DEEP LEARNING

AULA 2

Prof. Marco Teixeira

CONVERSA INICIAL

Anteriormente, estudamos os conceitos de Deep Learning e um pouco de sua história. Vimos que existem diferentes tipos de arquiteturas, cada uma voltada para um objetivo diferente. Algo que a maioria das arquiteturas possuem em comum é o uso das Redes Neurais Artificiais (RNAs), mais especificamente a Perceptron multicamada, normalmente utilizada no final de uma arquitetura para realizar a classificação das características extraídas dos dados. Como trata-se de uma parte importante para o funcionamento de arquiteturas de Deep Learning, nesta aula, estudaremos um pouco mais a fundo o funcionamento de RNAs, além de criarmos e treinarmos redes MLP (Perceptron multicamada).

Ao final desta aula, esperamos atingir os seguintes objetivos que serão avaliados ao longo da disciplina da forma indicada:

- Conhecer os conceitos básicos de uma RNA;
- Conhecer o processo de aprendizagem de uma RNA;
- Desenvolver um experimento prático.

TEMA 1 – REDES NEURAIS E SUA INSPIRAÇÃO

A criação de redes neurais artificiais veio da motivação de que o cérebro humano age e processa informações de maneira diferente ao do computador. O cérebro processa informações de maneira não linear, por exemplo, diferentemente da maioria dos algoritmos matemáticos existentes.

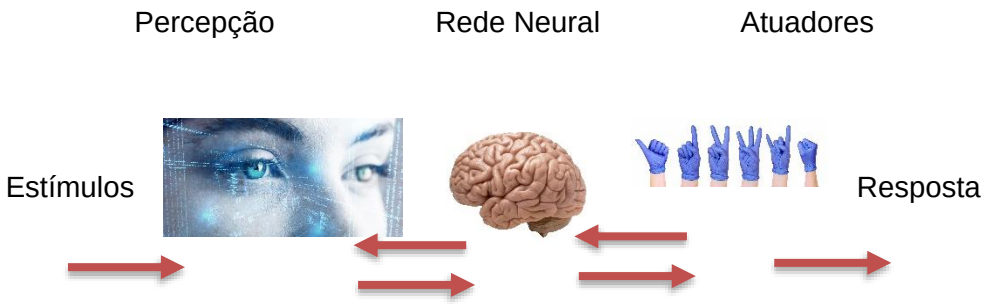
Um exemplo da complexidade do cérebro pode ser visto em um morcego. O morcego utiliza o sonar para identificar suas presas. Ele consegue identificar a distância de suas presas, tamanho, velocidade, entre várias outras informações, o que mesmo com a tecnologia existente hoje, é algo extremamente complexo de se replicar com o uso de técnicas de processamento de sinais de ultrassom e de computadores potentes.

Dessa forma, iniciou-se o estudo do cérebro humano para fins de tentativa de replicação de seu funcionamento por meio de computadores. Como veremos nesta aula, RNAs possuem conceitos similares aos que se conhece do funcionamento do cérebro humano até o momento, mas encontra-se longe de atingir o mesmo poder de similaridade, reconhecimento de padrões e outros aspectos existentes no cérebro humano.

1.1 Cérebro humano

Observe as figuras a seguir:

Figura 1 – Representação do sistema nervoso, a partir de um estimulo até uma resposta



Fonte: Teixeira, 2021, com base em Simon, 2001. Crédito: vectorfusionart/Adobe Stock; Danheighton/Adobe Stock; patila/Adobe Stock.

A figura ilustra os componentes do nosso sistema nervoso, assim como apresenta as direções de informações. Para executarmos uma ação com o nosso braço, por exemplo, o nosso cérebro continuamente envia informação para o nosso braço, analisando as suas respostas, para, então, ser capaz de realizar um controle do movimento com exatidão.

Com o avanço dos estudos no cérebro humano, principalmente pelo trabalho de Cajál (1911), foram introduzidos os neurônios, que, mais tarde, seriam utilizados como base para a criação das RNAs.

1.2 Conclusão

É preciso entendermos alguns comportamentos do cérebro humano para podermos compreender o comportamento da RNA. Assim, devemos entender as sinapses, que possuem como funcionalidade realizar a interação entre os neurônios e a memorização da informação. É estimado que o cérebro humano possua aproximadamente 10 bilhões de neurônios e 60 trilhões de sinapses.

TEMA 2 – REDES NEURAIS ARTIFICIAIS E SEU NEURÔNIO ARTIFICIAL

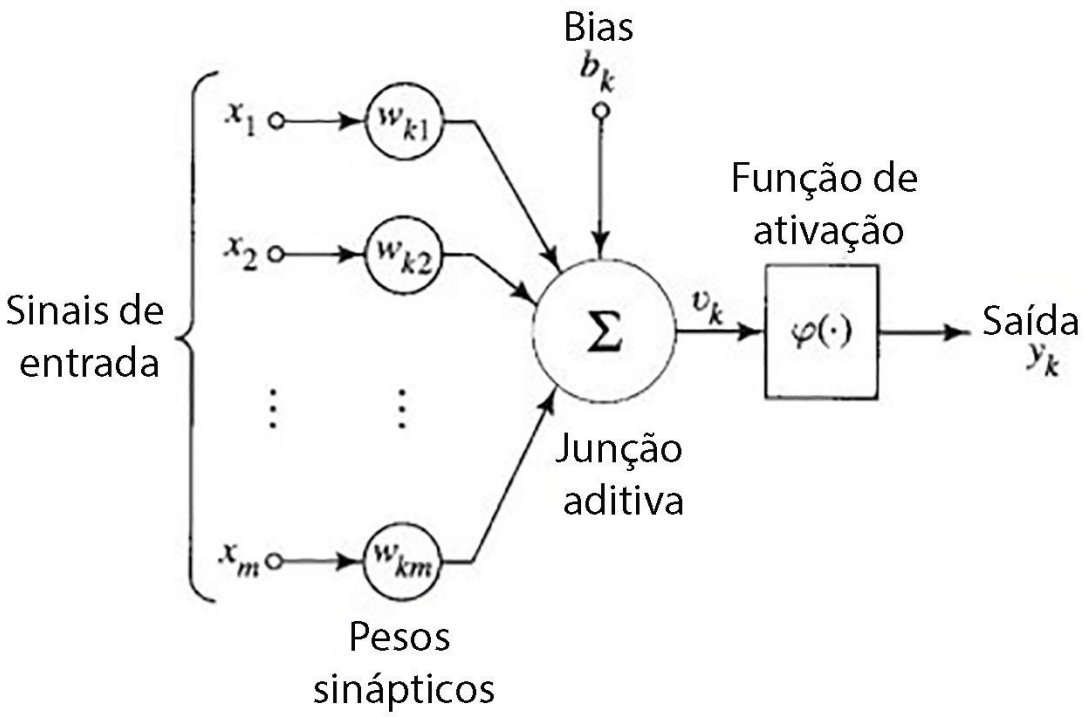
O tema 1 apresentou os conceitos do cérebro humano que inspiraram as RNAs. Como várias outras técnicas bioinspiradas, as redes neurais tentam imitar

o comportamento do cérebro humano. Para conseguir tal feito, cada componente do funcionamento do cérebro precisa ser “digitalizado”. Vamos começar falando sobre os neurônios artificiais.

