

**UNIVERSIDADE FEDERAL FLUMINENSE**  
**ANTONIO CARLOS DO NASCIMENTO CUNHA JUNIOR**

**ESTUDO E IMPLEMENTAÇÃO DE REDES NEURAIS PROFUNDAS  
COMO SOLUÇÃO PARA UM PROBLEMA DE CLASSIFICAÇÃO**

**Niterói**  
**2019**

**ANTONIO CARLOS DO NASCIMENTO CUNHA JUNIOR**

**ESTUDO E IMPLEMENTAÇÃO DE REDES NEURAIS PROFUNDAS  
COMO SOLUÇÃO PARA UM PROBLEMA DE CLASSIFICAÇÃO**

Trabalho de Conclusão de Curso  
submetido ao Curso de Tecnologia em  
Sistemas de Computação da  
Universidade Federal Fluminense como  
requisito parcial para obtenção do título  
de Tecnólogo em Sistemas de  
Computação.

**Orientador(a):  
ALTOBELLI DE BRITO MANTUAN**

**NITERÓI  
2019**

Ficha catalográfica automática - SDC/BEE  
Gerada com informações fornecidas pelo autor

C972e Cunha junior, Antonio Carlos do Nascimento  
Estudo e implementação de Redes Neurais Profundas como  
solução para um problema de classificação / Antonio Carlos  
do Nascimento Cunha junior ; Altobelli de Brito Mantuan,  
orientador. Niterói, 2019.  
50 f. : il.

Trabalho de Conclusão de Curso (Graduação em Tecnologia  
de Sistemas de Computação)-Universidade Federal Fluminense,  
Instituto de Computação, Niterói, 2019.

1. Inteligência Artificial. 2. Aprendizado de Máquina. 3.  
Classificação de Dados. 4. Produção intelectual. I.  
Mantuan, Altobelli de Brito, orientador. II. Universidade  
Federal Fluminense. Instituto de Computação. III. Título.

CDD -

Bibliotecária responsável: Fabiana Menezes Santos da Silva - CRB7/5274

**ANTONIO CARLOS DO NASCIMENTO CUNHA JUNIOR**

**ESTUDO E IMPLEMENTAÇÃO DE REDES NEURAIS PROFUNDAS  
COMO SOLUÇÃO PARA UM PROBLEMA DE CLASSIFICAÇÃO**

Trabalho de Conclusão de Curso  
submetido ao Curso de Tecnologia em  
Sistemas de Computação da  
Universidade Federal Fluminense como  
requisito parcial para obtenção do título  
de Tecnólogo em Sistemas de  
Computação.

Niterói, 6 de Dezembro de 2019.

Banca Examinadora:

---

Prof. Altobelli de Brito Mantuan, Msc. – Orientador  
UFF –Universidade Federal Fluminense

---

Prof. Eduardo Vera Sousa, Msc. – Avaliador  
UFF –Universidade Federal Fluminense

## **AGRADECIMENTOS**

Primeiramente agradeço a Deus, que sempre está comigo em todos os momentos.

Aos meus pais e minha irmã, por serem exemplos de força e perseverança constantes ao meu lado.

A todos os professores que passaram por minha vida, e que plantaram as sementes que germinam todos os dias.

Ao meu orientador, por tudo que ensinou e pela paciência ao longo deste processo.

E a Carl Sagan, principal responsável por despertar em mim o desejo de sempre buscar respostas.

*“Para mim, acentua nossa responsabilidade para nos portar mais amavelmente uns para com os outros, e para protegemos e acarinharmos o pálido ponto azul...o único lar que nós conhecemos.”.*

*Carl Sagan*

## RESUMO

O aprendizado de máquina, é a capacidade que computadores tem de aprenderem sem serem programados de forma explícita a esse fim. Explora a capacidade de aprender com os próprios erros e usar essa informação para aprimoramento na previsão de dados. Nesse trabalho, estudaremos como redes neurais artificiais são utilizadas nesse processo de aprendizagem, sua estrutura e os termos que compõem esse campo de estudos ligados a inteligência artificial. Aqui será implementada uma rede classificatória visando a predição de um determinado tipo de planta com base em algumas de suas características físicas. O algoritmo utilizado, apresentou uma taxa de acurácia acima de noventa por cento, provando sua eficiência para a solução do problema apresentado.

**Palavras-chaves:** Redes Neurais, Aprendizado de Máquina e Inteligência Artificial.

## **ABSTRACT**

Nowadays there is a huge consensus that machine learning has a fundamental participation in the improvement of artificial intelligence. In the fifties, Arthur Samuel defined machine learning as the computer ability to make predictions or decisions without being specifically programmed to perform them. It does not only learn from its own mistakes, but also take advantage of eventual failures to improve itself. In this present work we will study how neural networks are useful in this learning process, its structure and the common terms and definitions used in this magnificent field of study. Later, we will make the implementation of a neural network for classification purpose, which will predict the type of a plant species with basis on some of its physical attributes. Our algorithm has showed an accuracy rate above ninety percent, which provide us with strong evidences to prove its efficiency and achieve the solution to our problem.

**Key words: Neural Networks, Machine Learning and Artificial Intelligence.**



## LISTA DE ILUSTRAÇÕES

Figura 01: Programadoras trabalhando no <i>ENIAC</i> .....	15
Figura 02: Relação de Aprendizado de Máquina com Mineração de Dados .....	18
Figura 01: Funcionamento das camadas de uma Rede Neural .....	20
Figura 04: Representação de um Neurônio Artificial .....	20
Figura 05: Exemplo de Função de Ativação Linear .....	22
Figura 06: Exemplo de representação de Função de Ativação Não Linear .....	23
Figura 07: Exemplo de valores iniciados em uma RNA .....	24
Figura 08: Fórmula e representação da função Logística .....	25
Figura 09: Ilustração da Regra da Cadeia no nó da camada de saída .....	28
Figura 10: Código: Colunas e impressão por comando <i>head</i> .....	33
Figura 11: Código: Mapeamento da saída como colunas .....	34
Figura 12: Código: Função Split aplicada a treino e teste .....	34
Figura 13: Código: Impressão para verificação de treino e teste .....	35
Figura 14: Código: Modelagem das camadas da Rede Neural .....	36
Figura 15: Código: Treino do modelo da rede .....	37
Figura 16: Código: Comparação de curvas referentes a perda .....	38
Figura 17: Código: Comparação da função <i>Predict</i> com valores reais .....	49
Figura 18: Código: Predição dos casos de treino e acurácia do algoritmo .....	40
Figura 19: Código: Predição dos casos de teste e acurácia do algoritmo .....	41
Figura 20: Código: Alternativa ao modelo de rede, curvas .....	43
Figura 21: Código: Alternativa ao modelo de rede, matriz e acurácia .....	44

## LISTA DE EQUAÇÕES

Equação 1: Modelo de Neurônio .....	21
Equação 2: Função de Erro ou Custo .....	27
Equação 2: Gradiente em Relação a $w_i$ .....	28

## LISTA DE ABREVIATURAS E SIGLAS

ENIAC – *Electronic Numerical Integrator and Computer*

IA – Inteligência Artificial

ML – *Machine Learning*

TANH – *Hyperbolic Tangent Function*

RELU – *Rectified Linear Unit*

MSE – *Mean Squared Error*

GPU - *Graphics processing unit*

## SUMÁRIO

RESUMO .....	7
ABSTRACT .....	8
LISTA DE ILUSTRAÇÕES.....	9
LISTA DE EQUAÇÕES .....	10
LISTA DE ABREVIATURAS E SIGLAS .....	11
1 INTRODUÇÃO .....	14
2 FUNDAMENTAÇÃO TEÓRICA.....	15
2.1 APRENDIZADO DE MÁQUINA ( <i>MACHINE LEARNING</i> ) .....	15
2.1.1 ORIGEM NA INTELIGÊNCIA ARTIFICIAL .....	16
2.1.2 ÁREAS DE UTILIZAÇÃO E APLICAÇÃO.....	17
2.1.3 RELAÇÃO COM MINERAÇÃO DE DADOS .....	17
2.1.4 DADOS ESTRUTURADOS E NÃO ESTRUTURADOS .....	18
2.1.5 MÉTODOS DE APRENDIZADO .....	19
2.2 REDES NEURAIS ARTIFICIAIS .....	19
2.2.1 ARQUITETURAS E CAMADAS .....	20
2.2.2 NÓS (NEURÔNIOS) E PESOS DAS CONEXÕES .....	20
2.2.3 <i>BIAS</i> .....	21
2.2.4 FUNÇÕES DE ATIVAÇÃO .....	21
2.3 <i>BACKPROPAGATION</i> .....	23
2.3.1 INICIANDO UMA REDE .....	24
2.3.2 USANDO ATIVAÇÃO LOGÍSTICA .....	25
2.3.3 CÁLCULO DO ERRO .....	26
2.3.4 O GRADIENTE E A APLICAÇÃO DA REGRA DA CADEIA .....	28
2.3.5 O GRADIENTE DO ERRO TOTAL EM RELAÇÃO A SAÍDA .....	29

2.3.6	O GRADIENTE DA SAÍDA EM RELAÇÃO A SOMA.....	29
2.3.7	O GRADIENTE DA SOMA EM RELAÇÃO AO PESO .....	30
2.3.8	TAXA DE APRENDIZADO .....	30
3	IMPLEMENTAÇÃO.....	32
3.1	PROPOSTA .....	32
3.2	CONFIGURANDO A REDE .....	33
3.3	MODELAGEM E COMPILAÇÃO .....	36
3.4	ACURÁCIA DA REDE .....	40
3.5	TESTANDO OUTRA CONFIGURAÇÃO .....	42
4	CONCLUSÕES .....	45
	REFERÊNCIAS BIBLIOGRÁFICAS .....	47
	ANEXO .....	49

# 1 INTRODUÇÃO

A motivação para desenvolvimento deste tema vem da crescente participação da inteligência artificial em nosso cotidiano, e do interesse em aprender como ocorre a interação entre nós e os algoritmos nesse vasto cenário. A evolução tecnológica que tornou possível o crescimento do aprendizado de máquina, permitiu que pudéssemos deslumbrar um futuro, onde máquinas estejam cada vez mais integradas a nossa sociedade e onde a tecnologia poderá ser usada de forma positiva para tornar o mundo um lugar mais justo e acessível a todos.

