UNIVERSIDADE FEDERAL FLUMINENSE

ANTONIO CARLOS DO NASCIMENTO CUNHA JUNIOR

IMPLEMENTAÇÃO DE ALGORITMO DE CLASSIFICAÇÃO PARA REDES NEURAIS COM USO DE *BACKPROPAGATION*

Niterói

2019ANTONIO CARLOS DO NASCIMENTO CUNHA JUNIOR

IMPLEMENTAÇÃO DE ALGORITMO DE CLASSIFICAÇÃO PARA REDES NEURAIS COM USO DE BACKPROPAGATION

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

ANTONIO CARLOS DO NASCIMENTO CUNHA JUNIOR

ESTUDO E IMPLEMENTAÇÃO DE UM ALGORITMO DE CLASSIFICAÇÃO PARA REDES NEURAIS PROFUNDAS

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, \_\_\_ de \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ de ANO.

Banca Examinadora (provisório):

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Prof. ou Profa. <NOME>, <Título>. – Orientador ou Avaliador

<Sigla da Universidade> - <Nome da Universidade>

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Prof. ou Profa. <NOME>, <Título>. – Orientador ou Avaliador

<Sigla da Universidade> - <Nome da Universidade>

AGRADECIMENTOS

RESUMO

A ser feito.

Palavras-chaves: palavra1, palavra2 e palavra3.

ABSTRACT

A ser feito.

Key words:

LISTA DE ILUSTRAÇÕES

[**Figura 01: Programadoras trabalhando no *ENIAC* ………………………………….19**](#__RefHeading___Toc378694305)

**Figura 02: Relação de Aprendizado de Máquina com Mineração de Dados** **…….19**

**Figura 03: Funcionamento das camadas de uma Rede Neural ……………………19**

**Figura 04: Representação de um Neurônio Artificial ……………………………….19**

**Figura 05: Exemplo de Função de Ativação Linear .…………………………………19**

**Figura 06: Exemplo de representação de Função de Ativação Não Linear ……..19**

**Figura 07: Exemplo de valores iniciados em uma RNA …………………………….19**

**Figura 08: Fórmula e representação da função Logística ………………………….19**

**Figura 09: Ilustração da Regra da Cadeia no nó da camada de saída ……………19**

**Figura 10: Código:** **Colunas e impressão por comando *head* ……………………..19**

**Figura 11: Código: Mapeamento da saída como colunas ………………………….19**

**Figura 12: Código: Função Split aplicada a treino e teste ………………………….19**

**Figura 13: Código: Impressão para verificação de treino e teste …………………19**

**Figura 14: Código: Modelagem das camadas da Rede Neural ………………….…19**

**Figura 15: Código: Treino do modelo da rede .…………………………………….....19**

**Figura 16: Código: Comparação de curvas referentes a perda ...…………………19**

**Figura 17: Código: Comparação da função *Predict* com valores reais ………….19**

**Figura 18: Código: Predição dos casos de treino e acurácia do algoritmo ……..19**

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

**Figura 20: Código: Alternativa ao modelo de rede, curvas ………………………...19**

**Figura 21: Código: Alternativa ao modelo de rede, matriz e acurácia .…………...19**

LISTA DE EQUAÇÕES

[**Equação 1: Função de Ativação ………………………………………………………..19**](#__RefHeading___Toc378694305)

**Equação 2: Função de Erro ou Custo ………………………………………………….19**

**Equação 3: Gradiente em Relação a wi ……………………………………………….19**

LISTA DE ABREVIATURAS E SIGLAS

ENIAC – *Electronic Numerical Integrator and Computer*

IA – Inteligência Artificial

ML – *Machine Learning*

RELU – *Rectified Linear Unit*

**2 – F****UNDAMENTAÇÃO 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[[1]](#footnote-1) 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 1 - Programadoras trabalhando no ENIAC

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 MLhoje, 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 androides. 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 diferenciará 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 2 - 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 difícies 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 caracteristicas 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 diferente 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 3 - 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].