2.1 Neurônio artificial

O neurônio artificial é a base para a criação de RNAs. Ele foi desenvolvido como tentativa de imitar o comportamento de um neurônio biológico. A figura a seguir apresenta uma representação genérica de um neurônio artificial. O neurônio pode sofrer leves modificações e irá continuar sofrendo à medida que pesquisadores descobrem novas técnicas que aumentam as capacidades das redes neurais, ou novos detalhes sobre neurônios biológicos antes desconhecidos.

Figura 2 – Representação de um neurônio artificial



Fonte: Haykin, 2001.

Vamos abordar um pouco melhor os detalhes do neurônio artificial apresentado na figura 3. Os valores X_1 até X_m representam as entradas de dados no neurônio, similar ao que ocorre nos estímulos existentes no sistema nervoso humano. Assim como nos neurônios biológicos, os artificiais possuem um conjunto de sinapses. Cada sinapse está representada pela letra W, em que k

representa o neurônio ao qual ela está ligada (cada sinapse pode se conectar a um ou mais neurônios) em correspondente a sua identificação.

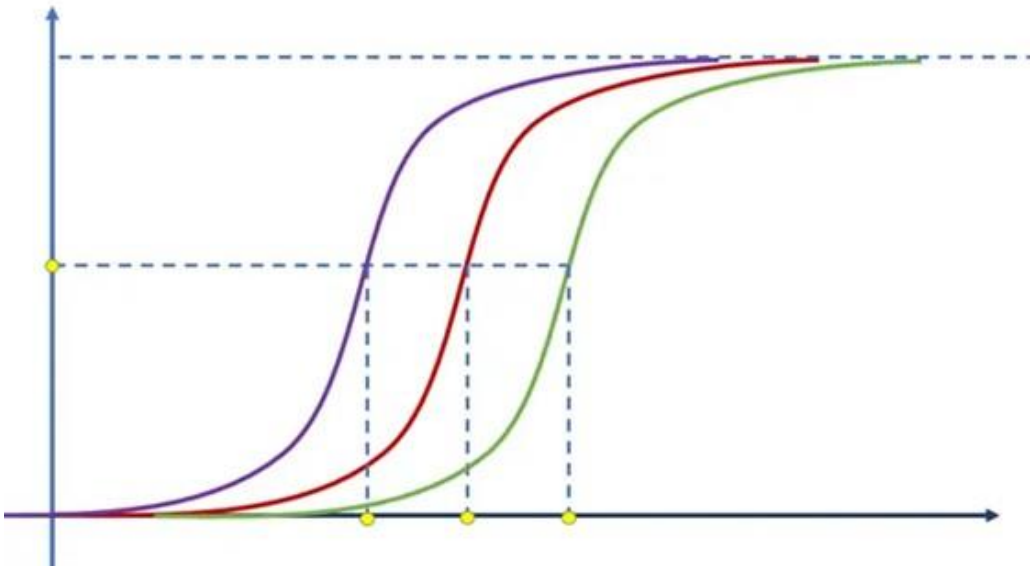
O somador tem como funcionalidade somar os sinais de entrada, que são ponderados pelas sinapses do neurônio, podendo ser identificado como um combinador linear. Já a função de ativação tem por finalidade restringir a saída do neurônio a intervalos permissíveis.

Dependendo do tipo de ativação e do tipo de RNA desenvolvida, a função de ativação pode ser puramente binária, dizendo que o neurônio está ativo ou não ativo.

2.2 O bias

Na figura 3 foi apresentado também o conceito de **bias**. O bias tem por finalidade aumentar ou diminuir a entrada na função de ativação, a depender se o seu valor é positivo ou negativo. Uma função de ativação sigmoide, por exemplo, pode ser deslocada de acordo com o valor do bias, como apresentado pela figura 3. Esse deslocamento é interessante para que a rede se torne genérica. O valor de bias é treinado juntamente com os valores das sinapses W, dessa forma, é possível que a função de ativação se desloque, chegando de fato ao objetivo desejado. Por outro lado, a adição da bias pode gerar um custo computacional maior durante o treinamento.

Figura 3 – Exemplo de deslocamento de uma função de ativação Sigmoide causado pelo uso de bias



Fonte: <https://cdn.shortpixel.ai/spai/w_722+q_+ret_img+to_webp/https://iaexpert.academy/wp-content/uploads/2020/09/image-15.png>.

O bias é representado por b_k. Todo o neurônio apresentado na figura anterior pode ser representado pela **Equação 1**. Vale ressaltar que assim como nosso cérebro, uma RNA é composta por diversos neurônios. Falaremos mais sobre isso nas próximas seções.

Equação 1 – Representação matemática de um neurônio artificial (entrada da função de ativação)

$$y_k = \varphi\,(u_k + b_k)$$

Onde b_k representa o bias, somado a u_k (**Equação 2**). O símbolo φ representa a função de ativação, o qual tem como saída o valor correspondente à resposta do neurônio nos dados de entrada, com base nos pesos sinápticos.

Equação 2 – Representação matemática de um neurônio artificial, somática dos pesos sinápticos multiplicados pelas entradas x

$$u_k = \sum_{j=1}^{m} w_{kj} x_j$$

2.3 Representação matemática de um neurônio

Na **Equação 2,** u_k representa o somatório dos pesos sinápticos multiplicado pelos valores de entrada. Esse valor é somado ao bias e, então, aplicado à função de ativação.

Como o bias é um valor somado após ser aplicado à somatória dos pesos sinápticos vezes os sinais de entrada, é possível simplificar a função apresentada na Equação 1, convertendo o valor de bias em um peso sináptico com entrada x = 1. Dessa forma, W_{k0} terá o peso de bias e será somado uma única vez na Equação 2, sendo possível converter a Equação 1 na Equação 3.

Equação 3 – Representação matemática do neurónio artificial, considerando o bias como um peso sináptico

$$y_k = \varphi(v_k) \qquad v_k = \sum_{j=0}^{m} w_{kj} x_j$$

2.4 Conclusão

Dessa forma, um neurônio artificial nada mais é que a somatória de seus pesos sinápticos (W_{kj}) multiplicados pelos valores de entrada (Xj) e aplicados a uma função de ativação, representado por Delta (δ).