Entender como acontece o processo de aprendizado de máquina, através das redes neurais, é um desejo estimulante que pode levar não só ao crescimento profissional e acadêmico como também humano pois nos coloca em profunda conexão com uma área que visa melhorar o coletivo, através da inteligência artificial.

O principal objetivo a ser apresentado neste trabalho é fazer um estudo sobre o funcionamento de uma rede neural etapa por etapa, explicando em cada ponto o que está ocorrendo e entendendo o propósito do *backpropagation* além de implementar uma rede classificadora com essa estrutura. Dentre as contribuições deste projeto, temos:

- Um estudo sobre como ocorre o processo de aprendizado de uma rede neural artificial, com explicação dos termos e nomenclaturas inerentes a proposta.
- Uma implementação de uma rede neural profunda como solução para um problema de classificação.
- Um estudo e exemplo a partir de uma base de dados real do modelo proposto.
- Uma análise sobre as diferentes aplicações possíveis de uma RNA bem como sua ligação com os campos de inteligência artificial, *machine learning* e mineração de dados.

O código utilizado neste estudo está disponível publicamente no repositório do *github*: [https://github.com/altobellibm/CEDERJ\\_2019\\_ANTONIO\\_JUNIOR](https://github.com/altobellibm/CEDERJ_2019_ANTONIO_JUNIOR)

## 2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo estão apresentados os itens e subitens objetos deste trabalho, bem como suas definições, ou, o que se tem como entendimento de seus conceitos. Estão também abordados, brevemente, eventos importantes na cronologia que nos levou até o presente momento de interesse profundo sobre *Machine Learning* e Redes Neurais.

### 2.1 APRENDIZADO DE MÁQUINA (MACHINE LEARNING)

Em uma época onde o ENIAC era o único computador disponível, e, sendo operado manualmente funcionava como uma grande calculadora, como mostrado na figura 1, Alan Turing <sup>1</sup> testava a capacidade de uma máquina aprender por meio de comunicação humana. Seus testes não surtiram os efeitos desejados, mas possibilitaram o desenvolvimento de sistemas que conseguissem.

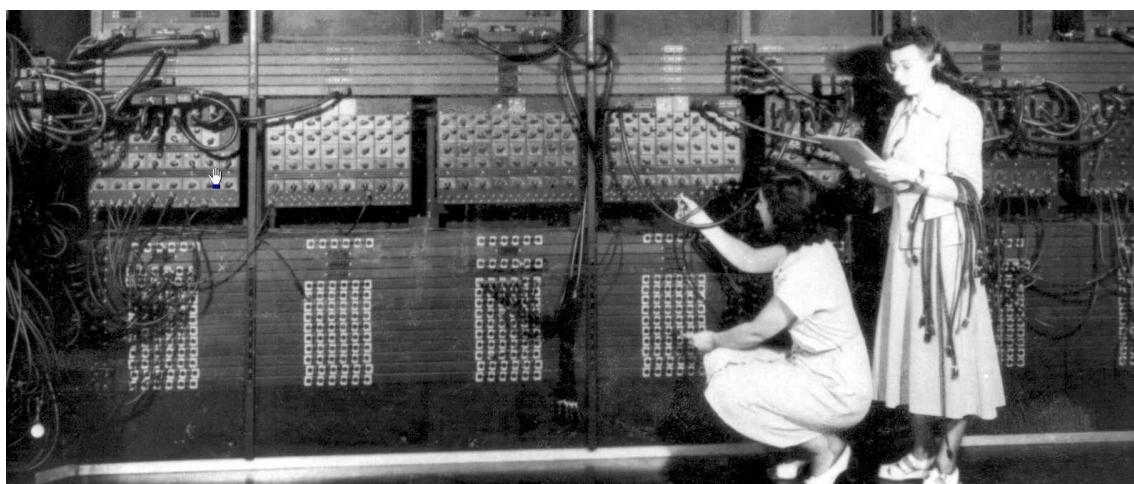


Figura 3 - Programadoras trabalhando no ENIAC

---

<sup>1</sup> **Alan Turing** (1912 – 1954): Considerado o pai da ciência da computação.

O título de precursor do Machine Learning (e até da própria Inteligência Artificial), ficou por conta de Arthur Samuel que desenvolveu o *Game of Checkers*, jogo de damas em que uma inteligência artificial conseguiu superar a estratégia humana e vencer o jogo. Arthur então definiria *Machine Learning* como 'um campo de estudos que dá ao computador a capacidade de aprender sem serem explicitamente programados para isso'.

O campo de estudos do ML hoje, foca no desenvolvimento de programas computacionais que possam acessar uma informação e usá-la para ela mesma. O processo de aprendizado começa com observações ou informações, como exemplos, experiências diretas ou instruções, visando, a detecção de padrões nessas informações, e a melhor maneira de tomar decisões no futuro baseando-se em exemplos que lhe foram providos. O objetivo dessa técnica é permitir esse aprendizado sem intervenção ou assistência humana direta, apenas pequenos ajustes muito pontuais.

### 2.1.1 ORIGEM NA INTELIGÊNCIA ARTIFICIAL

O grande sonho da Inteligência Artificial, desde o surgimento de sua ideia inicial nos idos dos anos cinquenta, era proporcionar máquinas que pudessem pensar de forma que consideramos inteligente. O conceito presente na IA impulsionou grandes produções científicas tanto no cinema como na literatura, mas é fácil perceber que estamos bem distantes ainda de convivermos com ciborgues ou andróides. Uma máquina que pensasse de forma tão inteligente quanto um ser humano, ou até mais, seria chamada de IA Forte, que segundo John Searle, "deveria ter um cérebro propriamente dito, de tal maneira que entendesse e possuísse estados cognitivos"[1].

Enquanto esse modelo não se torna realidade, convivemos habitualmente com modelos limitados de inteligência artificial (IA Fraca) que fazem tarefas específicas, como lidar com reconhecimento facial ou dirigir carros autônomos. Sistemas de inteligência artificial então, podem ser abastecidos com dados explícitos que o levam a executar uma tarefa com precisão invejável. E a técnica que uma IA utiliza é que diferenciá-la a sua classificação, uma dessas técnicas de aplicação é o *Machine Learning*.



### 2.1.2 ÁREAS DE UTILIZAÇÃO E APLICAÇÃO

Aos nos darmos conta que convivemos com IA's por toda nossa volta, passamos as vezes a nos perguntar de que forma tal aplicação foi instruída para sua funcionalidade. Assistentes virtuais, como dos sistemas operacionais dos *smartphones*, coletam e refinam dados de cada interação nossa com eles. Sistemas de tráfego, utilizam informações em tempo real do trânsito e da demanda, para calcular rotas e até preços variáveis. Recomendação de produtos gravam algumas informações de nossas buscas e interesses para oferecer algo que tenhamos maior chance de comprar.

Esses exemplos utilizam aplicações mesmo que pequenas de ML, assim como o fazem sistemas de detecção de fraudes, sistemas de busca, suporte ao usuário, serviço de controle de spam, redes sociais e sistemas de vigilância.

O campo de estudos do Aprendizado de Máquina então, está presente dentro do de Inteligência Artificial, como uma subárea, onde se torna como um método no qual se viabiliza a evolução dessa tecnologia, que necessita lidar com o crescente número de dados gerados e disponíveis para análise.

### 2.1.3 RELAÇÃO COM MINERAÇÃO DE DADOS

O grande problema desses dados, é a capacidade que nós temos, como seres humanos e limitados que somos em relação a nossa própria velocidade e racionalidade, de lidar com eles. Tornou-se necessário então, aprimorar a capacidade de instrucionalidade de programas, para que se pudesse acompanhar essa aceleração na geração de dados sem ser necessária uma intervenção humana a todo momento.

A mineração de dados aborda essa questão, na procura por padrões consistentes e na separação daquilo que é importante para o analista, daquilo que não é. Consiste em um processo que utiliza algoritmos do ML combinado com estatística para ajudar a colher informações importantes de banco de dados com uso de uma computação

de baixo custo, essa relação entre áreas é demonstrada na figura 2. A mineração é viável tanto para a pesquisa científica, como para o ramo empresarial e de serviços, e é eficiente tanto para dados estruturados, como para dados não estruturados.

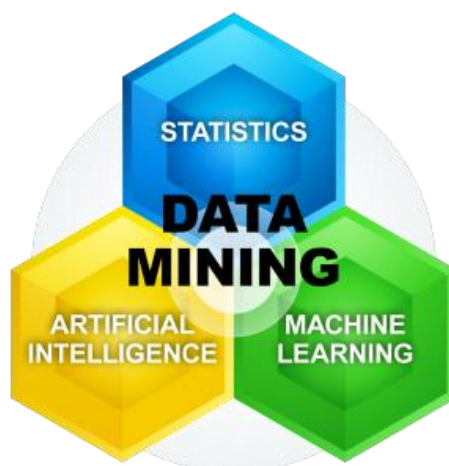


Figura 4 - Relação com Mineração de Dados

#### 2.1.4 DADOS ESTRUTURADOS E NÃO ESTRUTURADOS

Quando gerados e armazenados através de uma estrutura previamente definida para esta finalidade, os dados são considerados estruturados [2]. São fáceis de serem localizados e possuem ligações com outros dados que os relacionam a algo. Planilhas, formulários e principalmente banco de dados são exemplos de dados estruturados. Já quando não há essa organização e nem uma indexação prévia, os dados se tornam mais difíceis de serem localizados, desta forma, são definidos como não estruturados. Para este caso podem ser colocados como exemplo os arquivos de texto, imagens e vídeo em geral assim como informações de redes sociais.

