

```
import matplotlib.pyplot as plt
```

```
plt.figure()
plt.imshow(img)
plt.colorbar()
plt.grid(False)
plt.show()
```



Vamos agora testar a rede com uma imagem que não pertence ao conjunto de dados MNIST. Podemos escrever um digito, escanear e testar se a rede consegue fazer a predição. Para isso vamos utilizar o pacote Python PIL além do Matplotlib já importado.

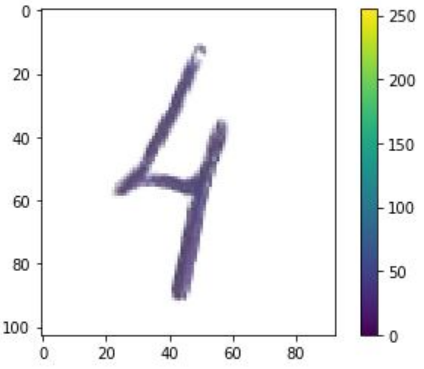
```
>>>from PIL import Image
```

Agora vamos utilizar o PIL para importar a imagem escaneada e nomeada como igm1 salva na mesma pasta do arquivo Jupyter notebook que estamos construindo a rede neural.

```
>>>image = Image.open('img4.png')
```

Para visualizar a imagem importada utilizamos os códigos abaixo.

```
>>>plt.figure()
>>>plt.imshow(image)
>>>plt.colorbar()
>>>plt.grid(False)
>>>plt.show()
```



Vamos utilizar uma sequência de códigos para preparar a imagem de acordo com as exigências da entrada da rede, isto é, vamos converter a imagem para o mesmo formato das imagens do conjunto MNIST e depois convertê-la para o formato de entrada 784x1.

```
>>>image.size #Mostra o formato da imagem carregada.
...(93, 103)
```

```
>>>img_resize = image.resize((28,28))#Converte a imagem para o formato 28
    x28 mesma dimensão das imagens do conjunto MNIST.
>>>img_resize.size # Mostra as novas dimensões da imagem.
...(28, 28)
```

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

```
>>>from numpy import asarray #importa o pacote asarray do Numpy, que tem
    função de converter uma imagem para array.
```

```
>>>data = asarray(image_cinza)#Converte a imagem para um array numpy.
```

```
>>>data.shape #Mostra o formato de array numpy.
...(28, 28)
```

```
>>>img = np.reshape(data, (784, 1))#Converte o array numpy de acordo com as
    exigências de entrada da rede, isto é, converte de 28x28 para 784x1.
```

Agora, que a imagem já foi tratada e está de acordo com as exigências da rede, podemos fazer a predição.

```
>>>predict(img)
... array([[1.49018560e-08],
           [8.19521709e-02],
           [8.44053498e-09],
           [3.88963466e-08],
           [7.15137593e-01],
           [4.54803572e-07],
           [5.80724567e-03],
           [3.34103776e-03],
           [2.34022611e-04],
           [5.97088484e-03]])
>>>np.argmax(predict(img))
...4
```

5 ROTEIRO DE APRESENTAÇÃO DE REDES NEURAIS NO ENSINO BÁSICO

É proposto para o novo ensino médio que a escola converse com a realidade atual e com os caminhos a serem seguidos pelos alunos em sua trajetória profissional, trazendo o que é essencial para o trabalho e para a vida em sociedade. A inteligência artificial está cada vez mais presente na vida das pessoas e nas mais variadas áreas como saúde, segurança e educação, sendo utilizada para resolver diversos tipos problemas.

Trabalhar aplicações relevantes nas aulas de matemática desperta a curiosidade e o interesse dos alunos para com o tema estudado e o processo de ensino-aprendizagem se torna mais atraente e significativo. Nesse sentido Moran destaca,

> Alunos curiosos e motivados facilitam enormemente o processo, estimulam as melhores qualidades do professor, tornam-se interlocutores lúcidos e parceiros de caminhada do professor-educador. Alunos motivados aprendem e ensinam, avançam mais, ajudam o professor a ajudá-los melhor. (MORAN, 2000, p.17).

Nessa perspectiva, apresentaremos aos professores do ensino médio uma proposta de ensino para educação básica, em aulas de matemática, com a temática redes neurais e Deep Learning, tendo como pressuposto que a ciência de dados está presente no nosso dia a dia e que os profissionais na área encontram-se em número reduzido, enfim, evidenciar que existe um mercado de atuação profissional muito promissor a ser desbravado.

5.1 Interpretação geométrica da derivada a partir da taxa média de variação

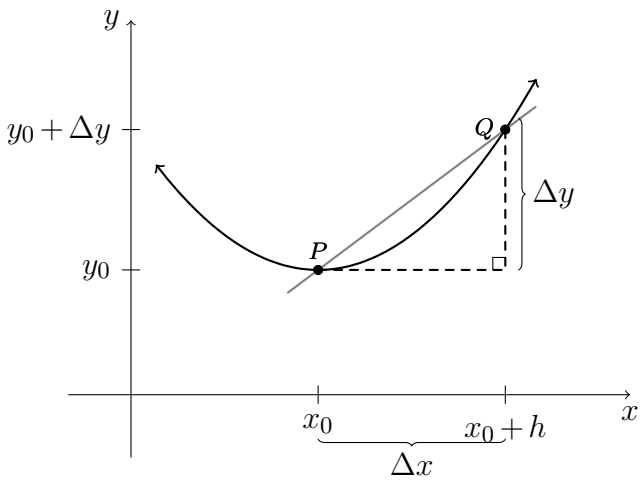
Para treinar uma rede neural devemos encontrar os pesos e os bias que minimizam a sua função de custo, para isso, vamos utilizar os conceitos de máximos e mínimos do cálculo diferencial, através do algoritmo gradiente descendente.