Ainda existe muito o que se estudar a respeito de RNAs, mas esperamos que o conceito pareça mais real e menos “místico”. Vamos dar uma olhada nas funções de ativações mais utilizadas a seguir.

TEMA 3 – REDES NEURAIS ARTIFICIAIS E SUAS FUNÇÕES DE ATIVAÇÃO

A função de ativação é responsável por converter a **Equação 3** em uma saída de nosso neurônio. Ela é fundamental para o uso de RNAs. Em redes de múltiplas camadas, é possível utilizar mais de um tipo de função de ativação em neurônios diferentes. Vamos abordar os principais tipos de ativação existentes.

3.1 Função de limiar

Função binária, retorna 1 caso v seja maior que 0 e 0 caso v seja menor que 0. A Equação 4 apresenta o seu comportamento.

Equação 4 – Função limiar

$$\varphi(v) = \begin{cases} 1 & \text{se } v \geq 0 \\ 0 & \text{se } v < 0 \end{cases}$$

Podemos implementar esta função de ativação sem muitas dificuldades utilizando o Python. Vamos criar uma função chamada FuncaoLimiar, a qual terá como entrada um valor (v) e como saída o seu valor de ativação. Lembrando que o valor de v é fornecido pela **Equação 3**.

Código 1 – Exemplo de implementação da função de ativação Limiar em Python

****

```
def FuncaoLimiar(v):
    if (v >=0):
        return 1
    else:
        return 0
*****************************************
```

Fonte: Teixeira, 2021.

3.2 Função linear por partes

Diferentemente da função limiar, a função linear por partes define um intervalo ao qual pode assumir valores entre 0 e 1. Passado esse intervalo, que pode ser considerado como um limite da função, os valores são 0 e 1.

Equação 5 – Função linear por partes

$$\varphi(v) = \begin{cases} 1, & v \geq +\frac{1}{2} \\ v, & +\frac{1}{2} > v > -\frac{1}{2} \\ 0, & v \leq -\frac{1}{2} \end{cases}$$

Seguindo o que fizemos com a função Limiar, vamos converter a Função Linear por partes em uma função Python. Vamos chamar essa função de “FuncaoLinearPorPartes”. O Código 2 apresenta o resultado de nossa função. Lembrando que o valor de v é fornecido pela **Equação 3**.

Repare que se o valor for superior a 0,5, retorna 1; se for inferior a -0,5, retorna 0; e se estiver no intervalo de 0.5 e -0.5, retorna o próprio valor de v.

Código 2 – Exemplo de implementação da Função linear por partes em Python

```
*****************************************
def FuncaoLinearPorPartes(v):
    if (v >= 1/2):
        return 1
    elif (v <= -1/2):
        return 0
    else:
        return v
*****************************************
```

Fonte: Teixeira, 2021.

3.3 Função sigmoide

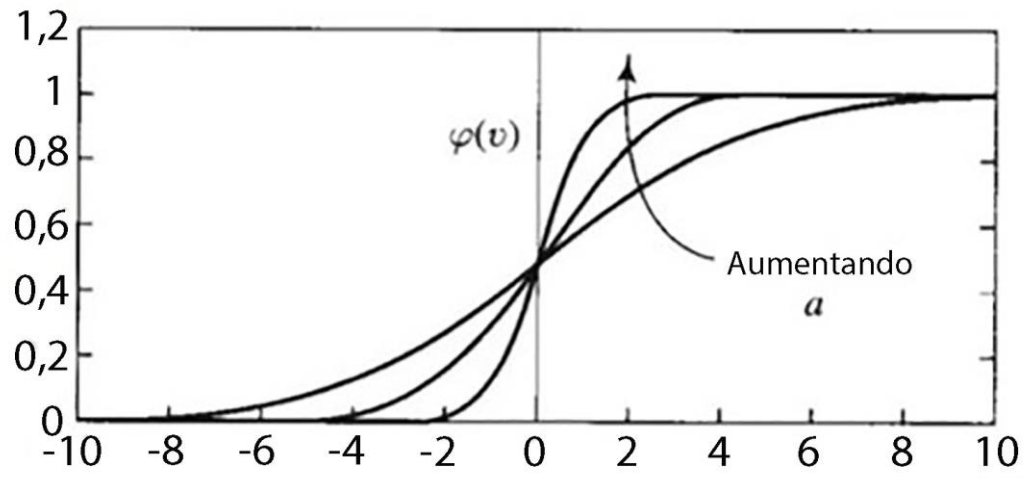
É o tipo de função mais utilizada em RNAs, principalmente pelos neurônios de saída. A **Equação 6** apresenta um exemplo, sendo ela a função logística, em que é possível assumir vários valores em um intervalo de 0 e 1.

Equação 6 – Função sigmoide

$$\varphi(v) = \frac{1}{1+\exp(-av)}$$

Observe que nessa equação existe uma variável não vista antes, chamada de “a”. Sua função é modificar a inclinação da função de ativação, como mostra a figura 4. Por padrão, o seu valor é 1, mas pode ser modificado.

Figura 4 – Exemplo do efeito a ser obtido ao modificar o valor de “a” em uma função sigmoide



Fonte: Haykin, 2001.

Para criarmos o código em Python da função Sigmoide, vamos utilizar uma biblioteca matemática em Python chamada “math”. A função é apresentada no código 3, sob o nome de “FuncaoSigmoide(v,a)”, recebendo dois valores, sendo o primeiro o v, obtido na **Equação 3**, e o segundo o valor de **“a”**. Caso não queira utilizar o valor de a, basta defini-lo como 1.

Código 3 – Exemplo de implementação da Função de ativação sigmoide em Python

```
*******************************************
import math
def FuncaoSigmoide(v,a):
    return ((1/(1+ math.exp(-(a*v)))))
*******************************************
```

Fonte: Teixeira, 2021.

Em alguns casos, é desejado que a saída seja em intervalos de -1 a 1, e não somente de 0 a 1 (caso da função sigmoide). Para tanto, em uma função sigmoide, é possível utilizar a função *tangente hiperbólica*, como apresentado na **Equação 7**.

Equação 7 – Tangente Hiperbólica

$$\varphi(v) = \tanh(v)$$

Para desenvolvermos o código da Tangente Hiperbólica em Python, utilizaremos também a função “math”, que torna o seu cálculo relativamente fácil, sendo necessário somente o uso de uma função existente na biblioteca. O código 4 apresenta a sua implementação. Lembrando que o valor de v é fornecido pela **Equação 3**.

Código 4 – Exemplo de implementação da Função de ativação Tangente Hiperbólica em Python

```
*******************************************
import math
def FuncaoTangenteHiperbolica(v):
    return (math.tanh(v))
*******************************************
```

Fonte: Teixeira, 2021.