2.2.2 – NÓS (NEURÔNIOS) E PESOS DAS CONEXÕES

Funcionando como pequenas unidades computacionais, os neurônios realizam a soma entre os valores de entradas, fornecendo um valor de saída. As diversas conexões provenientes de neurônios da camada anterior possuem pesos (ou força). Esse peso, funciona como um multiplicador daquela conexão, reforçando a sua importância para a soma a ser realizada pelo neurônio de destino.

Três neurônios de uma determinada camada *A*, por exemplo, possuem pesos diferentes em suas conexões com um neurônio de uma camada *B*. Entretanto, para um outro nó da mesma camada *B*, as conexões com os nós da camada *A* serão diferentes, e por consequência, o valor do peso de cada uma dessas conexões também diferirá.

2.2.4 – BIAS

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 servem para aumentar ou diminuir a importância das conexões entre neurônios, a Bias é referente ao próprio Neurônio. Portanto, o valor do peso afetará a conexão, mas a Bias afetará o valor computado em um neurônio e consequentemente, todas as conexões originárias nele.

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 esperado.

2.2.5 – FUNÇÃO DE ATIVAÇÃO

Para evitar que uma pequena alteração em peso ou bias afete toda a rede além do ponto ou segmento esperado, existe a função de ativação. 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 xis, 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.

2.2.6 – REPRESENTAÇÃO MATEMÁTICA

Como pode ser observado na figura 4, um neurônio recebe valores diversos de até sendo n o número de entradas, com pesos sinapticos até respectivos aos sinais. A função de soma calcula seu valor aplicado a um bias  observando sua função de ativação φ para definir seu sinal de saída .



Figura 4 - Representação de um Neurônio Artificial

A função de ativação age em cima do valor que pode ser definido de forma simplicada como a equação abaixo :

Equação 1 – Função de ativação

Esse valor recebido pode ser representado em relação ao resultado de sua função de ativação. Conforme demostrado 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 (*xi,yi*) de entradas da rede e suas consecutivas desejadas saídas. Também é esperado que a rede seja do tipo *feed-forward* [[2]](#footnote-2)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.

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:

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[[3]](#footnote-3) 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.

Aplicando a fórmula da função de ativação com a saída do neurônio intermediário h1, teremos .

Aplicando o mesmo em relação a h2, temos .

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:

Repetindo o processo de encontrar o valor de saída (*output*) com a função de ativação, teremos:  e .

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:

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 ½ 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 e , temos então respectivamente:

E para o erro total da rede, fazemos:

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 , , , e .

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

Equação 3 – Gradiente em relação a wi

Podemos interpretar a equação 3, como a derivada parcial do erro total em relação a , ou, simplesmente, o gradiente em relação a Aplicamos aqui nesta fórmula, a regra da cadeia[[4]](#footnote-4), conforme ilustrado na figura 9.

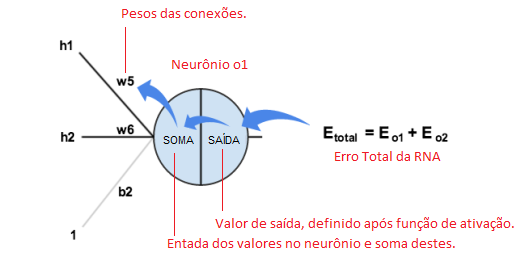


Figura 9 – Ilustração da regra da cadeia no nó da camada de saída.

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

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 .