Para que os alunos compreendam o algoritmo utilizado para treinar as redes neurais podemos trabalhar o conceito de derivada partir de sua interpretação geométrica, a inclinação da reta tangente a uma curva, feito a partir de aproximações de retas secantes e da taxa de variação média da função.

Considere uma curva que seja gráfico de uma função $y = f(x)$ e $P = (x_0, f(x_0))$, um ponto de f, onde será traçado a reta tangente. Atribuindo a x um acréscimo $\Delta x = h$, temos

que a variável y sofrerá um acréscimo Δy, assim, saímos do ponto $P(x, y)$ para um ponto $Q = (x + \Delta x, y + \Delta y)$. Veja o gráfico abaixo.

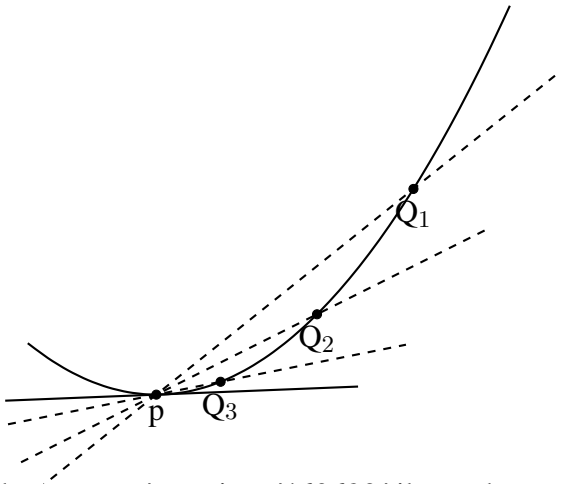
Figura 31 – Reta secante



Fonte: https://tex.stackexchange.com/questions/460632/tikz-and-secant-line-diagram. Acesso: OUT, 2020.

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 32 – Aproximação de uma reta tangente por retas secantes



Fonte: https://tex.stackexchange.com/questions/460632/tikz-and-secant-line-diagram. Acesso: OUT. 2020.

Como Q se aproxima de P, más diferente de P, temos uma posição limite para Q. Informalmente, 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.

Exemplo 5.1. seja $y = f(x) = x^2$ e $P = (2,4)$ um ponto de f e um acréscimo $\Delta x = h$ e o acréscimo de de y $\Delta y = (2+h)^2 - 4 = 4 + 4h + h_2 - 4$. Segue que o coeficiente angular da reta secante é

$$m = \frac{\Delta y}{\Delta x} = \frac{4h - h_2}{h}$$

Para determinar a derivada, basta fazer Δx se aproximar 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 se aproxima do coeficiente da reta tangente. Note que

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

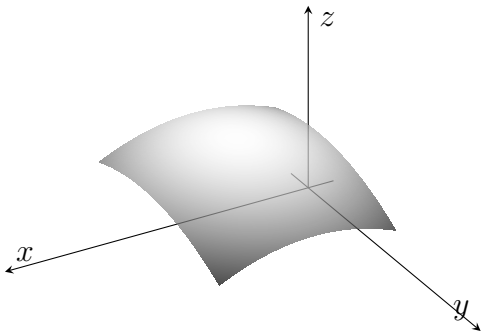
Como h tende a 0, mas nunca é igual a 0, temos:

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

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

No exemplo anterior, foi mostrado a interpretação geométrica da derivada para uma função com uma variável, más a função de custo nas redes neurais, tem como variáveis muitos pesos e bias. Vamos ver como isso funciona como duas variáveis, a interpretação para n variáveis é análoga. Considere a superfície $z = f(x,y)$ representada no gráfico abaixo.

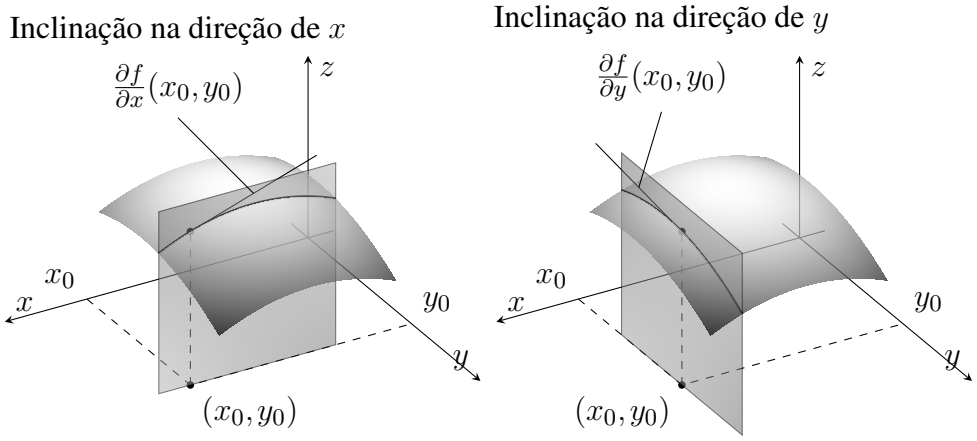
Figura 33 – Grafico de uma função de duas variáves



Fonte: https://tex.stackexchange.com/questions/479814/a-diagram-about-partial-derivatives-of-fx-y. Acesso: OUT, 2020.