3.4 Função ReLu

Esta função de ativação, juntamente com a Softmax, não estão presentes em nosso livro-base (Haykin, 2007), no entanto, são as funções mais amplamente utilizadas nos dias atuais. Uma justificativa para este fato é que a sua derivada é simples, facilitando a propagação do erro (próximo tema). Dessa forma, a Equação 8 apresenta a função de ativação ReLu. Basicamente, ela retorna 0 se o valor for negativo, e o próprio valor caso seja positivo. Podemos programá-la em Python sem dificuldades, utilizando a função "`match.max`" que irá retornar o maior valor entre 0 e o valor de X.

Equação 8 – Função de ativação ReLu

$$f(x) = \max(0, x)$$

3.5 Função Softmax

Também é uma função sigmoide muito utilizada na camada de saída da rede. Ela converte as saídas de cada classe em valores de 0 a 1, que correspondem à porcentagem de certeza com que essa classe está ou não ativa. Se na saída de sua rede existem dez classes, por exemplo, cada classe com a função de ativação Softmax, a que apresentar o maior valor terá a maior probabilidade de ser a correta. Dessa forma, o dado de entrada será classificado para essa classe. A função Softmax pode ser vista na Equação 9.

Equação 9 – Função de ativação Softmax

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

3.6 Conclusão

Neste momento, você conhece as principais funções de ativação utilizadas em um neurônio artificial, bem como sabe que o neurônio artificial é composto pela somatória dos pesos sinápticos (W_{kj}) vezes os valores de entrada (Xj) adicionados a uma função de ativação (δ). "E o que isso significa?" Imagine que você treinou uma RNA para identificar se um número é ímpar ou par. A rede

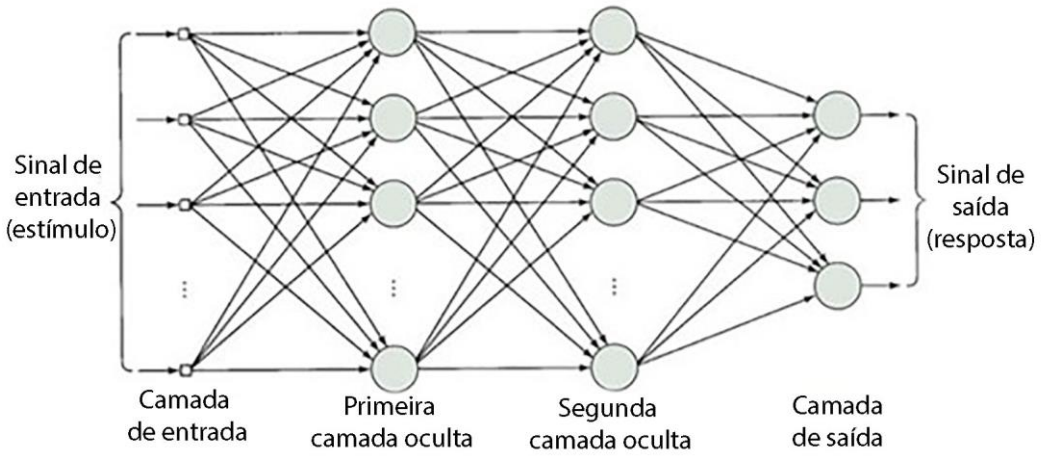
é composta por vários neurônios, além disso, em sua saída, existe um neurônio que ativa quando o número é ímpar, e outro que ativa quando o número é par. Dessa forma, os valores de W juntamente com o bias são escolhidos de forma que só ativem o neurônio certo de acordo com o valor de entrada.

A pergunta que fica agora é: como escolher os valores de W? Esses valores são escolhidos no processo de **treinamento** da RNA. Portanto, o processo de treinamento consiste em chegar aos melhores valores de W, que apresentam a maior quantidade de acertos em nossa rede. Vamos estudar um pouco sobre isso no próximo tema!

TEMA 4 – REDES NEURAIS ARTIFICIAIS E O PROCESSO DE APRENDIZAGEM

Até agora, vimos o funcionamento de um neurônio artificial isolado. Quando utilizamos uma rede para classificar problemas lineares e não lineares, normalmente é realizada uma combinação de vários neurônios para que seja possível atingir a generalização necessária. Temos um exemplo de uma rede com vários neurônios na figura 5, em que cada “bola” representa um neurônio apresentado na figura 2.

Figura 5 – Exemplo de uma RNA composta por vários neurônios e múltiplas camadas



Fonte: Haykin, 2001.

Veja que a rede é composta por várias camadas, cada uma composta por vários neurônios. Cada neurônio realiza os cálculos apresentados no tema anterior, logo, a saída y de um neurônio se torna a entrega x de sua camada seguinte. Essa rede é conhecida como Perceptrons de Múltiplas Camadas, ou

MLP. O objetivo é realizar o aprendizado de forma que os valores de W sejam treinados para que sejam genéricos o suficiente para a rede conseguir identificar o padrão nos dados de entrada.

Existem diferentes algoritmos de aprendizado para RNAs de múltiplas camadas. Nesta aula, vamos nos concentrar no mais utilizado neste tipo de rede, o **Backpropagation**.

4.1 Backpropagation: conceito

Como vimos até aqui, quando a rede está corretamente treinada, os dados navegam em sentido único: da camada de entrada, passam pelas camadas ocultas, e, então, vão em direção às camadas de saída. O algoritmo de treinamento Backpropagation tem como proposta navegar no sentido contrário da rede, com a finalidade de ajustar os pesos sinápticos (W) de acordo com o erro calculado.

Vamos começar a entender o processo. O primeiro detalhe é que se trata de uma técnica de aprendizagem supervisionada (com um professor), isto significa que para conseguirmos treinar a nossa rede, é preciso que tenhamos um conjunto de dados para treinamento e validação. Esse conjunto de dados deve ser composto por entradas e saídas, ou seja, precisamos de um conjunto de dados de entrada (X), em que temos conhecida a saída/label (y). Durante o treinamento, os pesos de W serão ajustados de acordo com o erro entre o (y) obtido de nossa rede e o objetivo real desejado (d) conhecido desejado.

Vamos chamar o valor desejado para os dados n de d(n). Dessa forma, podemos calcular o **Erro instantâneo**, onde (e_j) representa o erro no neurônio de saída (j).

Equação 10 – Cálculo do Erro Instantâneo

$$e_j(n) = d_j(n) - y_j(n)$$

Temos o erro instantâneo de um único neurônio de saída. Podemos, agora, calcular a **energia total do erro**, que é apresentada pela Equação 11, onde C representa todos os neurônios da camada de saída. O cálculo consiste na somatória de todos os erros elevados ao quadrado, multiplicado por ½.