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

A saída esperada, era de 0,01, a encontrada foi de 0,75136507. O zero na equação, corresponde a derivada de 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 muda em relação a sua soma. E calcular a derivada da saída implica em calcular a derivada de sua função de ativação logística, que pela figura 8, sabemos ser . Através da definição de distribuição logística[[5]](#footnote-5), 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:

2.3.7 – O GRADIENTE DA SOMA EM RELAÇÃO AO PESO

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

Como estamos lidando com a derivada da soma em relação a , o valor da bias e o valor do peso  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 , definimos:

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

2.3.8 – TAXA DE APRENDIZADO

Para diminuição do erro, devemos subtrair do valor de do peso w5 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 () em 0.5. Sendo assim, definimos nosso peso ajustado :

E repetindo todo o processo em relação a , e , encontraremos:

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 ( a ), toda vez que um dos pesos cujo ajuste já tenha sido feito ( a ) 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 a , o processo é similar, entretanto, leva-se em conta que a saida 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:

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*, utilizando máquina virtual cedida a este propósito pelo *Google*, através da ferramenta conhecida como *Google Colab*, será utilizada um banco público de dados, o repositório de aprendizado de máquina UCI. Serão importados, no código, funções pertencentes ao Keras, uma biblioteca de código aberto escrita em Python e voltada a redes neurais. Serão utilizadas também as bibliotecas Pandas e scikit-learn, também escritas em Python, sendo esta última comumente utilizada em algortimos 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 caracterísiticas 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 centrímetros 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 predizer 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 biológo inglês Ronald Fischer em 1936. É conhecido como *Iris flower data set* ou *Fisher's Iris data set* [10]. É composto de dados de 50 espécimes de cada um dos tipos de Íris citados acima.

**3.2 – INICIANDO A REDE**

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 carater 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.

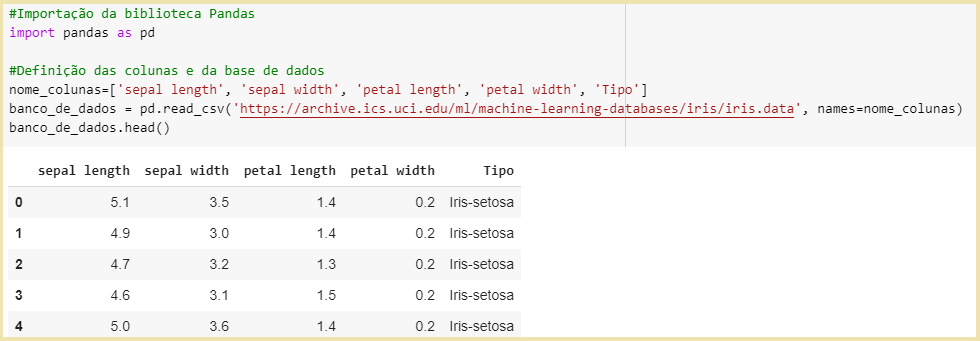


Figura 10 – Código: Colunas e impressão por comando *head*

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.

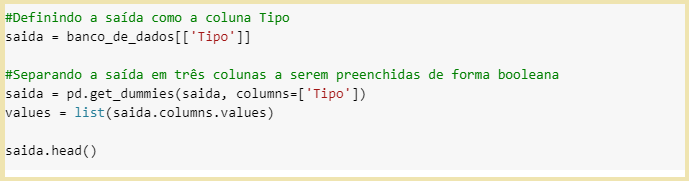


Figura 11 – Código: Mapeamento da saída como colunas

Na sequencia, 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 porcento. A função interpretará que vinte porcento dos dados devem ser deixados para o teste do algoritmo, enquanto que os oitenta porcento 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.

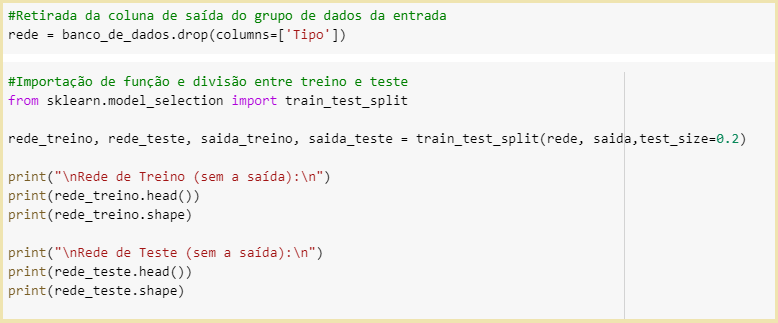


Figura 12 – Código: Função Split aplicada a treino e teste

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 porcento da tabela total de entradas.