As funções de custos costumam ter gráficos irregulares com muitos vales, mas para fins de exemplo, essa superfície é satisfatória. A interpretação geométrica das derivadas parciais de $z = f(x, y)$ são análogas a de uma variável. A derivada parcial de com relação a x, $\frac{\partial f}{\partial x}$ no ponto (x_0, y_0) é a inclinação da reta tangente a a curva gerada pela interseção do plano $y = y_0$ com a superfície f, do mesmo modo, a derivada parcial de com relação a y, $\frac{\partial f}{\partial y}$ no ponto (x_0, y_0) é a inclinação da reta tangente a curva gerada pela interseção do plano $x = x_0$ com a superfície f. Veja os gráficos abaixo.

Figura 34 – Interpetração geométrica da derivada parcial em funções de duas variáves



Fonte: https://tex.stackexchange.com/questions/479814/a-diagram-about-partial-derivatives-of-fx-y. Acesso: OUT, 2020.

Exemplo 5.2. Considere a função $z = f(x, y) = x^2 + y^2 + 10$. Para calcular a derivada parcial com relação a x, $\frac{\partial f}{\partial x}$, devemos considerar $y = y_0$ uma constante e derivar com relação a x do mesmo modo como em função real de uma variável real.

$$\frac{\partial f}{\partial x} = 2x$$

Analogamente temos:

$$\frac{\partial f}{\partial y} = 2y$$

Onde $\frac{\partial f}{\partial x} = 2x$ e $\frac{\partial f}{\partial y} = 2y$ são as taxas de variação da superfície nas direções dos eixos x e y respectivamente.

5.2 Deep learning 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* 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 que 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 é carregado o conjunto de dados na forma de quatro NumPy arrays onde:

- 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 a respectivo dígito conforme a tabela abaixo.

Figura 35 – Tabela de correspondencia entre classes e rótulos

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

Fonte: Elaborada pelo autor.

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

```
1 >>> class_names = ['Zero', 'Um', 'Dois', 'Três', 'Quatro',
2                   '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
```

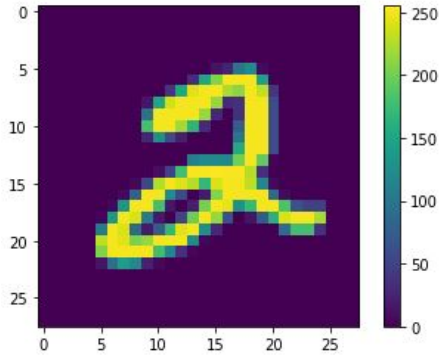
Note que o conjunto dos rótulos é uma numpy array de inteiros como dito antes, veja.

```
>>> 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
```

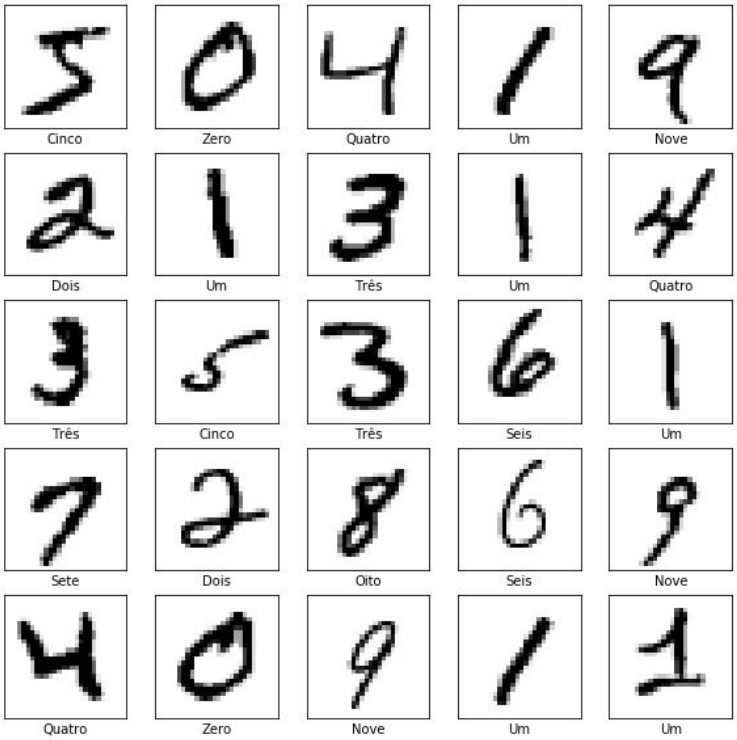


Observe, na imagem acima, que a tonalidade dos pixels das imagens do conjunto MNIST varia de 0 a 255. Antes do treinamento do modelo, é conveniente escalar os valores dos pixels entre 0 e 1, para isso, dividimos os valores por 255.0. É importante observar que o conjunto de treinamento e o conjunto de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