Equação 11 – Cálculo da Energia do Erro

$$E(n) = \frac{1}{2}\sum_{j \in C} e_j^2(n)$$

Agora, podemos calcular o Erro Quadrático Médio, como apresentado na **Equação 12**. O (n) se refere ao tamanho do conjunto de treinamento. Veja que o Erro Quadrático Médio é calculado no fim de um ciclo de treinamento, onde todos os conjuntos passam pela rede.

Equação 12 – Cálculo do Erro Quadrático Médio

$$E_{med} = \frac{1}{N}\sum_{n_{norm}=1}^{N} E(n_{norm})$$

Agora, devemos ajustar os parâmetros. É utilizado o **gradiente descendente** da energia total do erro para encontrar o erro mínimo. A correção de (W_{ji}) é proporcional à derivada parcial, como apresentado na Equação 13.

Equação 13 – Cálculo da derivada parcial

$$\Delta w_{ji}(n) = -\eta \frac{\partial E(n)}{\partial w_{ji}(n)}$$

Podemos reduzir a equação de forma a obter o resultado apresentado na Equação 14, onde δ representa a derivada presente na Equação 15.

Equação 14 – Regra delta para ajustes de pesos

$$\Delta w_{ji}(n) = \eta\, \delta_j(n)\, o_i(n)$$

Equação 15 – Gradiente local

$$\delta_j(n) = -\frac{\partial E(n)}{\partial u_j(n)}$$

Dessa forma, o algoritmo ajusta os pesos de (W) utilizando a Equação 16, se estiverem na camada de saída; e utilizando a Equação 17, onde u_j corresponde ao valor presente na equação 3, se estiverem na camada oculta.

Equação 16 – Ajuste final para neurônios em camadas de saída

$$\delta_j(n) = \varphi'\left(u_j(n)\right)\sum_k \delta_k(n) w_{kj}(n)$$

Equação 17 – Ajuste final para neurônios em camadas ocultas

$$\delta_j(n) = \varphi'\left(u_j(n)\right) e_j(n)$$

Onde a derivada da função de ativação depende da função utilizada pelo neurônio em questão. Se a função de ativação for sigmoide, por exemplo, a sua derivada é definida pela equação 18.

Equação 18 – Derivada da função de ativação sigmoide

$$\varphi'\left(u_j(n)\right) = a\, y_j(n)\left[1 - y_j(n)\right]$$

Este processo se repete até que algum critério de convergência seja estabelecido. O processo costuma ser lento e pode enfrentar problemas, como a presença de **ótimos locais** (estamos buscando o ótimo global).

O treinamento da rede é um passo importante e fundamental para o seu funcionamento e pode aparentar ser confuso para quem o olha pela primeira vez. Sugerimos a leitura e estudo desta seção com calma. No próximo tema, iremos praticar um pouco.

4.2 Backpropagation: conclusão

O algoritmo Backpropagation busca minimizar uma função de custo baseada no erro instantâneo. O algoritmo apresentado aqui é considerado o algoritmo clássico de treinamento e aprendizado em uma RNA, responsável por alavancar a tecnologia.

No entanto, existem variantes do algoritmo apresentado nesta seção. É possível utilizar diferentes funções de custos, baseados em outros fatores além do erro instantâneo. Existem também técnicas de treinamento com validação cruzada para ajustes dos pesos. Contudo, todas as técnicas recentes são derivadas das apresentadas nesta seção, com modificações que buscam otimizar o processo de aprendizado. Existem diferentes técnicas de otimização para tornar o treinamento mais rápido, chegando de forma eficiente até o mínimo global, reduzindo o erro e aumentando a acurácia.

TEMA 5 – VAMOS CRIAR A NOSSA PRIMEIRA REDE NEURAL ARTIFICIAL

Neste tema, iremos praticar os conceitos vistos até então, de forma leve e prática. Utilizaremos bibliotecas prontas para o Python, de modo que poderemos nos preocupar apenas em definir o tipo de rede, as quantidades de camadas e as funções de ativação, sem a necessidade de criarmos todo o código necessário para o treinamento do mesmo. Vamos começar apresentando a biblioteca.

5.1 Ambiente de desenvolvimento

No exemplo que iremos desenvolver a seguir, iremos utilizar a biblioteca “TensorFlow 2” juntamente com o “keras”. Para instalar essa biblioteca via pip, digite:

pip install tensorflow

A configuração do computador utilizado para esse experimento é:

- Ubuntu 20.04;

- Python 3.8.10;
- TensorFlow 2.7.0;
- Keras 2.7.0.

5.2 Introdução aos frameworks

Agora, vamos praticar um pouco! Você irá perceber que para criar e treinar uma MLP, não é preciso ser um matemático brilhante ou um estatístico. Você precisa conhecer os conceitos para poder projetar a sua rede da melhor forma possível, a fim de cumprir o objetivo que você deseja alcançar. Normalmente, a MLP é utilizada no final de uma arquitetura de Deep Learning para realizar a classificação, podendo ser retreinada para a aplicação de transferência do aprendizado (teremos uma aula somente deste assunto), em que uma arquitetura já treinada para identificar um tipo de situação tenha apenas o seu classificador retreinado para identificar outro tipo de situação.

Outro ponto importante é que você não deve ser refém de uma tecnologia em específico. Além do TensorFlow, existem diversos outros frameworks para a criação e treinamento de redes. Basicamente todos utilizam o mesmo conceito, o que muda de um para o outro é a maneira como as funções são chamadas/implementadas. Em um Framework, por exemplo, o comando rede.add() pode significar a adição de uma nova camada na rede, enquanto em outro, pode ser preciso definir toda a rede de uma única vez em um arquivo json, mas em ambos os casos, a rede terá o mesmo formato.

5.3 Importando as bibliotecas

Para este experimento, vamos utilizar as bibliotecas do Keras datasets.mnist, models.Sequential, layers.Dense e utils. O TensorFlow, assim como o Keras, são projetados para operar diferentes técnicas de aprendizado de máquina, desde uma simples MLP (como iremos realizar agora), até técnicas avançadas como CNN com múltiplas camadas convolucionais. Nesta aula, utilizaremos somente redes sequenciais e densas, ou seja, totalmente conectadas em sequência. O código 5 apresenta a importação dessas bibliotecas. Também será utilizada a biblioteca “matplotlib” para a criação de gráficos.

Código 5 – Importação das bibliotecas utilizadas para a criação e treinamento de uma MLP em Python com Keras e TensorFlow

```
**********
import tensorflow
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
**********
```

Fonte: Teixeira, 2021.