Apesar das divergências no que é defendido como a relação entre estruturados e não estruturados, sabe-se que há muito mais dados não estruturados do que estruturados [3]. Até recentemente, apenas os estruturados eram utilizados em análises e buscas, mas aplicações com algoritmos de ML tornaram possível a mineração de dados não estruturados.

### 2.1.5 MÉTODOS DE APRENDIZADO

O método de aprendizado é a forma como o algoritmo vai aprender a interagir com as informações que encontrar durante sua aplicação. É considerado supervisionado, de acordo com Jason Brownlee, “quando existem variáveis de entrada e saída e há o uso de um algoritmo que aprende a função de mapeamento entre as duas.” [4, p.16]. Em um exemplo simples, Mostra-se ao programa uma foto (entrada), e diz que aquilo que corresponde a foto é um cachorro ou gato (saída). Através desse aprendizado, ele saberá ao analisar cachorros e gatos, diferencia-los. Quanto mais treinado na diferença entre cachorros e gatos, mais características ele aprenderá sobre cada um, e mais perfeito se tornará.

No caso de aprendizado não supervisionado, segundo Brownlee, ‘não há uma saída correspondente a entrada, e o objetivo é modelar a estrutura ou distribuição subjacente nos dados para saber mais sobre eles. O algoritmo é deixado por conta própria para descobrir e apresentar a estrutura interessante nos dados’. [4, p.17]. Saberá diferenciar os exemplos de entrada, mas sem saber que são gatos e cachorros. O método de aprendizado a ser escolhido depende muito do tipo de dados e do objetivo que se almeja.

## 2.2 REDES NEURAIS ARTIFICIAIS

Entre as entradas e saídas de um algoritmo voltado ao aprendizado, há um vasto caminho a ser percorrido pela informação. Se imaginarmos a entrada como um ponto A, e a saída como um ponto B, então passaremos a pensar na implementação desse caminho entre eles. Uma rede neural é uma forma de implementação, onde múltiplas camadas podem ser inseridas nesse caminho, com nós funcionando como neurônios e diferentes conexões interligando os neurônios das diversas camadas, vide figura 3 abaixo.

Um modelo primitivo, embora prático, do que viria a se transformar nos estudos de redes neurais, foi apresentado pela primeira vez em 1958 por *Frank Rosenblatt*, e foi chamado de *Perceptron* [5].

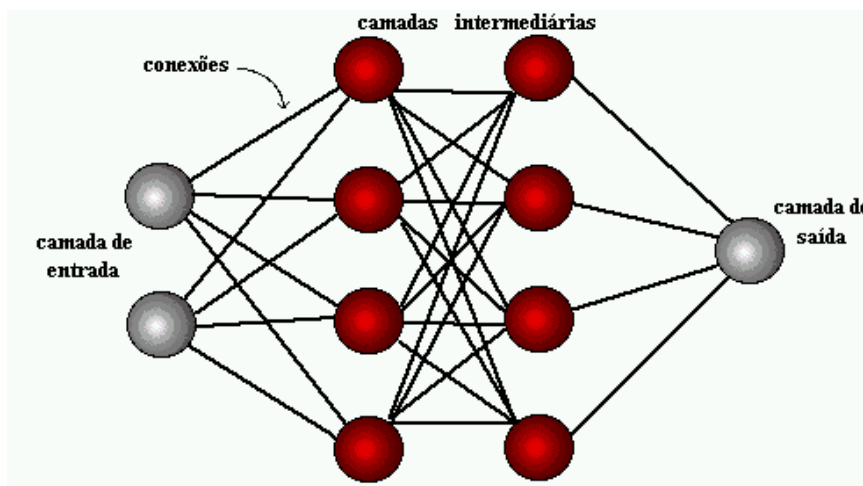


Figura 5 - Funcionamento das camadas de uma Rede Neural

### 2.2.1 ARQUITETURA E CAMADAS

Além da regra de aprendizado, o que definirá o modelo de estrutura de uma rede neural é o número de camadas e a forma como os neurônios dessas camadas se comunicam [6, p. 46]. Podemos, por exemplo, ter uma rede de camada única ou de múltiplas camadas. No primeiro modelo, os nós da entrada se conectam diretamente aos de saída, sendo que somente na camada de saída há qualquer tipo de computação, por isso chamada de única. No segundo modelo, múltiplas camadas ocultas participam do processo entre a entrada e a saída [6, p. 47].

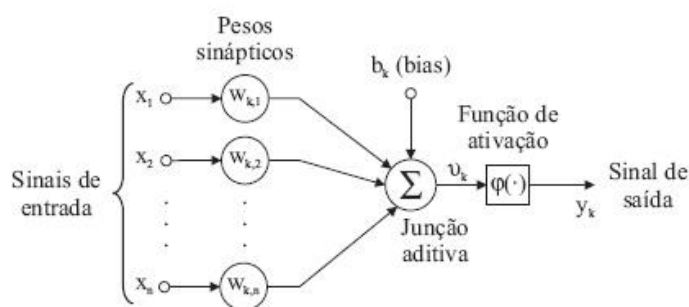


Figura 4 - Representação de um Neurônio Artificial

### 2.2.2 NEURÔNIOS E PESOS

Funcionando como pequenas unidades computacionais, os neurônios realizam a soma entre os valores que chegam até ele. Esses valores de entrada, possuem pesos, que indicam a força que aquele valor vai ter na computação do neurônio. Inicialmente esse valor geralmente é baixo, sendo adaptado conforme a rede aprende.

Como pode ser observado na figura 4, um neurônio recebe valores diversos de  $x_1$  até  $x_n$  sendo  $n$  o número de entradas, com pesos sinápticos  $w_{k1}$  até  $w_{kn}$  respectivos aos sinais. Podemos observar também, na equação abaixo, a representação matemática de um neurônio.

$$v_k = \sum_{i=1}^N w_{ki} x_i + b_k$$

Equação 1 – Modelo de Neurônio

### 2.2.3 BIAS

Observando a figura quatro, vemos que a função de soma calcula seu valor aplicado a um bias  $b_k$  observando sua função de ativação  $\varphi$  para definir seu sinal de saída  $y_k$ . Em redes de múltiplas camadas, os neurônios intermediários, possuem uma constante atrelada a sua função de ativação, que recebe o nome de Bias [7, cap. 4]. Se os pesos funcionam em razão dos valores de entrada em um neurônio, a Bias é referente ao próprio Neurônio, conforme podemos ver na equação acima.

No caso de todas as entradas de um neurônio qualquer em uma camada qualquer tiverem valor nulo, independentemente do valor dos pesos, sem uma Bias, a função soma no neurônio será nula. Com a Bias, é adicionado um valor após a soma, que ajuda a aproximar o valor resultante do desejado [7, cap. 4]. É extremamente necessário em redes que não funcionam com propagação direta, e, portanto, necessitam de correções que são implementadas com base no erro do resultado obtido em relação ao resultado esperado.

### 2.2.4 FUNÇÃO DE ATIVAÇÃO

As funções de ativação servem para inserir uma não linearidade na rede. Elas funcionam com base no valor de saída de um neurônio, como que decidindo o comportamento daquela saída com base no definido na função.

No modelo mais simples, o binário, a função de ativação simplesmente decide se aquele neurônio vai ser ativado ou não. Ela verifica, por exemplo, se o valor de saída de um neurônio é maior que um determinado valor  $x$ , e, caso positivo, ativa o nó de forma que seu valor prossiga na rede como um dos valores de entrada de um ou mais nós da camada seguinte. No caso negativo, o neurônio é considerado desimportante e seu valor é ignorado [8, cap. 8].

Entretanto, esta função de ativação não é efetiva em redes com o mínimo de complexidade, de forma que se torna mais útil a escolha de um tipo de ativação de categoria não linear, como Sigmóide, Tanh ou ReLU.

Conforme demonstrado na figura 5, uma função linear simples define  $f(x)$  como o próprio  $x$ . Nesse caso, o *output* do neurônio não estaria limitado em um arco com um valor máximo e um mínimo, podendo propagar um valor qualquer desde menos infinito a mais infinito.

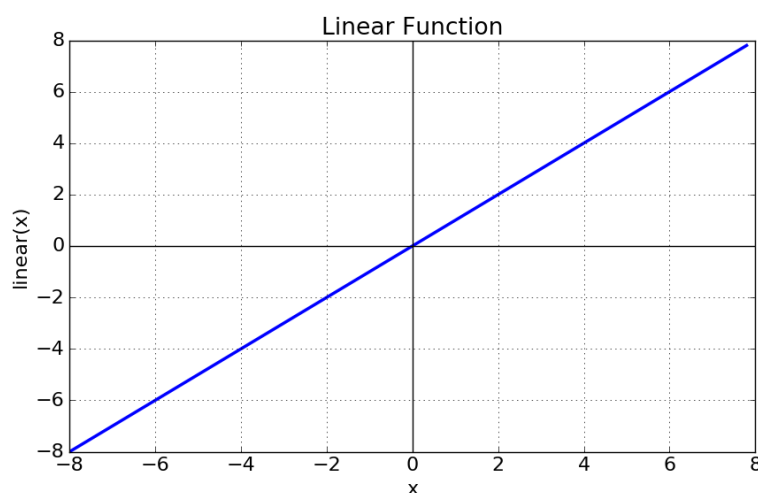


Figura 5 – Exemplo de Função de Ativação Linear

Uma função não linear, propaga um valor dentro de uma delimitação, dependendo daquilo recebido do neurônio. Sua representação foge da forma de linha e suas subclassificações são feitas com base na curva formada. Um exemplo de função de ativação não linear pode ser visto na figura 6.