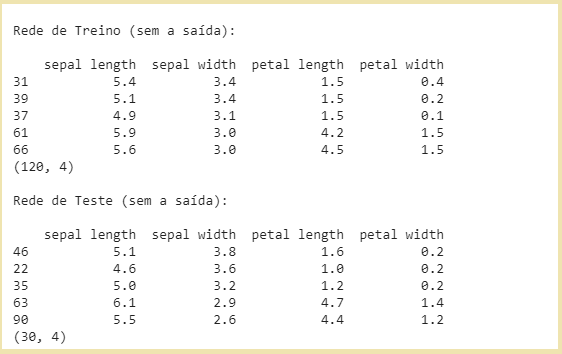


Figura 13 – Código: Exemplo de impressão para verificação de treino e teste

**3.3 – MODELAGEM E COMPILAÇÃO**

Neste etapa, definiremos o nosso modelo. Importamos do *Keras* o tipo *Sequencial* 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 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.

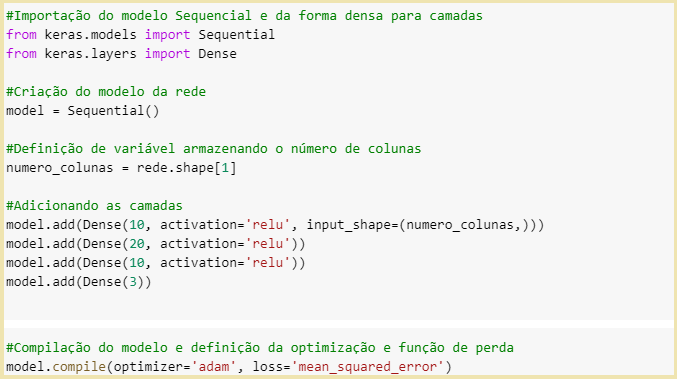


Figura 14 – Código: Modelagem das camadas da Rede Neural

Na última linha do código na figura acima, há a definição do optimizador *Adam*, que é um algoritmo para a taxa de aprendizado, bem como há a definição do erro quadrático médico 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 dimunindo a incidencia 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.

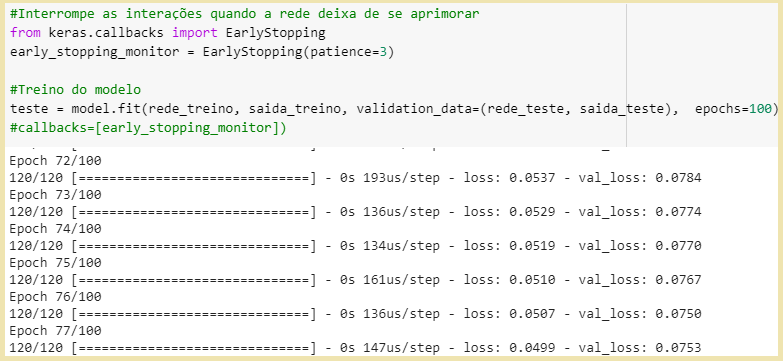


Figura 15 – Código: Treino do modelo da rede

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

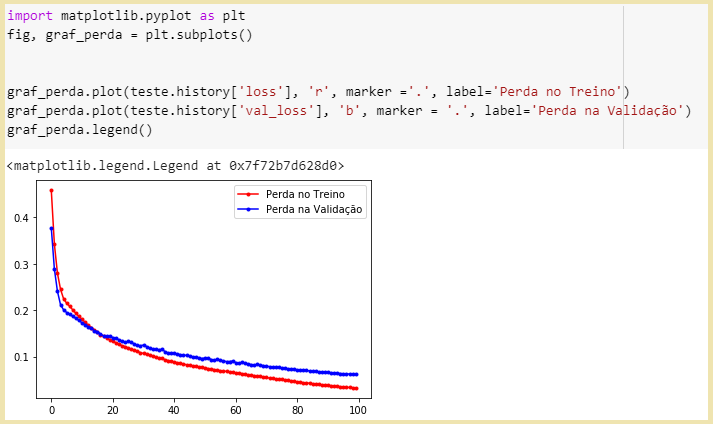


Figura 16 – Código: Comparação de curvas referentes a perda

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 predileções 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*.