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

Podemos analisar se os dados estão prontos para construção e treinamento da rede neural, para isso, vamos utilizar o código abaixo para visualizar as 16 primeiras imagens do conjunto de treinamento com os respectivos nomes das classes à qual pertence.

```
>>>plt.figure(figsize=(10,10)) #Crie ou ative uma figura. existente/figsize
    (float,float): determina a largura e altura da imagem em polegadas.
>>>for i in range(16): # para i em range(25)=[0,2,...,24]
...     plt.subplot(5,5,i+1)#Divide a imagem em 25 imagens.
...     plt.xticks([])#retira a escala do eixo x.
...     plt.yticks([])#retira a escala do eixo y.
...     plt.grid(False)
...     plt.imshow(train_images[i], cmap=plt.cm.binary)#Exibe os dados como
    uma imagem em 2D. train_images[i]:i-ésima imagem de train\_images
...     plt.xlabel(class_names[train_labels[i]]), cmap: colormap define a
    cor em escala de cinza.
>>>plt.show() #Mostra a figura
```



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.

Definição 5.1. 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 esta camada só tem esta 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 de 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.

```
1 >>>model.compile(optimizer='adam',
2 ...                   loss=tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True),
3 ...                   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 minimizá-lo.
- *Metrics* Usada para monitoramento dos passos de treinamento e teste. No caso de *accuracy* se trata da fração das imagens que foram classificadas corretamente.

5.2.1 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. É feito predições com conjunto de teste que neste caso é a array test_imagens

4. É feito a verificação da precisão das 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 a 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 (sobreajute).

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
```

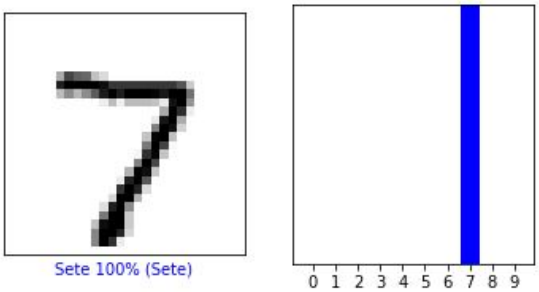
O que mostra que a rede neural fez a previsão correta. Podemos fazer um gráfico de barras indicando as probabilidades de cada classe para uma determinada imagem utilizando os códigos a seguir.

```
>>> 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)
```

```
19
20 >>>def plot_value_array(i, predictions_array, true_label):
21 ...     true_label = true_label[i]
22 ...     plt.grid(False)
23 ...     plt.xticks(range(10))
24 ...     plt.yticks([])
25 ...     thisplot = plt.bar(range(10), predictions_array, color="#777777")
26 ...     plt.ylim([0, 1])
27 ...     predicted_label = np.argmax(predictions_array)
28
29 ...     thisplot[predicted_label].set_color('red')
30 ...     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.

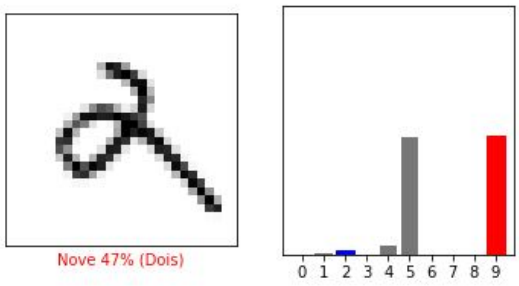
```
1 >>> i = 0
2 >>>plt.figure(figsize=(6,3))
3 >>>plt.subplot(1,2,1)
4 >>>plot_image(i, predictions[i], test_labels, test_images)
5 >>>plt.subplot(1,2,2)
6 >>>plot_value_array(i, predictions[i],  test_labels)
7 >>>plt.show()
```




```
8
```

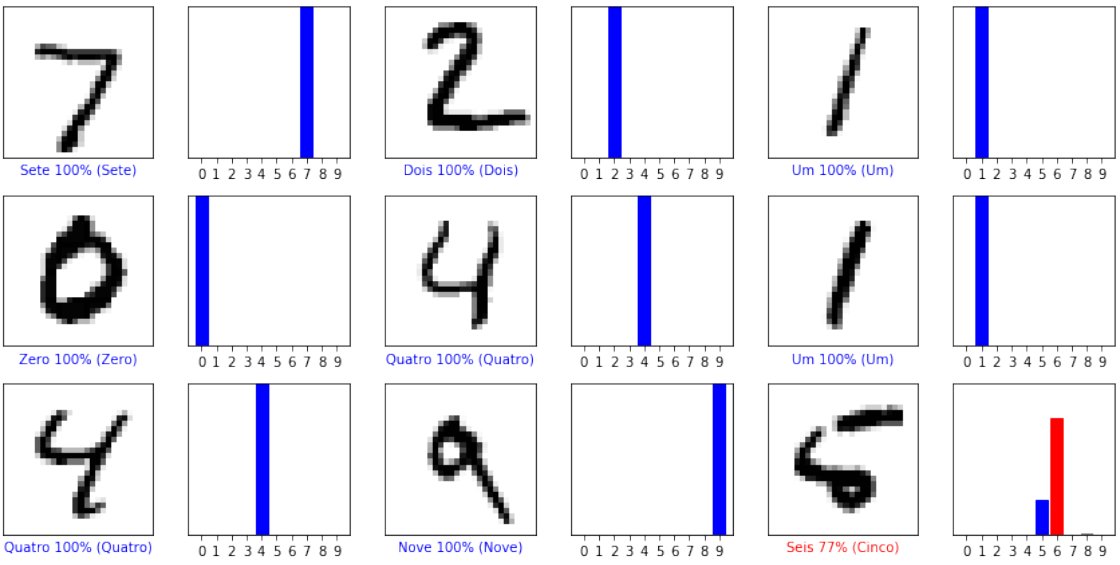
```
1 >>> i = 149
2 >>>plt.figure(figsize=(6,3))
3 >>>plt.subplot(1,2,1)
4 >>>plot_image(i, predictions[i], test_labels, test_images)
5 >>>plt.subplot(1,2,2)
6 >>>plot_value_array(i, predictions[i],  test_labels)
```