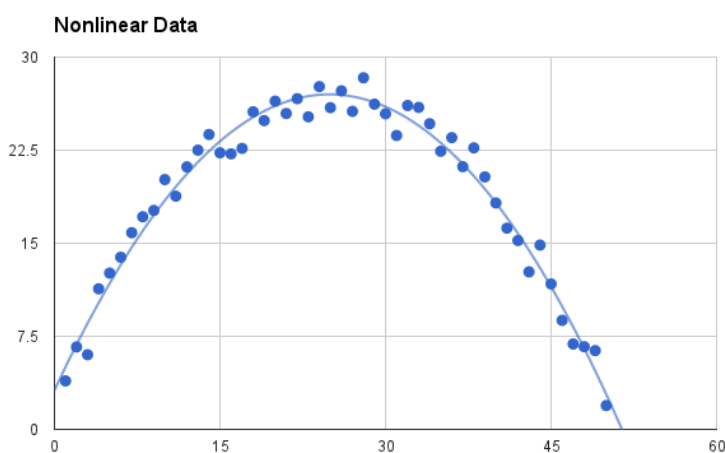


Figura 6 – Exemplo de representação de Função de Ativação não linear

## 2.3 BACKPROPAGATION (RETRO PROPAGAÇÃO)

Entre as formas de treinamento por aprendizado supervisionado de uma rede neural, o uso do algoritmo de *Backpropagation* é o mais comum. Sua eficácia consiste nas repetitivas iterações dentro da rede a partir do erro no resultado final em relação ao que se esperava.

Corrige-se os valores dos pesos desde a última camada em direção a primeira, visando diminuir o erro. A cada correção (iteração), o resultado obtido pela rede pode sofrer uma pequena alteração, se aproximando, assim, do desejado. Para a correta aplicação desse algoritmo, são esperados pares  $(x_i, y_i)$  de entradas da rede e suas

consecutivas desejadas saídas. Também é esperado que a rede seja do tipo *feed-forward*<sup>2</sup> e que possua uma função de erro.

### 2.3.1 INICIANDO UMA REDE

Podemos iniciar uma rede de teste, como da figura 7, usando as seguintes configurações:

- Entradas (*inputs*) **i1** e **i2** com valores 0.05 e 0.10 respectivamente.
- Pesos (*weights*) variados conectando neurônios de camadas distintas.
- Camadas ocultas (*hidden*) **h1** e **h2** com valores a serem definidos.
- Camadas de saída (*outputs*) **o1** e **o2** com esperados valores 0.01 e 0.99.
- Bias **b1** de valor 0.35 e **b2** de valor 0.60 correspondendo respectivamente a camada oculta e a camada de saída.

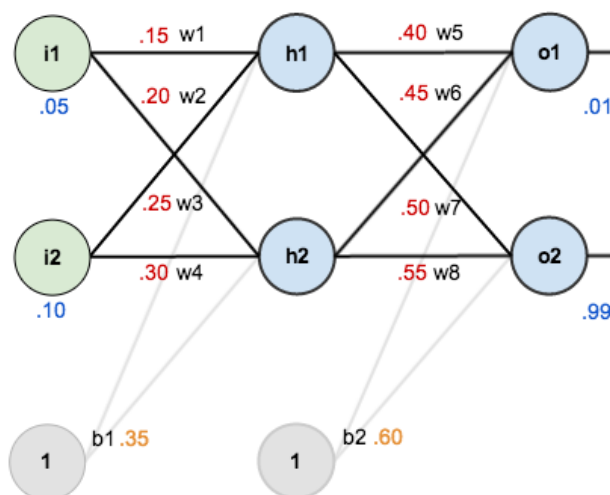


Figura 7 – Exemplo de valores iniciados em uma RNA.

<sup>2</sup> Uma rede onde todos os neurônios de cada camada se conectam com todos os neurônios da camada posterior, mas não ocorrem conexões entre neurônios de uma mesma camada.



Podemos passar a calcular o valor de entrada nos neurônios ocultos da rede, observando as conexões e os valores referentes a elas que afetam cada um. O neurônio **h1** receberá o valor de entrada **i1** multiplicado pelo peso **w1**, somando com o valor de entrada **i2** multiplicado pelo peso **w3** e somando esse resultado ainda com o bias da camada 1 (**b1**). O mesmo raciocínio usaremos para o neurônio h2. Assim, definindo:

$$somh_1 = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$somh_1 = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

$$somh_2 = w_3 * i_1 + w_4 * i_2 + b_1 * 1$$

$$somh_2 = 0.25 * 0.05 + 0.3 * 0.1 + 0.35 * 1 = 0.3925$$

### 2.3.2 USANDO ATIVAÇÃO LOGÍSTICA

Nesta rede, é utilizada como função de ativação dos neurônios, a função logística (ou função Sigmoide). Isto é, a função existirá dentro de um intervalo compreendido entre 0 e 1, é não-linear e diferenciável. Sua representação gráfica e sua fórmula<sup>3</sup> matemática podem ser observadas na figura 8. Para mais exemplos de funções de ativação, verifique o Anexo A.

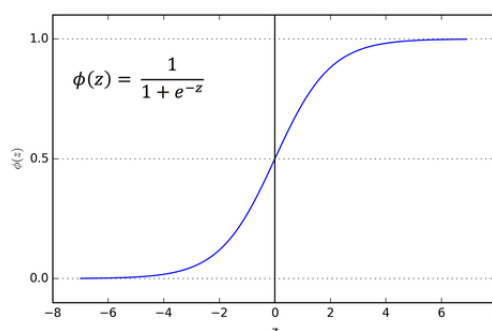


Figura 8 – Fórmula e representação da função Logística.

---

<sup>3</sup> O “e” presente na fórmula refere-se a constante matemática, também conhecida como número de Euler cujo valor aproximado é 2,718.

Aplicando a fórmula da função de ativação com a saída do neurônio intermediário  $h_1$ , teremos  $saidah_1 = \frac{1}{1+e^{-somh_1}} \rightarrow \frac{1}{1+e^{-0.3775}} = 0.593269992$ .

Aplicando o mesmo em relação a  $h_2$ , temos  $saidah_2 = 0.596884378$ .

O processo se repete em relação aos neurônios da camada de saída, agora as entradas serão justamente os valores calculados de saída da camada anterior. Assim, definindo:

$$somo_1 = w_5 * saidah_1 + w_6 * saidah_2 + b_2 * 1$$

$$somo_1 = 0.40 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.1050905967$$

$$somo_2 = w_7 * saidah_1 + w_8 * saidah_2 + b_2 * 1$$

$$somo_2 = 0.50 * 0.593269992 + 0.55 * 0.596884378 + 0.6 * 1 = 1.2249213679$$

Repetindo o processo de encontrar o valor de saída (*output*) com a função de ativação, teremos:  $saidao_1 = 0.75136507$  e  $saidao_2 = 0.772928465$ .

### 2.3.3 CÁLCULO DO ERRO

Para os ajustes serem feitos, iteração após iteração, calcula-se então o valor desse erro obtido. Esse cálculo pode ser feito de forma diferente dependendo do tipo de rede, mas para nosso caso aqui utilizaremos uma fórmula conhecida na estatística como erro quadrático médio (MSE), onde para cada saída, tira-se a diferença entre o resultado verdadeiro e o resultado obtido, elevando ao quadrado. Repete-se o processo para todo par de saída da rede/saída desejada.

Conforme definido acima, a função de erro (também chamada de função de custo) poderia ser representada como:

$$E_{TOTAL} = \sum \frac{1}{2} (t - o)^2$$

Equação 2 – Função de erro (ou custo)

Ou seja, o erro total sendo igual a soma da metade do *target* (saída esperada) menos o *output* (saída encontrada) ao quadrado. A fração  $\frac{1}{2}$  que divide o resultado

entre parênteses ao meio, serve para cancelar expoentes futuramente e facilitar o processo. Essa inserção não interfere na equação, já que é uma constante e o resultado da fórmula será multiplicado por uma taxa de aprendizado futuramente.

Para  $saidao_1$  e  $saidao_2$ , temos então respectivamente:

$$E_{o1} = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$

$$E_{o2} = \frac{1}{2}(0.99 - 0.772928465)^2 = 0.023560026$$

E para o erro total da rede, fazemos:

$$E_{TOTAL} = E_{o1} + E_{o2}$$

$$E_{TOTAL} = 0.298371109$$

#### 2.3.4 O GRADIENTE E A APLICAÇÃO DA REGRA DA CADEIA

Conforme dito no item 2.3, precisamos adaptar o valor dos pesos para diminuir o valor da perda/erro. Isso é feito para cada neurônio, de trás para frente. Em um primeiro momento, se calcula então o erro dos pesos das conexões da camada de saída, ou seja, de  $w_5$ ,  $w_6$ ,  $w_7$ , e  $w_8$ .

Para tal, precisamos por exemplo saber o tanto que uma mudança no valor de  $w_5$  afeta o valor de erro total. Representamos isso como:

$$\frac{\partial E_{total}}{\partial w_i}$$

Equação 3 – Gradiente em relação a  $w_i$

Podemos interpretar a equação 3, como a derivada parcial do erro total em relação a  $w_5$ , ou, simplesmente, o gradiente em relação a  $w_5$ . Aplicamos aqui nesta fórmula, a regra da cadeia<sup>4</sup>, conforme ilustrado na figura 9.

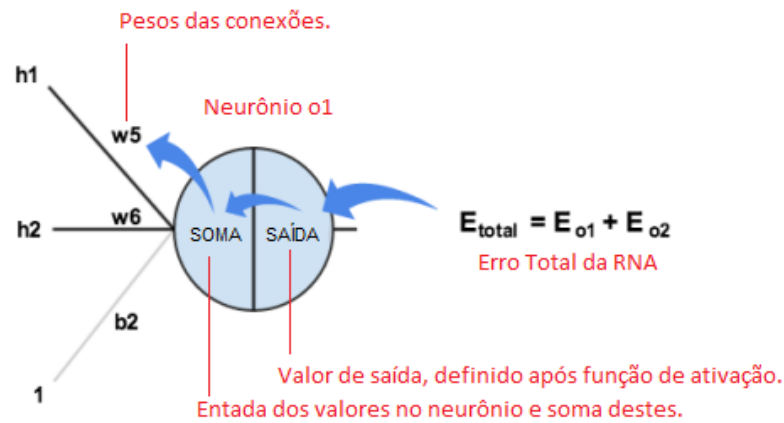


Figura 9 – Ilustração da regra da cadeia no nó da camada de saída.