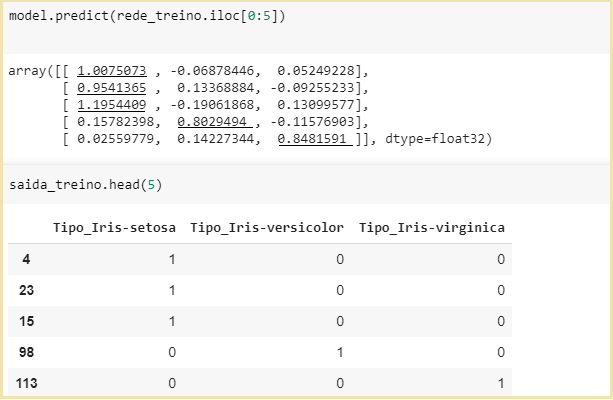


Figura 17 – Código: Comparação da função *Predict* com valores reais

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 porcento (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 porcento para o tipo *setosa* no *index* 4, noventa e cinco porcento também para *setosa* no *index* 23 e bem mais de cem porcento para o mesmo tipo no *index* 15. Calculou em oitenta porcento para o tipo *versicolor* no *index* 98 e, logo abaixo, oitenta e quatro porcento para o *virginica*. Na saída real, pelo comando *head*, vemos que acertará os cinco casos.

**3.4 – ACURÁCIA DA REDE**

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. Fazendo uso da biblioteca *skylearn*, 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.

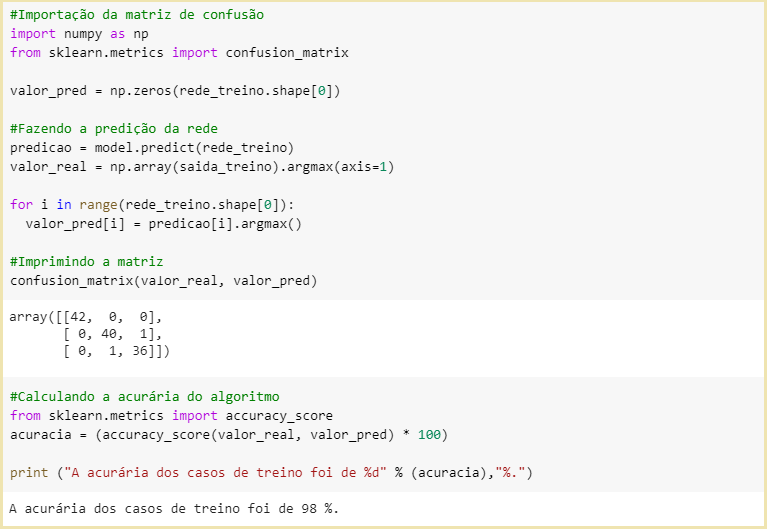


Figura 18 – Código: Predição dos casos de treino e acurácia do algoritmo

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%**.