```
>>> plt.show()
```



Podemos plotar as n primeiras predições, com seus respectivos rótulos, com as mesmas configurações da plotagem anterior, veja o código.

```
num_rows = 3
num_cols = 3
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
  plt.subplot(num_rows, 2*num_cols, 2*i+1)
  plot_image(i, predictions[i], test_labels, test_images)
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_value_array(i, predictions[i], test_labels)
plt.tight_layout()
plt.show()
```



Enfim, como 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)
```

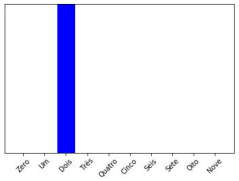
Note que a imagem tem o tamanho de 28x28 mesmo tamanho de entrada da rede. O *tf.keras* é otimizado para fazer previsões em lote de imagens a cada vez. Desse modo temos que criar um lote com apenas uma imagem.

```
>>>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
```

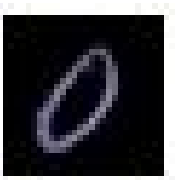
```

>>>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).
```

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



```
3
```

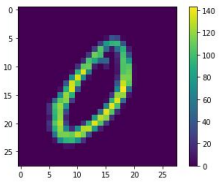
Como a rede foi configurada para fazer previsões em lote de imagens a cada vez, vamos criar um lote com image_cinza.

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

```
1 >>>img2.shape #Retorna a forma da matriz, isto é, uma tupla referente ao nú
      mero de elementos de cada dimensão.
2 ...(1, 28, 28) #Eixo inserido na primeira posição: axis=0
```

Podemos visualizar a imagem com o código abaixo

```
1 >>>plt.figure()
2 >>>plt.imshow(img3[0])
3 >>>plt.colorbar()
4 >>>plt.grid(False)
5 >>>plt.show()
```




```
6
```

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)
```

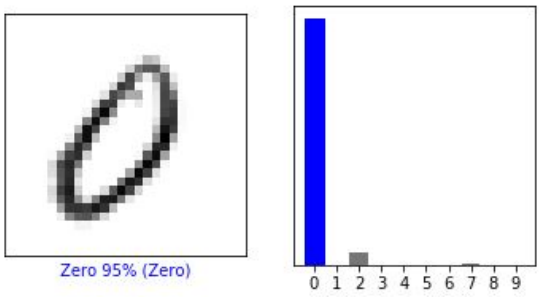
```
>>>predict
...array([[9.4660956e-01, 1.8237397e-04, 4.8826247e-02, 1.5532545e-06,
        1.2159522e-03, 2.3121331e-05, 3.6144556e-04, 2.6822658e-03,
        9.1493923e-05, 5.8982700e-06]], dtype=float32)
```