Como temos que encontrar o gradiente do erro total em relação a  $w_5$ , desdobramos a equação de forma que este gradiente seja igual ao gradiente do erro total em relação a saída de  $o_1$ , multiplicado pelo gradiente da saída de  $o_1$  em relação a soma calculada em  $o_1$ , multiplicado ainda pelo gradiente desta soma em relação ao peso  $w_5$ . Ou seja:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial_{saida o_1}} * \frac{\partial_{saida o_1}}{\partial_{soma o_1}} * \frac{\partial_{soma o_1}}{\partial w_5}$$

### 2.3.5 O GRADIENTE DO ERRO TOTAL EM RELAÇÃO A SAÍDA

Para facilitar a visão geral da equação, dividiremos em três partes a busca da solução, e começaremos pela primeira que é o gradiente do erro total em relação a saída  $o_1$ .

<sup>4</sup> Fórmula desenvolvida por Gottfried Leibniz para a derivada de uma função composta de duas funções.

Como vimos,  $E_{TOTAL} = E_{o1} + E_{o2}$ , entretanto, como estamos calculando em relação a  $o_1$ , o valor de  $E_{o2}$  não terá relevância ou influência, se tratanto de um nó sem ligação com  $o_1$ , então será interpretado como constante pela equação, e sua derivada, será zero. Definimos então a primeira parte como:

$$\frac{\partial E_{total}}{\partial saidao_1} = 2 * \frac{1}{2} (esperado o_1 - encontrado o_1)^{2-1} * -1 + 0$$

A saída esperada, era de 0,01, a encontrada foi de 0,75136507. O zero na equação, corresponde a derivada de  $E_{o2}$  e a equação é multiplicada por menos um para seu resultado ser positivo, caso precise. O resultado será de  $-(0,01 - 0,75136507)$  que resultará em **0,74136507**.

### 2.3.6 O GRADIENTE DA SAÍDA EM RELAÇÃO A SOMA

Nessa parte, encontraremos o quanto a saída de  $o_1$  muda em relação a sua soma. E calcular a derivada da saída  $o_1$  implica em calcular a derivada de sua função de ativação logística, que pela figura 8, sabemos ser  $1 / 1 + e^{-soma_{o1}}$ . Através da definição de distribuição logística<sup>5</sup>, a derivada parcial da função logística é a saída (sua função) multiplicado por um menos sua saída. Então sua definição pode ser simplificada da seguinte forma:

$$\frac{\partial saidao_1}{\partial somo_1} = saidao_1 (1 - saidao_1) = 0,75136507 (1 - 0,75136507) = \mathbf{0,186815602}$$

### 2.3.7 O GRADIENTE DA SOMA EM RELAÇÃO AO PESO

---

<sup>5</sup> Um exemplo de distribuição de probabilidade contínua. Utilizada nos campos da teoria das probabilidades e estatística.

Na terceira parte da equação, lembrando como se calcula o valor de entrada em um neurônio, igual visto no capítulo 2.3.2, temos:

$$somo_1 = w_5 * saidah_1 + w_6 * saidah_2 + b_2 * 1$$

Como estamos lidando com a derivada da soma em relação a  $w_5$ , o valor da bias e o valor do peso  $w_6$  não terão influência, e a exemplo do exposto no início do item 2.3.5, seus valores serão zero. Desta forma, para o gradiente desta soma em relação a  $w_5$ , definimos:

$$\frac{\partial somo_1}{\partial w_5} = 1 * saída h_1 * w_5^{1-1} + 0 + 0 = saída h_1 = \mathbf{0.593269992}$$

Colocando todos os três pedaços da equação juntos, temos o resultado obtido após aplicação do conceito da regra da cadeia, do gradiente do erro total em relação ao peso  $w_5$ :

$$\frac{\partial Etotal}{\partial w_5} = 0.74136507 * 0.186815602 * 0.593269992 = \mathbf{0.082167041}$$

### 2.3.8 TAXA DE APRENDIZADO

Para diminuição do erro, devemos subtrair do valor de do peso  $w_5$  o valor encontrado. Adicionalmente, podemos multiplicar antes este valor por uma constante denominada taxa de aprendizado. Ela é opcional e serve para delimitar o impacto desse ajuste nos pesos em relação ao gradiente de perda [9]. Seguindo nosso exemplo, delimitaremos a nossa taxa de aprendizado ( $\alpha$ ) em 0.5. Sendo assim, definimos nosso peso ajustado  $w_5^+$ :

$$w_5^+ = w_5 - \alpha * \frac{\partial Etotal}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = \mathbf{0.35891648}$$

E repetindo todo o processo em relação a  $w_6$ ,  $w_7$  e  $w_8$ , encontraremos:

$$w_6^+ = 0.408666186$$

$$w_7^+ = 0.511301270$$

$$w_8^+ = 0.561370121$$

Entretanto, essa atualização de pesos se dará após o término do processo, fato que inclui a atualização também dos pesos que conectam a camada oculta com a camada de entrada. Para o *backpropagation* desses pesos ( $w_1$  a  $w_4$ ), toda vez que um dos pesos cujo ajuste já tenha sido feito ( $w_5$  a  $w_8$ ) seja referenciado, o valor a ser utilizado como referência é o original e não o ajustado. Em verdade, todos os pesos da rede são ajustados simultaneamente, após a conclusão da atualização.

Para esta etapa dos pesos  $w_1$  a  $w_4$ , o processo é similar, entretanto, leva-se em conta que a saída dos pesos da camada oculta “ $h$ ” influenciará nos valores de entrada de todos os pesos da camada de saída “ $o$ ”, portanto, na etapa do cálculo do gradiente do erro, não haverá um componente iniciado de valor zero, já que todos influenciarão no objeto calculado.

Após a repetição do processo com a devida ressalva para os pesos em questão, os resultados obtidos serão:

$$w_1^+ = 0.149780716$$

$$w_2^+ = 0.19956143$$

$$w_3^+ = 0.24975114$$

$$w_4^+ = 0.29950229$$

Com o ajuste dos pesos, estes valores passam a substituir os originais, e alimentando a rede a partir dos valores de entrada (0.05 e 0.10), o erro calculado da rede passa de 0.298371109 para 0.291027924. O processo passa a se repetir com repetidas iterações visando sempre buscar o menor valor de erro em relação aos *outputs* esperados. Após dez mil execuções, o valor do erro da rede será de 0.0000351085 e as saídas serão 0.015912196 (para o desejado valor de 0.01) e 0.984065734 (para o desejado valor de 0.99).

### 3 IMPLEMENTAÇÃO

Aqui será implementada uma rede, utilizando backpropagation, cujo objetivo será classificar uma espécie de planta. O algoritmo será implementado utilizando a linguagem *Python* [10], utilizando máquina virtual cedida a este propósito pelo *Google*, através da ferramenta conhecida como *Google Colab* [11], será utilizada um banco público de dados, o repositório de aprendizado de máquina UCI [12]. Serão importados, no código, funções pertencentes ao *Keras* [13], uma biblioteca de código aberto escrita em *Python* e voltada a redes neurais. Serão utilizadas também as bibliotecas *Pandas* [14] e *scikit-learn* [15], também escritas em *Python*, sendo esta última comumente utilizada em algoritmos de classificação.

#### 3.1 PROPOSTA

A implementação visa demonstrar, na prática, como funciona uma rede classificadora. O caso específico aqui, é um algoritmo que vai ser treinado a partir de um conjunto de dados que compõem as características físicas de uma planta da espécie *Íris*. São informadas para a rede o comprimento em centímetros da sépala, a largura em centímetros da sépala, o comprimento em centímetros da pétala e a largura em centímetros da pétala. Esses dados são acompanhados da classificação da planta, que são três: *Íris-setosa*, *Íris-versicolor* e *Íris virginica*.

Após o treino com mais de uma centena de casos, o teste de mais alguns casos visa apurar se o algoritmo consegue prever corretamente a qual dos tipos uma espécime pertence com suas características informadas.

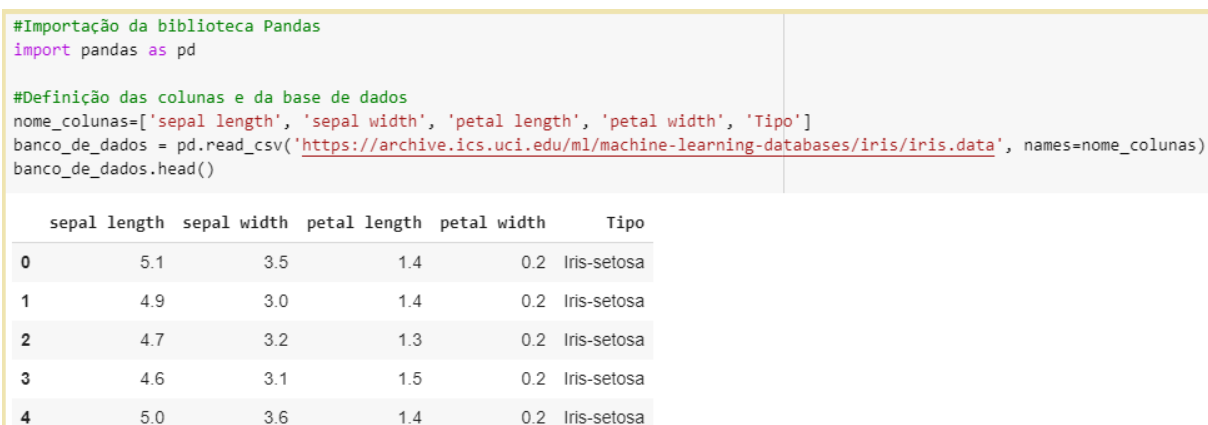
Esse conjunto de dados é comum em artigos e trabalhos estatísticos e foi introduzido pelo estatístico e biólogo inglês Ronald Fischer em 1936. É conhecido como *Iris flower data set* ou *Fisher's Iris data set* [16]. É composto de dados de 50 espécimes de cada um dos tipos de *Íris* citados acima.