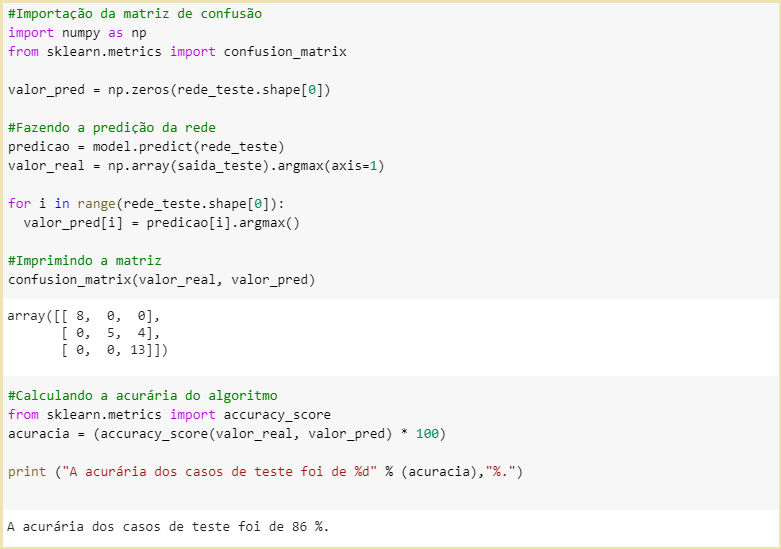


Figura 19 – Código: Predição dos casos de teste e acurácia do algoritmo

**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 modem 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.

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.

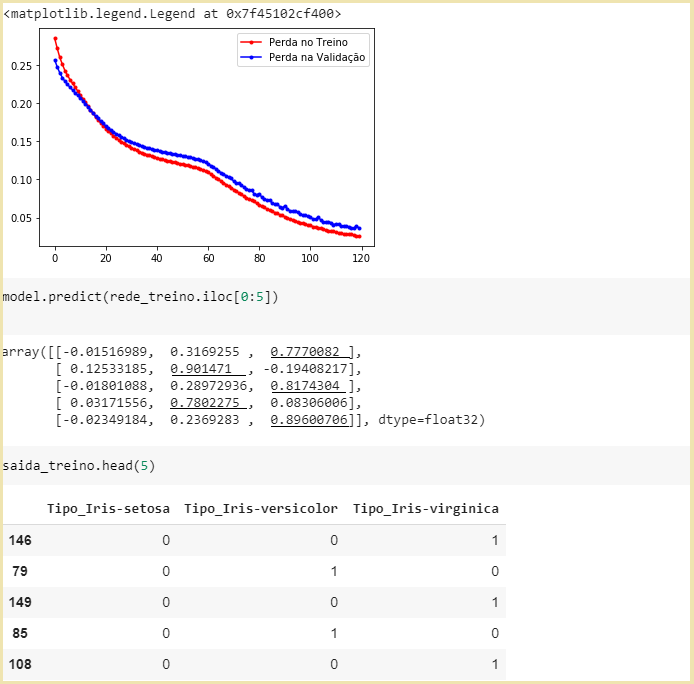


Figura 20 – Código: Alternativa ao modelo de rede, curvas

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 acurária de teste foi maior.