5.4 Dataset

O Keras possui um conjunto de datasets prontos para podermos praticar. Este fato é interessante, pois a maior dificuldade em treinar uma MLP ou qualquer outro algoritmo supervisionado é o dataset. O conjunto de dados precisa ser grande e genérico o suficiente para que a rede não se torne “viciada” a um tipo de dado específico. Para todo conjunto de entrada, é preciso ter um conjunto de saída (objetivo) conhecido.

O dataset que iremos utilizar neste experimento é o mnist (https://www.tensorflow.org/datasets/catalog/mnist). Para importarmos o dataset para o nosso código, iremos utilizar a função “mnist.load_data()”. Esta função retorna o dataset já dividido em dados de treino (60 mil) e dados de validação (1 mil). O código a seguir realiza esse processo.

Código 6 – Importação do dataset mnist, dividido em dados de treinamento e de teste

```
**********
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
X_train = X_train.reshape(X_train.shape[0], feature_vector_length)
X_test = X_test.reshape(X_test.shape[0], feature_vector_length)
**********
```

Fonte: Teixeira, 2021.

Repare que para converter a imagem em um vetor que será adicionado ao nosso MLP, foi utilizada a função “reshape”. Dessa forma, a imagem é convertida do formato original, que é uma matriz 28x28, para um vetor de tamanho 784. Antes de continuarmos, vamos dar uma olhada nos dados. O código 7 executa a visualização de alguns dados (10), enquanto a Figura x apresenta o resultado do código. O mnist é composto por dez classes, sendo elas imagens dos números de 0 a 9, conforme apresentado na figura 6.

Código 7 – Visualização dos dados de treinamento

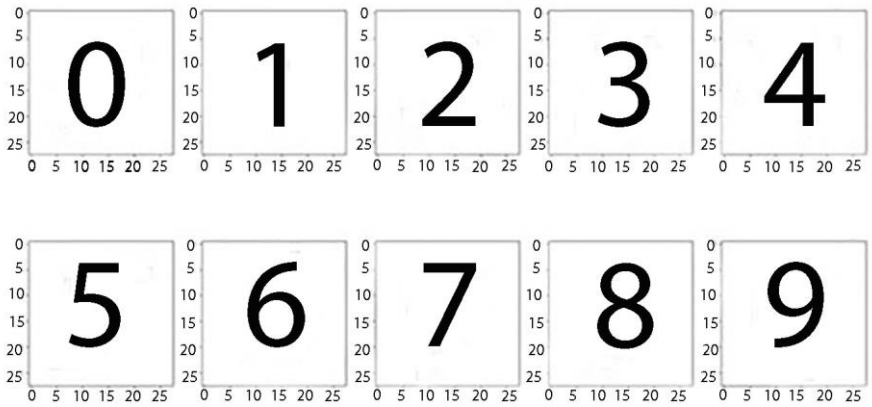
```
**********************************************

fig, ax = plt.subplots(2,5)
ax[0,0].imshow(X_train[1].reshape(28,28), cmap=plt.cm.binary)
ax[0,1].imshow(X_train[6].reshape(28,28), cmap=plt.cm.binary)
ax[0,2].imshow(X_train[5].reshape(28,28), cmap=plt.cm.binary)
ax[0,3].imshow(X_train[7].reshape(28,28), cmap=plt.cm.binary)
ax[0,4].imshow(X_train[2].reshape(28,28), cmap=plt.cm.binary)
ax[1,0].imshow(X_train[0].reshape(28,28), cmap=plt.cm.binary)
ax[1,1].imshow(X_train[13].reshape(28,28), cmap=plt.cm.binary)
ax[1,2].imshow(X_train[15].reshape(28,28), cmap=plt.cm.binary)
ax[1,3].imshow(X_train[17].reshape(28,28), cmap=plt.cm.binary)
ax[1,4].imshow(X_train[4].reshape(28,28), cmap=plt.cm.binary)
plt.show()

**********************************************
```

Fonte: Teixeira, 2021.

Figura 6 – Visualização dos dados de treinamento

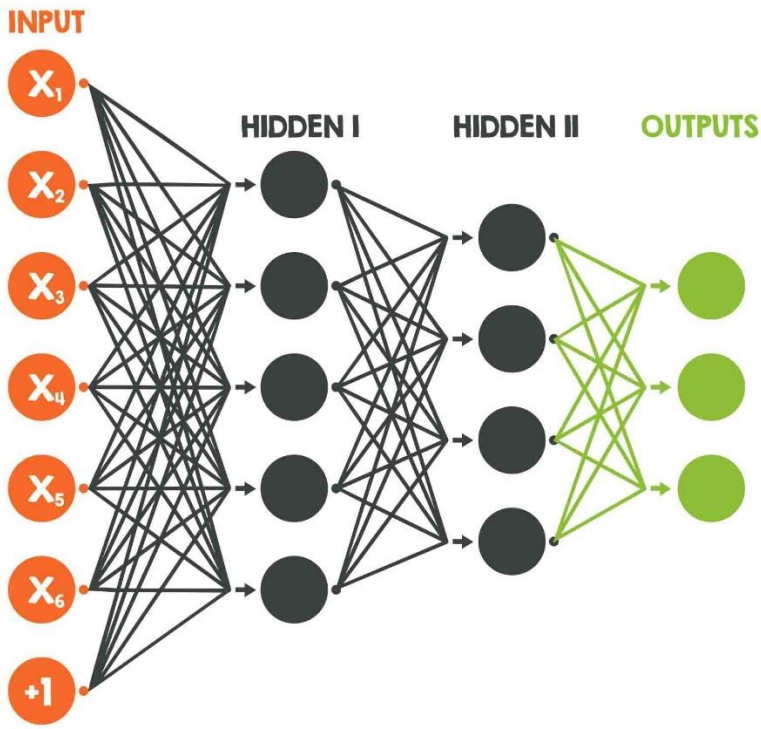


Fonte: Teixeira, 2021.

5.5 Criando a arquitetura da rede

Como vimos, uma MLP normalmente é formada por uma camada de entrada, uma ou mais camadas de neurônios ocultos e uma camada de saída. Temos 784 dados de entrada e dez classes (de 0 a 9) como saída. Dessa forma, para esse exemplo, iremos criar uma rede com 350 neurônios na primeira camada, 50 neurônios na camada oculta e 10 neurônios da camada de saída. A **Figura 7** ilustra a arquitetura pretendida.

Figura 7 – Arquitetura de nossa MLP proposta



Créditos: ShadeDesign/Shutterstock.

Agora, vamos para o código e criar a nossa rede. Primeiro, definimos o tipo do nosso modelo, que, nesse caso, é sequencial. Em seguida, adicionamos as camadas e as funções de ativação. Nesse exemplo, iremos utilizar as funções de ativação “relu” para as camadas ocultas e “softmax” para a camada de saída, o código pode ser visto no Código 8.