## 3.2 CONFIGURANDO A REDE

Fazendo uso da ferramenta *Google Colab*, configuramos a máquina virtual para fazer seu processamento através da *GPU*. No *colab*, precisamos salvar nosso código em fonte externa, pois, apesar da gratuidade da ferramenta, a máquina fica disponível por apenas doze horas, sendo necessário importa-la da fonte externa quando do término desse período.

No início da codificação, importamos a biblioteca *Pandas* e o conjunto de dados e definimos os nomes das colunas, que não constavam inicialmente no conjunto de dados. Executamos o comando *head* apenas como caráter informativo, para demonstrar de que forma os dados estão dispostos. Esse comando, quando sem parametro informado, imprime na tela as cinco primeiras linhas, conforme mostrado na imagem abaixo.



```
#Importação da biblioteca Pandas
import pandas as pd

#Definição das colunas e da base de dados
nome_colunas=['sepal length', 'sepal width', 'petal length', 'petal width', 'Tipo']
banco_de_dados = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data', names=nome_colunas)
banco_de_dados.head()
```

	sepal length	sepal width	petal length	petal width	Tipo
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Figura 10 – Código: Colunas e impressão por comando *head*

A seguir, definimos como saída apenas a coluna do tipo correspondente a amostra em questão, e definimos valores para os tipos, passando três células e não mais uma, como visto na figura abaixo. Isso se deve devido ao fato do algoritmo trabalhar com números, e os dados da coluna estarem na forma de *string*. Então iremos trabalhar com a probabilidade de um conjunto de dados (linha) corresponder a um tipo.

```
#Definindo a saída como a coluna Tipo
saida = banco_de_dados[['Tipo']]

#Separando a saída em três colunas a serem preenchidas de forma booleana
saida = pd.get_dummies(saida, columns=['Tipo'])
values = list(saida.columns.values)

saida.head()
```

Figura 11 – Código: Mapeamento da saída como colunas

Na sequência, trabalhamos com a definição da rede como o nosso banco de dados, com a retirada da coluna referente a saída, mantendo apenas os dados de entrada. Fazemos também a importação da função *split*, da biblioteca *sklearn* conforme demonstrado na figura 12.

Com essa função, passamos a rede e a saída como parametros junto com um valor definido para *test\_size*. No caso, 0.2, ou seja, vinte por cento. A função interpretará que vinte por cento dos dados devem ser deixados para o teste do algoritmo, enquanto que os oitenta por cento restantes sejam de treino. Essa divisão é aplicada tanto aos dados de entrada como aos de saída correspondentes e de forma randômica sem perda de indexação entre eles.

```
#Retirada da coluna de saída do grupo de dados da entrada
rede = banco_de_dados.drop(columns=['Tipo'])

#Importação de função e divisão entre treino e teste
from sklearn.model_selection import train_test_split

rede_treino, rede_teste, saida_treino, saida_teste = train_test_split(rede, saida, test_size=0.2)

print("\nRede de Treino (sem a saída):\n")
print(rede_treino.head())
print(rede_treino.shape)

print("\nRede de Teste (sem a saída):\n")
print(rede_teste.head())
print(rede_teste.shape)
```

Figura 12 – Código: Função Split aplicada a treino e teste

Para efeito de demonstração e de verificação efetiva da programação até o momento, é printado na tela tanto o cabeçalho da rede de treino, como da rede de teste, junto com o comando *shape*.

Este serve para informar a forma das tabelas, mostrando o número de linhas e o número de colunas. Como pode ser observado na figura 13, em um exemplo de treino, a rede começa com as linhas de *index* 31, 39, 37, 61 e 66 e suas características físicas dos exemplares estão informados nas colunas ao lado. Abaixo, sua dimensão de 120 linhas por 4 colunas.

O mesmo ocorre com a rede de testes, que ficou com 30 linhas e 4 colunas, que correspondente a vinte por cento da tabela total de entradas.

Rede de Treino (sem a saída):				
	sepal length	sepal width	petal length	petal width
31	5.4	3.4	1.5	0.4
39	5.1	3.4	1.5	0.2
37	4.9	3.1	1.5	0.1
61	5.9	3.0	4.2	1.5
66	5.6	3.0	4.5	1.5
(120, 4)				
Rede de Teste (sem a saída):				
	sepal length	sepal width	petal length	petal width
46	5.1	3.8	1.6	0.2
22	4.6	3.6	1.0	0.2
35	5.0	3.2	1.2	0.2
63	6.1	2.9	4.7	1.4
90	5.5	2.6	4.4	1.2
(30, 4)				

Figura 13 – Código: Exemplo de impressão para verificação de treino e teste

### 3.3 MODELAGEM E COMPILAÇÃO

Neste etapa, definiremos o nosso modelo. Importamos do *Keras* o tipo *Sequencial*<sup>6</sup> para aplicar a rede, como também o fazemos com o tipo *Dense* para aplicar as camadas. Passamos a dimensão da rede, em colunas, como o número de colunas da

---

<sup>6</sup> É um modelo do *Keras* composto por uma pilha linear de camadas. É criado declarando instâncias da camada para o construtor.

camada de entrada. Criamos duas camadas ocultas e uma de saída e aplicamos o modo de ativação *ReLU* para todas elas.

Como visto na figura 14 abaixo, definimos dez neurônios para a camada de entrada, vinte para a camada seguinte e dez para a terceira camada. Para a camada de saída, são definidos três neurônios, nesse caso correspondentes ao número de tipos de plantas possíveis.

Quanto ao número de neurônios das camadas intermediárias e de entrada, a escolha se deu após rodadas de treino, mediante a apresentação de um melhor resultado de predição.

```
#Importação do modelo Sequencial e da forma densa para camadas
from keras.models import Sequential
from keras.layers import Dense

#Criação do modelo da rede
model = Sequential()

#Definição de variável armazenando o número de colunas
numero_colunas = rede.shape[1]

#Adicionando as camadas
model.add(Dense(10, activation='relu', input_shape=(numero_colunas,)))
model.add(Dense(20, activation='relu'))
model.add(Dense(10, activation='relu'))
model.add(Dense(3))

#Compilação do modelo e definição da otimização e função de perda
model.compile(optimizer='adam', loss='mean_squared_error')
```

Figura 14 – Código: Modelagem das camadas da Rede Neural

Na última linha do código na figura acima, há a definição do otimizador *Adam* [17], que é um algoritmo para a taxa de aprendizado, bem como há a definição do erro quadrático médio para a função de perda da rede.

A próxima etapa é treinar o modelo, e para ajudar nisso importamos do *keras* a função *EarlyStopping*. Ela serve para regularizar o treino diminuindo a incidência de sobreajuste (*overfitting*), que ocorre quando um modelo, apesar de se ajustar bem a um conjunto de dados, se mostra ineficiente com novos grupos de informações.

Criamos o nosso modelo com o nome teste e passamos pra ele a rede de treino e a saída de treino, tendo a rede e a saída de teste como fatores de validação. Informamos o número de interações (*epochs*) como sendo cem, mas deixamos comentada a parte do código referente a função de regularização, para usarmos caso seja uma necessidade.

Na imagem abaixo, além do código, podemos ver que o monitor vai atualizando os valores de perda e perda na validação, a medida que as interações vão avançando.

```
#Interrompe as interações quando a rede deixa de se aprimorar
from keras.callbacks import EarlyStopping
early_stopping_monitor = EarlyStopping(patience=3)

#Treino do modelo
teste = model.fit(rede_treino, saida_treino, validation_data=(rede_teste, saida_teste), epochs=100)
#callbacks=[early_stopping_monitor])
```

```
Epoch 72/100
120/120 [=====] - 0s 193us/step - loss: 0.0537 - val_loss: 0.0784
Epoch 73/100
120/120 [=====] - 0s 136us/step - loss: 0.0529 - val_loss: 0.0774
Epoch 74/100
120/120 [=====] - 0s 134us/step - loss: 0.0519 - val_loss: 0.0770
Epoch 75/100
120/120 [=====] - 0s 161us/step - loss: 0.0510 - val_loss: 0.0767
Epoch 76/100
120/120 [=====] - 0s 136us/step - loss: 0.0507 - val_loss: 0.0750
Epoch 77/100
120/120 [=====] - 0s 147us/step - loss: 0.0499 - val_loss: 0.0753
```

Figura 15 – Código: Treino do modelo da rede

Para visualizarmos as informações referentes a perda no treino e perda na validação mais facilmente, importamos uma função da biblioteca *matplotlib* [18] do *Python*, que ajuda com a criação de figuras em duas dimensões de natureza matemática.

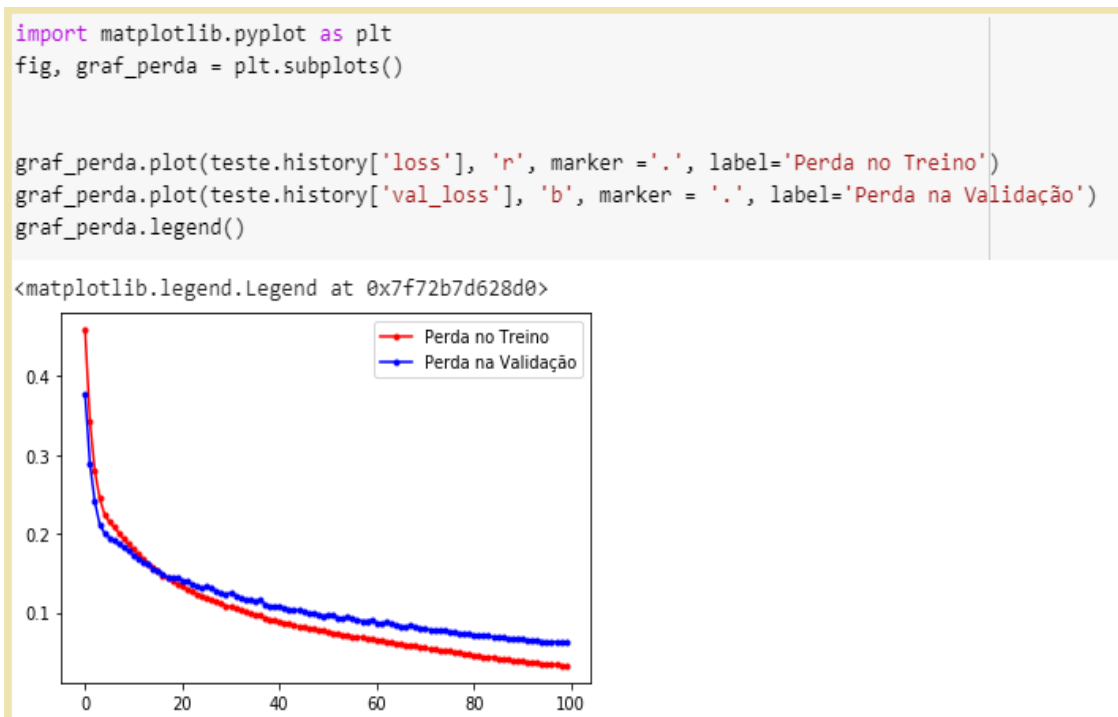


Figura 16 – Código: Comparação de curvas referentes a perda

A ela, aplicamos duas curvas, uma referente a perda no treino, que definimos com a letra *r* como sendo de cor vermelha (*red*), e outra referente a perda na validação, com a cor definida como azul (parâmetro *b* refere-se a *blue*). O resultado pode ser visto na imagem 16 acima.