```
>>>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()
```



8

CONSIDERAÇÕES FINAIS

Diante de uma sociedade moderna é evidente a necessidade de uma educação de qualidade para todos, porém tem sido um desafio para os professores proporcionar aulas inovadoras. O mundo educacional tem mudado e exige-se um profissional capacitado que tenha conhecimento multidisciplinar, capaz de interagir e manipular as novas tecnologias, que está cada vez mais presente nas mais diversas atividades que realizamos, não sendo diferente na educação, trazendo novas possibilidades para enriquecer as aulas.

Assim, o presente trabalho propõe aos professores de matemática a utilização de redes neurais e deep learning no ensino básico, com a finalidade de se adequar as novas realidades e deixar o ensino de matemática mais significativo, visando combater o desinteresse dos alunos, visto que, temas que envolve tecnologia são agradáveis aos estudantes tornando as aulas mais atrativas e participativas.

A apresentação aos docentes de propostas de trabalhos de ensino-aprendizagem, com temas atuais e relevantes, além de melhorar a qualidade do processo de ensino-aprendizagem, pode despertar, nos alunos, o interesse na área estudada, abrindo assim, novas opções para escolha das carreiras profissionais a seguir.

A formação continuada de professores de matemática, necessita de estratégias que ofereçam um significado relevante e atual à construção de seus conceitos. E, nesse contexto, este trabalho mostra que novas tecnologias têm grande potencial para enriquecer a prática pedagógica no ensino de matemática.

Portanto, as novas práticas interdisciplinares no ensino complementam a aprendizagem de forma relevante, faz o aluno gostar de aprender, impulsionado por conhecimentos atuais e relevantes para seu modo de viver, transformando realidade escolar e construindo um momento de êxito na prática pedagógica.

REFERÊNCIAS

ÁVILA, Geraldo. Limites e Derivadas do Ensino Médio?. **Revista Professor de Matemática, n° 60**,Rio de Janeiro, p. 30-38, mai./ago.2006.

CRESPO, A. A. **Estatística Fácil**. 17.ed. São Paulo: Saraiva, 2002

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

GRUS, Joel. **Data Science do Zero**. 2. ed. São Paulo: Novatec, 2014.

HAYKIN, Simon. **Redes Neurais: Princíos e prática**. 2nd Edition. Porto Alegre: Bookman, 2008.

JAMES, Gareth et al. **An Introduction to Statistical Learning**. New York 2013: Springer, 2013 (Corrected at 8th printing 2017).

MARTINS, F. E. **Estatística e Probabilidade**. 2ª.ed. São Paulo: ATLAS, 2010.

MATPLOTLIB. **Matplotlib Version 3.1.2**. Disponível em: <https://matplotlib.org/3.1.1/index.html>. Acessado em: 18 de Junho, 2020.

MENEZES, Nilo Ney Coutinho. **Introdução à Programação com Python** 2. ed. São Paulo: Novatec, 2014.

MORAN, José Manuel. MASSETTO, Marcos Tarciso. BEHRENS, Marilda Aparecida. **Novas Tecnologias E Mediação Pedagógica**. 13.ed. Campinas-SP: Papirus, 2002.

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

PINTO, Diomara. MORGADO, Maria Cândida Ferreira. **Cáculo Diferencial e Integral de Funções de Várias Variáveis**. 3ª Edição. Rio de Janeiro: Editora UFRJ, 2009.

PYTHON. **The Python Tutorial 3.8**. Disponível em: <http://python.org/>. Acessado em: 12 de abril, 2020.

RUSSELL, Stuart. **Inteligência Artificial**. Tradução da 3nd Edition. Rio de Janeiro: Elsevier, 2013.

SANTIAGO, Luiz. **Entendendo a biblioteca NumPy**. Disponível em: <https://medium.com/ensina-ai/entendendo-a-biblioteca-numpy-4858fde63355>. Acessado em: 22 de Julho, 2020.

TENSORFLOW. **Treinamente de Redes Neurais**. Disponível em: <https://www.tensorflow.org/>. Acessado em: 20 de setembro, 2020.

A NOÇÕES DE NUMPY

O Numpy é uma biblioteca científica do Python feita para trabalhar com álgebra linear. A classe array do Numpy é chamada de ndarray e oferece mais funcionalidades que a biblioteca array padrão do Python que trabalha apenas com matrizes unidimensionais. O numpy traz uma grande quantidade de funções e operações amplamente utilizadas nos algoritmos de modelos de aprendizado de máquina, processamento de imagens e computação gráfica, além de tarefas matemáticas como integração numérica, diferenciação interpolação. A biblioteca pode ser usada como substituto do MATLAB combinado com o Scipy e Matplotlib.

Os Arrays NumPy são usados para armazenar os dados de treinamento dos mais diversos tipos como imagens que são representadas por array multidimensional, e também os parâmetros de aprendizado do modelo utilizado.

Como estamos trabalhando com o pacote anaconda o numpy já vem instalado e o que temos que fazer apenas é importar o *numpy* com o comando abaixo.

Exemplo A.1.

```
1 >>> import numpy as np#importa a biblioteca NumPy e renomeia como np.
```

O Conceito mais relevante no NumPy é o Array, chamado *ndarray*, que é uma tabela de elementos todos do mesmo tipo, normalmente números, com grande número de funções e operações, o que facilita a velocidade de escrita do código e com desempenho excelente. Veja como criar um array NumPy no exemplo abaixo.

Exemplo A.2.

```
1 >>> arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
2 >>> arr
3 ...array([[ 1,  2,  3,  4,  5],
4        [ 6,  7,  8,  9, 10]])
```

No exemplo acima temos um array com duas dimensões com cinco elementos.

Podemos criar arrays unidimensionais com sequencias de números com o atributo *np.arrange()*, para redimensionar um array usamos o atributo *.reshape*, veja no exemplo abaixo.

Exemplo A.3.

```
>>> arr = np.arange(4)
>>> arr
...array([0, 1, 2, 3])
>>> arr_r = np.arange(4).reshape(2, 2)
>>> arr_r
...array([[0, 1],
       [2, 3]])
```

Com arrays NumPy podemos somar, subtrair, multiplicar e dividir, vejamos alguns exemplos de operações com arrays.

Exemplo A.4.

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[5, 6], [7, 8]])
>>> a+b
... array([[ 6,  8],
       [10, 12]])
>>> a-b
... array([[ 5, 12],
       [21, 32]])
>>> a*b # Note que esta multiplicação é feita elemento à elemento, não é a
    usual de matrizes.
... array([[ 5, 12],
       [21, 32]])
>>> a/b
... array([[0.2       , 0.33333333],
       [0.42857143, 0.5       ]])
```

A multiplicação de arrays usual, pode é feita com o método *.dot*.