Código 8 – Criação da arquitetura de nossa MLP

```
**********
modelo = Sequential()
modelo.add(Dense(350, input_shape=input_shape, activation='relu'))
modelo.add(Dense(50, activation='relu'))
modelo.add(Dense(num_classes, activation='softmax'))
modelo.summary()
**********
```

Fonte: Teixeira, 2021.

5.6 Executando o treinamento

Tendo os dados separados e a arquitetura da rede criada, podemos, então, dar início ao processo de treinamento. Primeiro, devemos definir os parâmetros de treinamento. No nosso caso, iremos utilizar a função de erro: “erro quadrático médio” e o otimizador “Adam”. Como métrica de desempenho, iremos utilizar a acurácia. O código 9 apresenta como definir essas configurações.

Código 9 – Definindo os parâmetros de treinamento

```
**********
modelo.compile(loss='mean_squared_error', optimizer='adam', metrics=['accuracy'])
**********
```

Fonte: Teixeira, 2021.

Agora, podemos dar início ao treinamento. É possível definir alguns parâmetros na hora de iniciarmos o treinamento, os principais são o número de épocas em que o algoritmo de treinamento será executado, se ele será dividido em “batch” e o valor de cada “batch”. Para esse exemplo, deixaremos um número de 20 épocas, divididos em batches de tamanho 100. Outro detalhe são os dados de validação durante o treinamento, pois o parâmetro “validation_split” pode ser utilizado para dividir os dados de treinamento em dados de validação. No nosso exemplo, 20% dos dados de treinamento serão utilizados para a validação, como apresentado no código 10.

Os dados de treinamento serão armazenados em uma variável chamada “historico” para que possamos analisar os seus resultados posteriormente.

Código 10 – Executando o treinamento

```
**********************************************
historico = modelo.fit(X_train, Y_train, epochs=20, batch_size=100, validation_split=0.2)
**********************************************
```

Fonte: Teixeira, 2021.

5.7 Verificando os resultados

Agora, temos uma rede treinada. A variável “modelo” agora consiste em uma MLP treinada, na qual podemos adicionar uma imagem do conjunto “mnist”, e ela irá tentar informar qual é o número presente na imagem. Vamos validar a nossa rede com o nosso conjunto de teste e verificar o resultado.

Código 11 – Aplicando a rede treinada ao conjunto de testes

```
**********************************************
teste_resultados = modelo.evaluate(X_test, Y_test, verbose=1)
print(f'Resultado nos testes - Erro: {test_results[0]} - Acuracia: {teste_resultados[1]}%')
**********************************************
```

Fonte: Teixeira, 2021.

No final da execução deste trecho do código, teremos como saída o erro e a acurácia de nossa rede, calculado com base no conjunto de testes. Lembrando que o conjunto de testes também é um conjunto com seu “label”, ou seja, seu objetivo, conhecido. Dessa forma, a acurácia e o erro são calculados adicionando a imagem à rede e verificando se ela acertou ou não o label desejado. No nosso caso, teremos algo como:

```
**********************************************
$ Resultado nos testes - Erro: 0.003299425821751356 - Acuracia:
0.9797000288963318%
**********************************************
```

Isso significa que a nossa rede possui uma acurácia de 97,9%. Nem sempre conseguimos números tão bons. O que acontece aqui é que se trata de um dataset simples, com tamanho reduzido (28x28) e poucas classes (10). Vamos analisar um pouco mais o nosso treinamento. Como todos os dados foram salvos na variável “historico”, é possível visualizar a curva de aprendizagem de nossa rede, como apresentado no código 12 e na **Figura 8**.

Código 12 – Exibição do gráfico de erro e acurácia do treinamento

```
**********************************************
fig, ax = plt.subplots(1,2, figsize=(16,8))
ax[0].plot(historico.history['loss'], color='b', label="Erro: Treinamento")
ax[0].plot(historico.history['val_loss'], color='r', label="Erro: Validação",axes =ax[0])
legend = ax[0].legend(loc='best', shadow=True)
ax[1].plot(historico.history['accuracy'], color='b', label="Acurácia: Treinamento")
ax[1].plot(historico.history['val_accuracy'], color='r',label="Acurácia: validação")
legend = ax[1].legend(loc='best', shadow=True)
plt.show()
**********************************************
```

Fonte: Teixeira, 2021.

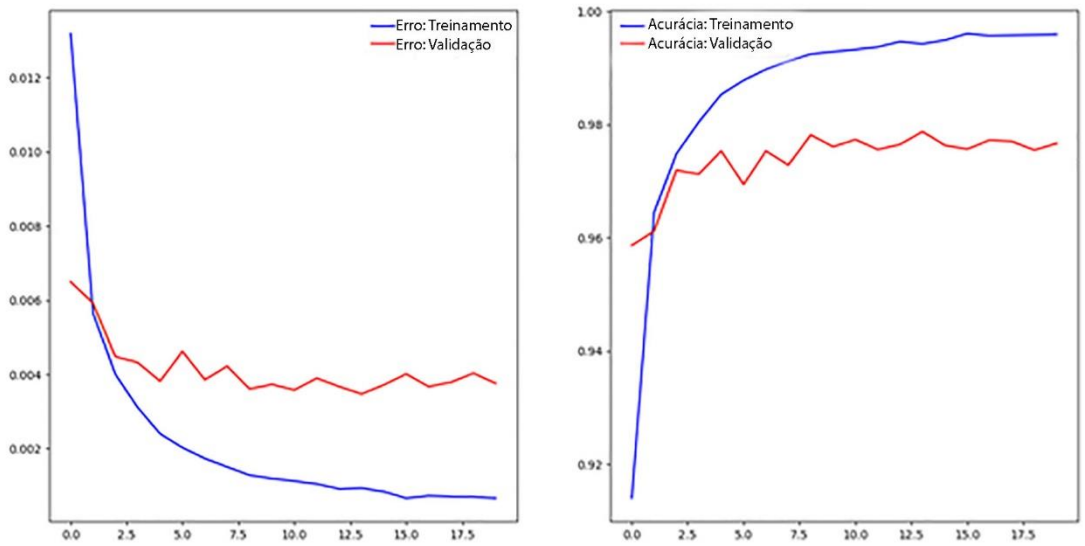


Figura 8 – Gráfico de erro e acurácia durante o treino

Fonte: Teixeira, 2021.

Analisando o gráfico, é possível verificar que o erro tende a ir a zero enquanto a acurácia tende a ir a 100. No entanto, observamos que os dados de

validação sofrem leves alterações, além disso, desde a primeira interação, eles já se encontram perto do resultado final, com acurácia acima de 96. Como mencionado anteriormente, esse é um exemplo simples, sendo necessário pouco poder computacional para ser resolvido. Não espere resultados tão impressionantes em problemas complexos.