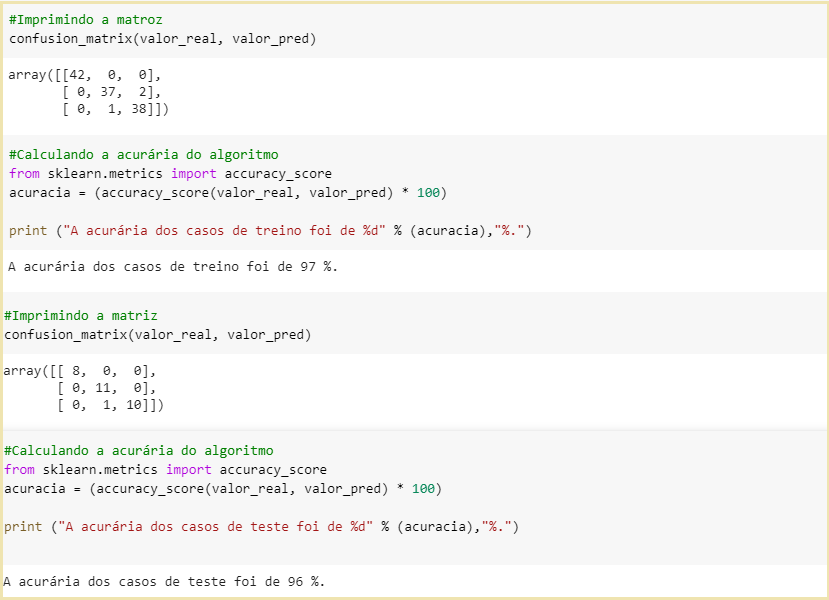


Figura 21 – Código: Alternativa ao modelo de rede, matriz e acurácia

**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> .

1. *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/>.
2. BROWNLEE, Jason. *Master Machine Learning Algorithms - Discover How They Work and Implement Them from Scratch*, 2016.
3. ROSENBLATT, Frank. *The Perceptron--a perceiving and The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain*, 1958.
4. HAYKIN, S. Redes neurais: princípios e prática. Porto Alegre: Bookman, 2001
5. *Deep Learning Book* – Em Português, Online e Gratuito. Acessado em 01/10/2019.

Disponível em <http://deeplearningbook.com.br/o-neuronio-biologico-e-matematico/> .

1. *Deep Learning Book* – Em Português, Online e Gratuito. Acessado em 01/10/2019. Disponível em <http://deeplearningbook.com.br/funcao-de-ativacao/> .
2. *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>.
3. FISCHER, R.A. *Annals of Eugenics*, 7: 179-188 - *The Use of Multiple Measurements in Taxonomic Problems*, 1936. Disponível em: <https://digital.library.adelaide.edu.au/dspace/handle/2440/15227>

**APÊNDICE A – ANALOGIA PARA COMPREENSÃO DO PROBLEMA DO BIG DATA**

(REESCREVER, NÃO ESTÁ PRONTO, SÓ SE DER TEMPO)

É como se um grande galpão, de tamanho aparentemente ilimitado, recebesse várias caixas de destinatários diferentes por hora, e a capacidade de analisar e separar as caixas, de acordo com seu conteúdo e importância para serem enviados a seu destino, estivesse abaixo do necessário para lidar com a quantidade de caixas que chegam. Que fazemos então? A princípio, pensamos em minimizar o problema, para então controlá-lo. Organizamos corredores nesse galpão, passamos a desenvolver empilhadeiras e esteiras para ajudar na movimentação das caixas. Nesta analogia, podemos entender que as empilhadeiras e esteiras são os programas convencionais, que, em suas linguagens de programação, ajudam os computadores a interpretar instruções e realizar tarefas cada vez mais complexas. Entretanto, como podemos perceber na computação, escrever comandos para eventos e tarefas não elimina a necessidade do protagonismo humano, e isso é até bom em muitas sub áreas da computação. Já no galpão, as esteiras e empilhadeiras não eliminam o fato de cada caixa ter que ser conferida e separada e embora as máquinas ajudem em partes da tarefa, elas não o fazem por completo. A popularização da internet e a disponibilidade de dados via multiplataforma nos leva então ao caos completo da informação, passamos a dobrar o volume de dados disponíveis na rede a cada dois anos, e nosso galpão não parece chegar a seu limite de armazenamento tão cedo, pois cada vez mais caixas passam a chegar por hora, e cada vez mais a relação de caixas recebidas por caixas enviadas aumenta. O protagonismo humano aqui torna-se tão inútil nessa situação que se torna algo como desesperador ver o acúmulo de caixas numa velocidade maior que se poderia prever. Esse acúmulo, essa quantidade inexplorada de informação crescendo em pilhas e pilhas pelo galpão, na computação, chamamos de Big Data.

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

(A SER DESENVOLVIDO)



1. **Alan Turing** (1912 – 1954):Considerado o pai da ciência da computação. [↑](#footnote-ref-1)
2. 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. [↑](#footnote-ref-2)
3. O “e” presente na fórmula refere-se a constante matemática, também conhecida como número de Euller cujo valor aproximado é 2,718. [↑](#footnote-ref-3)
4. Fórmula desenvolvida por Gottfried Leibniz para a derivada de uma função composta de duas funções. [↑](#footnote-ref-4)
5. Um exemplo de distribuição de probabilidade contínua. Utilizada nos campos da teoria das probabilidades e estatística. [↑](#footnote-ref-5)