Exemplo A.5.

```
>>> np.dot(a, b)
... array([[19, 22],
       [43, 50]])
```

A função Numpy *np.random.randn()* é usada para criar um Numpy array aleatoriamente, de modo que, os elemento geram distribuições gaussianas com 0 de média e 1 de desvio padrão.

Exemplo A.6.

```
>>> k = np.random.randn(3,3)
>>> k
...array([[-1.23689552,  1.11827602,  0.31976829],
       [ 0.09536155, -1.86599742, -1.00528972],
       [-0.75937376, -0.79048878, -0.37162402]])
>>>np.mean(k) #Calcula a média aritmética do elemento do array Numpy.
...-0.49958482009219174
>>>k.std() #Calcula o desvio padrão dos elemento do array Numpy.
...0.8483783002343603
```

Podemos calcular o produto escalar de matrizes em python usando a função *numpy.dot()*.

Exemplo A.7.

```
>>>v = np.array([(1, 2), (2, 2)])
>>>v
...array([[1, 2],
       [2, 2]])
>>>w = np.array([(2, 2), (1, 2)])
>>>w
...array([[2, 2],
       [1, 2]])
>>>u = np.dot(v, w)
...array([[4, 6],
       [6, 8]])
```

B NOÇÕES DE PANDAS

O Pandas é a mais completa biblioteca Python usada para manipulação, leitura e visualização de dados, sendo fundamental para se trabalhar com análise de dados. O Pandas já vem instalado e o que temos que fazer apenas é importar o *Pandas* com o comando abaixo.

Exemplo B.1.

```
>>> import pandas as pd#importa a biblioteca Pandas e renomeia como pd.
```

Em Pandas, destacamos duas estruturas de dados:

- **Séries** são objetos unidimensionais, uma lista de valores, semelhante à um array, que possui índice, chamado *index*.

```
>>> s = pd.Series([2,4,6,8,10])
>>> s
0     2
1     4
2     6
3     8
4    10
dtype: int64
```

Note que, como não definimos os índices, foi gerado o padrão. Veja, no código abaixo, como podemos definir índices.

```
>>> idades = pd.Series([5,4,6,9,15], index=['Maria', 'Rafael', 'Júlia'
    , 'Carlos', 'Marcos'])
>>> idades
Maria     5
Rafael    4
Júlia     6
Carlos    9
Marcos   15
dtype: int64
```

A função do *index* facilitar a referência do valor usando o seu rótulo.

```
>>> idades['Maria']
10
```

Outras funcionalidades proporcionadas pela estrutura Series é que seus métodos fornecem dados estatísticos sobre seu valores, como média *.mean()*, variância *.var()* e desvio padrão *.std()*. Podemos utilizar o código *.describe()* para resumir as estatísticas da Series.

```
>>> idades.describe()
count      5.000000
mean       7.800000
std        4.438468
min        4.000000
25\%       5.000000
50\%       6.000000
75\%       9.000000
max       15.000000
dtype: float64
```

- **DataFrame**: É uma estrutura bidimensional, análoga a uma planilha. veja o código abaixo.

```
>>> df = pd.DataFrame({'Aluno':['Maria', 'Rafael', 'Júlia', 'Carlos',
    'Marcos'], 'Notas':[6.5, 7.0, 9.0, 8.5, 7.5], 'Faltas'
    :[1,0,3,2,5]})
>>>df
```

	Aluno	Notas	Faltas
0	Maria	6.5	1
1	Rafael	7.0	0
2	Júlia	9.0	3
3	Carlos	8.5	2
4	Marcos	7.5	5

Podemos acessar os valores de uma coluna pelo respectivo nome.

```
>>> df['Notas']
0    6.5
1    7.0
2    9.0
3    8.5
4    7.5
Name: Notas, dtype: float64
```

Podemos, também, utilizar o método *.describe()* para visualizar o resumo dos dados estatísticos do DataFrame.

```
>>> df.describe()
```

	Notas	Faltas
count	5.000000	5.000000
mean	7.700000	2.200000
std	1.036822	1.923538
min	6.500000	0.000000
25%	7.000000	1.000000
50%	7.500000	2.000000
75%	8.500000	3.000000
max	9.000000	5.000000

Podemos utilizar o método *loc* para resgatar dados do DatraFrame pelo index.

```
>>> df.loc[3]
Aluno      Carlos
Notas         8.5
Faltas          2
Name: 3, dtype: object
```