5.8 Código completo

Código 12 – Código completo

```
                                        ******************************************
import tensorflow
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
feature_vector_length = 784
num_classes = 10
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
X_train = X_train.reshape(X_train.shape[0], feature_vector_length)
X_test = X_test.reshape(X_test.shape[0], feature_vector_length)
print("Quantidade de elementos de treino: {}". format(len(X_train)))
print("Quantidade de elementos de teste: {}". format(len(X_test)))
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
Y_train = to_categorical(Y_train, num_classes)
Y_test = to_categorical(Y_test, num_classes)
input_shape = (feature_vector_length,)
print(f'Feature shape: {input_shape}')
Imprime a figura
fig, ax = plt.subplots(2,5)
ax[0,0].imshow(X_train[1].reshape(28,28), cmap=plt.cm.binary)
ax[0,1].imshow(X_train[6].reshape(28,28), cmap=plt.cm.binary)
ax[0,2].imshow(X_train[5].reshape(28,28), cmap=plt.cm.binary)
ax[0,3].imshow(X_train[7].reshape(28,28), cmap=plt.cm.binary)
ax[0,4].imshow(X_train[2].reshape(28,28), cmap=plt.cm.binary)
ax[1,0].imshow(X_train[0].reshape(28,28), cmap=plt.cm.binary)
ax[1,1].imshow(X_train[13].reshape(28,28), cmap=plt.cm.binary)
```

```
ax[1,2].imshow(X_train[15].reshape(28,28), cmap=plt.cm.binary)
ax[1,3].imshow(X_train[17].reshape(28,28), cmap=plt.cm.binary)
ax[1,4].imshow(X_train[4].reshape(28,28), cmap=plt.cm.binary)
plt.show()
print('Label: {}'.format(Y_train[1]))
modelo = Sequential()
modelo.add(Dense(350, input_shape=input_shape, activation='relu'))
modelo.add(Dense(50, activation='relu'))
modelo.add(Dense(num_classes, activation='softmax'))
modelo.summary()
modelo.compile(loss='mean_squared_error', optimizer='adam', metrics=['accuracy'])
historico = modelo.fit(X_train, Y_train, epochs=20, batch_size=100, validation_split=0.2)
teste_resultados = modelo.evaluate(X_test, Y_test, verbose=1)
print(f'Resultado nos testes - Erro: {teste_resultados[0]} - Acuracia: {teste_resultados[1]}%')
fig, ax = plt.subplots(1,2, figsize=(16,8))
ax[0].plot(historico.history['loss'], color='b', label="Erro: Treinamento")
ax[0].plot(historico.history['val_loss'], color='r', label="Erro: Validação",axes =ax[0])
legend = ax[0].legend(loc='best', shadow=True)
ax[1].plot(historico.history['accuracy'], color='b', label="Acurácia: Treinamento")
ax[1].plot(historico.history['val_accuracy'], color='r',label="Acurácia: Validação")
legend = ax[1].legend(loc='best', shadow=True)
plt.show()
```

****

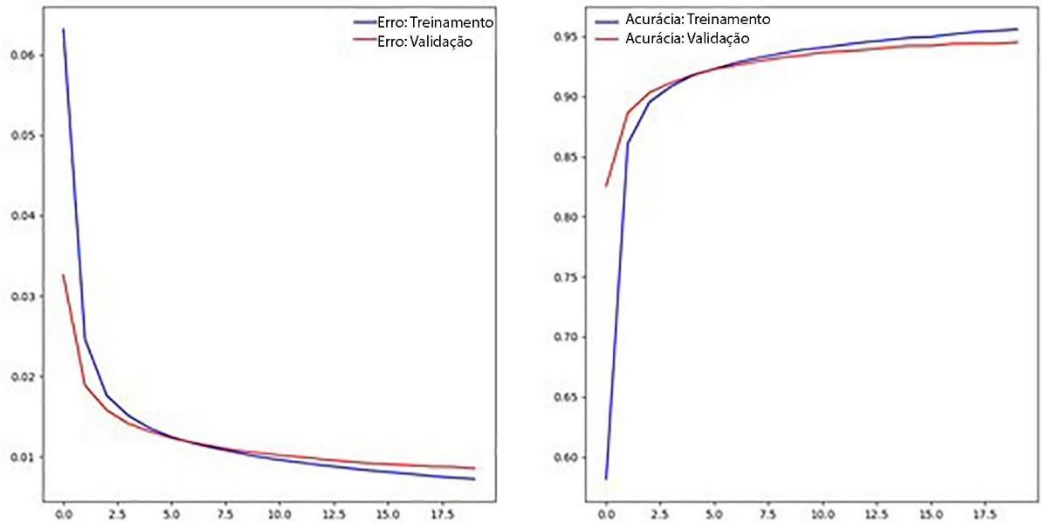
Fonte: Teixeira, 2021.

NA PRÁTICA

Agora que criamos a nossa primeira MLP, sugerimos que modifique os parâmetros da rede, como quantidade de neurônios por camadas, quantidade de épocas, tamanho do batch, percentual de dados de validação, funções de ativação, entre outros, para que possa verificar os efeitos sobre o treinamento e a acurácia da rede.

Por exemplo, o que acontece se modificarmos a rede para uma única camada oculta de tamanho 30, uma camada de saída de tamanho 10, batch de tamanho 1000 e dados de validação em 30%, com 20 épocas? O resultado pode ser observado na Figura 9.

Figura 9 – Resultado após modificação da arquitetura da rede e para metrôs de treinamento



Aparentemente, o resultado obtido foi melhor, mas quando verificamos com os dados de teste, a rede apresenta uma acurácia de 94%, ou seja, menor que a anterior. Os dados de treinamento podem “enganar”, pois a rede pode se tornar “viciada” caso execute o treinamento por muito tempo sempre com os mesmos dados. Por isso, é interessante observar a acurácia e o erro com dados de teste não utilizados durante o treinamento.

FINALIZANDO

Nesta aula, estudamos os conceitos de redes neurais artificiais. Vimos que sua inspiração vem da biologia, conhecemos a sua base matemática e estudamos o processo utilizado para realizar o seu aprendizado. Para finalizar, realizamos um experimento prático com a linguagem de programação Python e a biblioteca TensorFLow e Keras. Percebemos que a criação e treinamento de uma rede é um processo fácil, que exige poucas linhas de código, no entanto, a sua teoria é complexa, sendo importante entendermos ela para que possamos tomar a melhor decisão na hora de escolhermos os parâmetros de nossa rede. Posteriormente, iremos estudar as redes convulsionais.

REFERÊNCIAS

HAYKIN, S. **Redes neurais**: princípios e prática. Porto Alegre: Bookman, 2007.

Y CAJÁL, S. R. **Histology of the nervous system**. Oxford: Oxford University Press, 1995.