De acordo com o que vimos no item 2.3 sobre perda (erro e cálculo do erro), observamos na imagem que a cada iteração o valor de perda cai, tanto para o treino como para o teste da rede, e de forma coordenada uma com a outra, indicação que o modelo não está viciado com os casos de treino e que também pode ser aplicado a outros casos, os de teste, já que o padrão de aprimoramento se manteve bem próximo.

Para fins de curiosidade, podemos olhar para a forma como o algoritmo faz a sua predição. Na figura 17, pedimos pela impressão das predições do algoritmo sobre os cinco primeiros exemplares de plantas da rede de treino, e logo abaixo, dos mesmos cinco exemplares correspondentes na saída de treino. Lembrando que os casos do banco de dados de plantas do tipo Íris foram embaralhados quando do uso da função *split*.

```
model.predict(rede_treino.iloc[0:5])

array([[ 1.0075073, -0.06878446,  0.05249228],
       [ 0.9541365,  0.13368884, -0.09255233],
       [ 1.1954409, -0.19061868,  0.13099577],
       [ 0.15782398,  0.8029494, -0.11576903],
       [ 0.02559779,  0.14227344,  0.8481591 ]], dtype=float32)

saida_treino.head(5)
```

	Tipo_Iris-setosa	Tipo_Iris-versicolor	Tipo_Iris-virginica
<b>4</b>	1	0	0
<b>23</b>	1	0	0
<b>15</b>	1	0	0
<b>98</b>	0	1	0
<b>113</b>	0	0	1

Figura 17 – Código: Comparação da função *Predict* com valores reais

Na execução da função *predict*, o algoritmo retornou valores flutuantes para a probabilidade em relação a ser cada um dos três tipos.

Na próxima etapa, ele vai apontar aquele tipo onde a probabilidade mais se aproximar dos cem por cento (valor 1.0).

Foi sublinhado em cada caso o valor onde há a maior probabilidade, para compararmos com a saída real.

Aqui, em valores aproximados, ele apontou cem por cento para o tipo *setosa* no *index* 4, noventa e cinco por cento também para *setosa* no *index* 23 e bem mais de cem por cento para o mesmo tipo no *index* 15.

Também calculou em oitenta por cento para o tipo *versicolor* no *index* 98 e, logo abaixo, oitenta e quatro por cento para o *virginica*.

Na saída real, pelo comando *head*, vemos que acertará os cinco casos quando for fazer a predição através da função específica.

Agora que vimos o funcionamento do algoritmo em suas etapas intermediárias, chegamos a parte final, onde teremos apontado o tipo de planta para cada entrada a exemplo do explicado acima com base na probabilidade de cada caso.

### 3.4 ACURÁCIA DA REDE

Fazendo uso da biblioteca *sklearn*, importamos a função da matriz de confusão, para, nos casos de erro, podermos ver a direção para qual resposta o algoritmo errou. Essa matriz é bem útil quando os neurônios da camada de saída são muitos, mas também nos serve nesse caso de três possibilidades.

Já no caso da biblioteca *numpy*, vamos usar a função *zeros* que retorna um vetor “valor\_pred” de predições preenchido com zeros. Aplicamos o *predict* em um vetor temporário e criamos um vetor “valor\_real” que será preenchido com o índice do campo onde estar o maior valor (no caso, o número um *booleano* correspondente ao tipo de planta) através da função *argmax*.

Finalmente, preenchemos cada *i* do vetor de zeros de predições com o índice do campo de maior valor do *i* do vetor temporário. Imprimimos a matriz de confusão confrontando os valores reais com os valores de predições.

Observando a figura 18, vemos que, na diagonal, temos os casos onde a saída de acordo com a rede foi igual a saída real. Foram quarenta e dois acertos para o tipo *setosa*, quarenta acertos para o *versicolor* e trinta e seis para o tipo *virginica*, tivemos ainda dois erros, totalizando os 120 casos de treino.

```
#Importação da matriz de confusão
import numpy as np
from sklearn.metrics import confusion_matrix

valor_pred = np.zeros(rede_treino.shape[0])

#Fazendo a predição da rede
predicao = model.predict(rede_treino)
valor_real = np.array(saida_treino).argmax(axis=1)

for i in range(rede_treino.shape[0]):
    valor_pred[i] = predicao[i].argmax()

#Imprimindo a matriz
confusion_matrix(valor_real, valor_pred)

array([[42,  0,  0],
       [ 0, 40,  1],
       [ 0,  1, 36]])

#Calculando a acurácia do algoritmo
from sklearn.metrics import accuracy_score
acuracia = (accuracy_score(valor_real, valor_pred) * 100)

print ("A acurácia dos casos de treino foi de %d" % (acuracia), "%.")

A acurácia dos casos de treino foi de 98 %.
```

Figura 18 – Código: Predição dos casos de treino e acurácia do algoritmo



Com a função `score` usada nas linhas finais do código acima, definimos a acurácia entre os valores da rede e os reais, e multiplicamos por cem para impressão bem formatada, e o retornado foi **98%** de sucesso.

Repetindo os passos realizados nesse item, apenas alterando os vetores para trabalharem em relação aos casos de teste, podemos conferir os resultados em relação aos trinta casos. A figura 19, abaixo, ilustra os retornos obtidos, sendo oito acertos para o tipo *setosa*, cinco acertos para o *versicolor*, e treze acertos para o *virginica*. Somam-se quatro casos de erro, apontando uma acurácia de **86%**.

```
#Importação da matriz de confusão
import numpy as np
from sklearn.metrics import confusion_matrix

valor_pred = np.zeros(rede_teste.shape[0])

#Fazendo a predição da rede
predicao = model.predict(rede_teste)
valor_real = np.array(saida_teste).argmax(axis=1)

for i in range(rede_teste.shape[0]):
    valor_pred[i] = predicao[i].argmax()

#Imprimindo a matriz
confusion_matrix(valor_real, valor_pred)

array([[ 8,  0,  0],
       [ 0,  5,  4],
       [ 0,  0, 13]])

#Calculando a acurácia do algoritmo
from sklearn.metrics import accuracy_score
acuracia = (accuracy_score(valor_real, valor_pred) * 100)

print ("A acurácia dos casos de teste foi de %d" % (acuracia), "%.")

A acurácia dos casos de teste foi de 86 %.
```

Figura 19 – Código: Predição dos casos de teste e acurácia do algoritmo

### 3.5 TESTANDO OUTRA CONFIGURAÇÃO

Interessante observar as mudanças que podem ser providas quando alteradas as configurações da rede. Aqui, para efeito de comparação por exemplo, podemos fazer ajustes em alguns parâmetros para conferir de que forma o resultado sofre alteração.

Em redes de grandes proporções e com grande volume de dados, testes nas configurações da rede podem significar mudanças significativas na precisão do modelo.

No nosso caso, resetamos a rede e voltamos a etapa de definição do modelo, alterando a função de ativação da última camada intermediária de *ReLU* para *softmax*.

Mexemos também no código da função *model.fit*, descomentando a parte referente ao *earlystopping*, inserindo-o na função, e por fim aumentamos o número de *epochs* (iterações), de 100 para 150.

Podemos conferir na figura 20, que as curvas de perda, apesar de estarem diferentes em relação ao observado no que aplicamos antes, continuam em uma boa sintonia entre Treino e Validação.

Interessante observar, que mesmo com 150 iterações definidas, a função *earlystopping* fez o algoritmo parar a incrementação antes da *epoch* 120, por não verificar mais um incremento minimamente significativo na nossa rede, o que significaria que estaria desperdiçando recursos de processamento em algo que não traria um retorno que compensasse.

Depois, é possível verificar em sublinhado na função *predict*, qual tipo de planta a rede vai apontar como referente a aquele exemplar, nesse caso aqui ela continua acertando as cinco linhas que imprimiu.