O Pandas pode, também, fornece uma extensa gama de funcionalidades para leitura e análise de dados prontos. O Pandas pode ler diversos tipos de arquivos como csv, xlsx e html, para isso, basta utilizar os códigos *pd.read_csv*, *pd.read_xlsx* e *pd.read_html* respectivamente. Vamos carregar os dados housing.csv que mostra o preço de casas em diferentes regiões da California.

```
>>> pd.read\_csv('housing.csv')
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	NEAR BAY
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	NEAR BAY
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0	NEAR BAY
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0	NEAR BAY
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0	NEAR BAY
...	...	...	...	...	...	...	...	...	...	...
20635	-121.09	39.48	25.0	1665.0	374.0	845.0	330.0	1.5603	78100.0	INLAND
20636	-121.21	39.49	18.0	697.0	150.0	356.0	114.0	2.5568	77100.0	INLAND
20637	-121.22	39.43	17.0	2254.0	485.0	1007.0	433.0	1.7000	92300.0	INLAND
20638	-121.32	39.43	18.0	1860.0	409.0	741.0	349.0	1.8672	84700.0	INLAND
20639	-121.24	39.37	16.0	2785.0	616.0	1387.0	530.0	2.3886	89400.0	INLAND

20640 rows × 10 columns

C NOÇÕES DE MATPLOTLIB

O Matplotlib é uma biblioteca Python destinada para criar gráficos em Python. Para utilizarmos essa biblioteca devemos importá-la com o seguinte código.

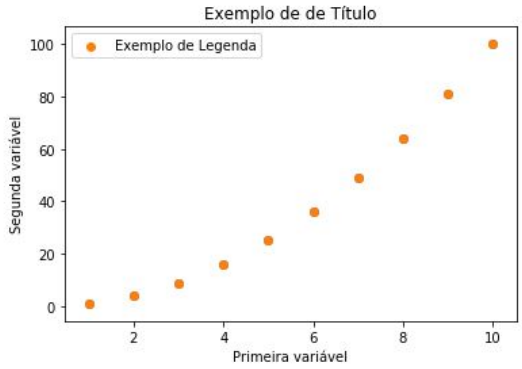
Exemplo C.1.

```
>>> import matplotlib.pyplot as plt
```

O pyplot é uma estrutura do matplotlib que o faz funcionar de modo semelhante ao MATLAB, isto é, cada função do pyplot faz uma alteração no gráfico. Assim podemos criar gráficos com poucas linhas de código e de modo muito intuitivo como no exemplo abaixo.

Exemplo C.2.

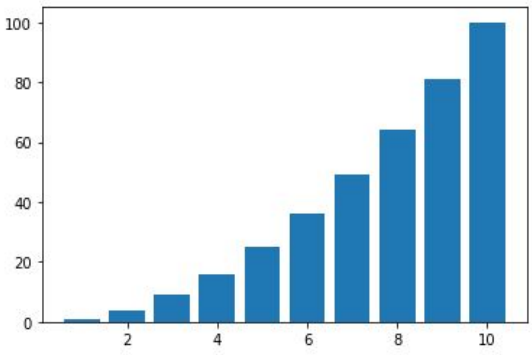
```
#Primeiro definimos as variáveis
>>> x = [1, 2 , 3, 4, 5, 6, 7, 8, 9, 10]
>>>y = [1, 4 , 9, 16, 25, 36, 49, 64, 81, 100]

# Inserindo descrições no gráfico
>>>plt.scatter(x,y) #cria um gráfico de dispersão
>>>plt.title('Exemplo de de Título')
>>>plt.xlabel('Primeira variável')
>>>plt.ylabel('Segunda variável')

#Inseri uma legenda para o gráfico de dispersão.
>>>plt.scatter(x, y, label = 'Exemplo de Legenda')
>>>plt.legend()

#Mostra o gráfico criado
>>>plt.show
```



17

Exemplo C.3.

```
1 >>> plt.bar(x, y) #cria um gráfico de barras
2 >>> plt.show #mostra o gráfico
```



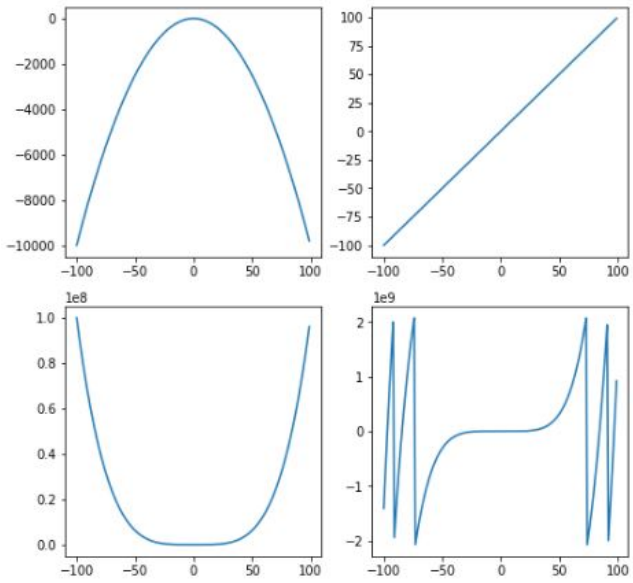
3

Podemos também construir mais de um gráfico em uma única figura, de uma só vez, usando o comando *plt.subplot* como no exemplo abaixo.

Exemplo C.4.

```
1 >>>z = np.arange(-100,100,1)
2 >>>f = -z**2-4
3 >>>g = z
4 >>>h = z**4
5 >>>i = z**5
6 >>> plt.figure(figsize=(8,8))#define a altura e a largura da figura
7 >>>plt.subplot(2,2,1)
8 >>>plt.plot(z, f)
9 >>>plt.subplot(2,2,2)
```

```
>>> plt.plot(z, g)
>>> plt.subplot(2,2,3)
>>> plt.plot(z, h)
>>> plt.subplot(2,2,4)
>>> plt.plot(z, i)
>>> plt.show
```



Como vimos o Matplotlib disponibiliza diversas funcionalidades para personalizar os gráficos sendo, desse modo, uma biblioteca muito utilizada para tratamento de dados.