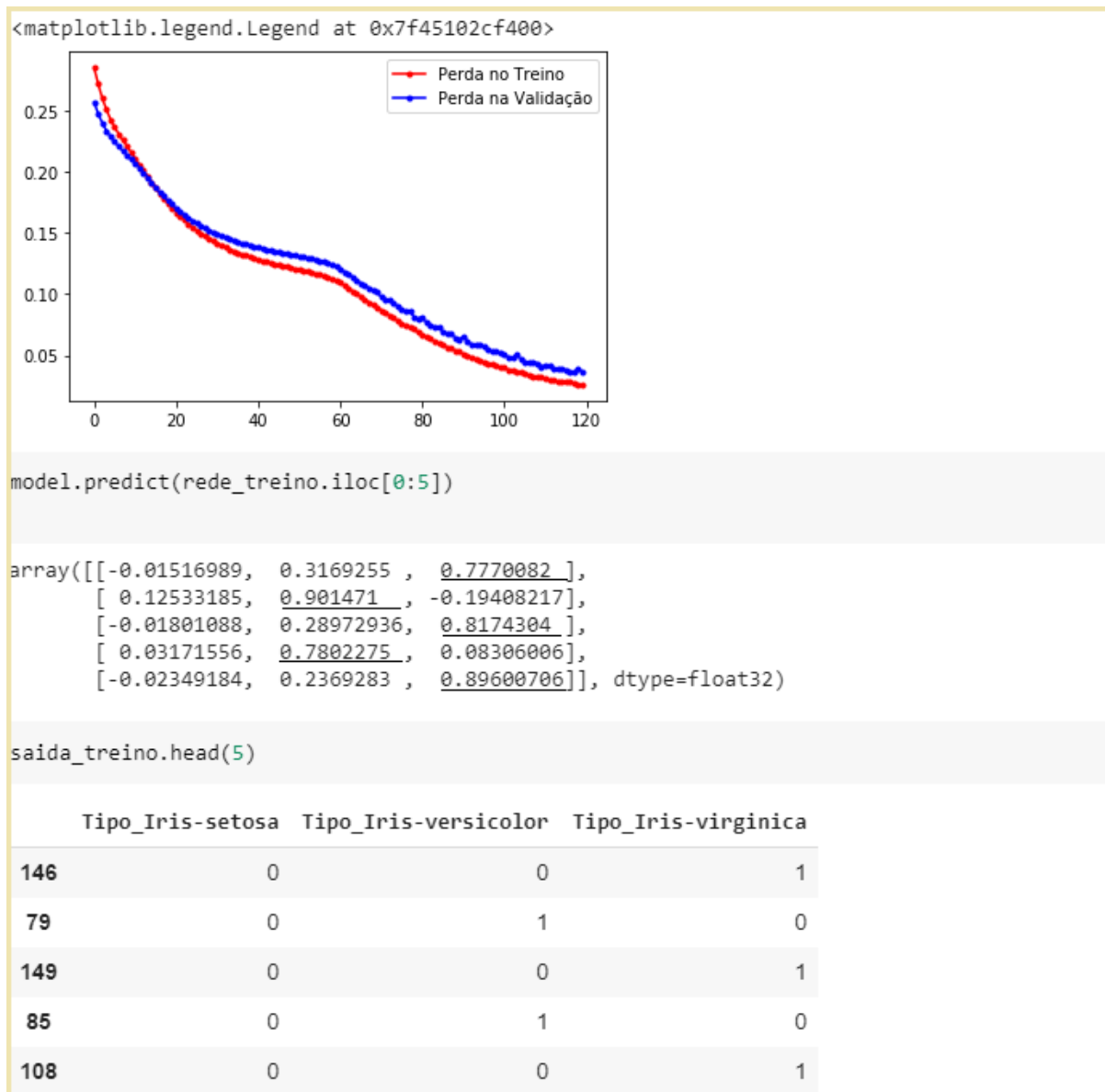


Figura 20 – Código: Alternativa ao modelo de rede, curvas

As matrizes de confusão, continuam apontando um alto índice de acertos para cada tipo de planta, tendo sido conferidos três erros nos casos de treino e um nos casos de teste. Na figura abaixo, vemos a impressão dessas matrizes, como também da acurácia, e dessa vez, a acurácia de teste foi maior.

```

#Imprimindo a matriz
confusion_matrix(valor_real, valor_pred)

array([[42,  0,  0],
       [ 0, 37,  2],
       [ 0,  1, 38]])

#Calculando a acurácia do algoritmo
from sklearn.metrics import accuracy_score
acuracia = (accuracy_score(valor_real, valor_pred) * 100)

print ("A acurácia dos casos de treino foi de %d" % (acuracia), "%.")

A acurácia dos casos de treino foi de 97 %.

#Imprimindo a matriz
confusion_matrix(valor_real, valor_pred)

array([[ 8,  0,  0],
       [ 0, 11,  0],
       [ 0,  1, 10]])

#Calculando a acurácia do algoritmo
from sklearn.metrics import accuracy_score
acuracia = (accuracy_score(valor_real, valor_pred) * 100)

print ("A acurácia dos casos de teste foi de %d" % (acuracia), "%.")

A acurácia dos casos de teste foi de 96 %.

```

Figura 21 – Código: Alternativa ao modelo de rede, matriz e acurácia

## 4 CONCLUSÕES

Com base no apresentado aqui, foi possível observar como o aprendizado de máquina está presente a nossa volta, constituindo parte do meio com o qual interagimos.

Constatamos a grande evolução que tivemos desde meados do século passado, quando esta abrangente área de estudos ainda engatilhava nos sonhos de percursores como Arthur Samuel.

Nesse trabalho, foi proposto um estudo sobre redes neurais, onde foram apresentadas suas características e os conceitos envolvidos. Foi apresentado um exemplo de teste, onde está presente o uso de *backpropagation*, e onde foi possível observar as etapas e o funcionamento da rede para um entendimento geral.

Foi proposta também, uma implementação de rede de classificação, utilizando um banco de dados com informações sobre exemplares de uma planta conhecida como Íris, cujo objetivo era classificar a qual tipo dessa espécie cada exemplar pertencia.

Após a realização desse experimento, algumas observações puderam ser constatadas:

- A definição do modo de ativação teve grande importância no sucesso da predição. Como a melhor escolha da ativação depende do tipo de rede e de base de dados, tivemos maior sucesso com o par *ReLU-softmax*.
- Foi perceptível a diminuição do custo/perda ao longo das iterações da rede neural. O número de *epochs* configurado para um valor baixo retornaria uma acurácia menos perfeita.
- A mudança do modo de ativação da última camada de *ReLU* para *softmax* e o incremento do número de iterações possibilitou um aumento de 10% na acurácia dos casos de treino no caso aqui testado.
- A perda ao longo da validação acompanha a perda ao longo do treino da rede, o que é muito bom para a confiança do algoritmo.

Após essas conclusões, e levando em conta o estudo que foi feito aqui, podemos dizer que seria interessante do ponto de vista de trabalhos futuros:

- O aprimoramento do modelo, de forma que pudesse ser aplicado a outras espécimes de plantas sem perda de precisão.
- Um comparativo entre as funções de ativação existentes em relação ao número de iterações, camadas e nós, para redes classificadoras que seguissem este padrão, com objetivo de diminuir o tempo de processamento e aumentar o tempo de resposta.
- Desenvolvimento de uma aplicação que captasse as dimensões de uma planta através de lentes de câmera e realizasse a classificação com base na imagem, sem depender de um banco de dados contendo valores de entrada. Tal aplicação não necessitaria se limitar a plantas, podendo ser estendida a mais tipos de classificações.
- Estudo de outros tipos de redes neurais profundas presentes na literatura, para aplicação como solução de problemas de classificação.

## REFERÊNCIAS BIBLIOGRÁFICAS




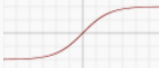
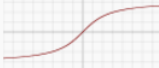




1. SEARLE, John. *Minds, brains, and programs*. *Behavioral and Brain Sciences*, 1980.
2. TAYLOR, Christine. *Structured vs. Unstructured Data*. Acessado em 06/10/2019.  
Disponível em <https://www.datamation.com/big-data/structured-vs-unstructured-data.html> .
3. *Unstructured Data and the 80 percent rule*. 2008. Acessado em 08/09/2019.  
Disponível em <http://breakthroughanalysis.com/2008/08/01/unstructured-data-and-the-80-percent-rule/>.
4. BROWNLEE, Jason. *Master Machine Learning Algorithms - Discover How They Work and Implement Them from Scratch*, 2016.
5. ROSENBLATT, Frank. *The Perceptron--a perceiving and The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain*, 1958.
6. HAYKIN, S. *Redes neurais: princípios e prática*. Porto Alegre: Bookman, 2001
7. *Deep Learning Book* – Em Português, Online e Gratuito. Acessado em 01/10/2019.  
Disponível em <http://deeplearningbook.com.br/o-neuronio-biologico-e-matematico/> .
8. *Deep Learning Book* – Em Português, Online e Gratuito. Acessado em 01/10/2019.  
Disponível em <http://deeplearningbook.com.br/funcao-de-ativacao/> .
9. *Understanding Learning Rates and How It Improves Performance in Deep Learning*.  
Acessado em 28/10/2019. Disponível em <https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10>.
10. VAN ROSSUM, G. e DRAKE, F.L. *Python tutorial, Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands*, 1995.
11. *Google Colaboratory*. Acessado em 05/12/2019. Disponível em <https://colab.research.google.com/notebooks/welcome.ipynb>
12. DUA D. e GRAFF, C. *UCI Machine Learning Repository Irvine, CA: University of California, School of Information and Computer Science*. 2019.
13. CHOLLET, Francois e OUTROS. *Keras*, 2015. Disponível em <http://keras.io>.

14. MCKINNEY, W. e OUTROS. *Data structures for statistical computing in python*. In *Proceedings of the 9th Python in Science Conference*. pp. 51–56, 2010.
15. PEDREGOSA, Fabian e OUTROS. *Scikit-learn: Machine Learning in Python*. JMLR 12, pp. 2825-2830, 2011
16. FISCHER, R.A. *Annals of Eugenics*, 7: 179-188 - *The Use of Multiple Measurements in Taxonomic Problems*, 1936.
17. P. KINGMA, Diederik e BA, Jimmy. *Adam: A Method for Stochastic Optimization*. arXiv preprint arXiv:1412.6980, 2014.
18. J. D. HUNTER. *Matplotlib: A 2D Graphics Environment, Computing in Science & Engineering*, vol. 9, no. 3, pp. 90-95, 2007.



## ANEXO A – PLANILHA DE FUNÇÕES DE ATIVAÇÃO E CASOS DE USO

Aqui temos uma tabela, resumida, com tipos de funções de ativação, sua representação geométrica e sua respectiva equação.

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

Disponível no site: <https://hackernoon.com/everything-you-need-to-know-about-neural-networks-8988c3ee4491>