UNIVERSIDADE REGIONAL INTEGRADA DO ALTO URUGUAI E DAS MISSÕES
CAMPUS DE ERECHIM
DEPARTAMENTO DE ENGENHARIAS E CIÊNCIA DA COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

VINICIUS LOEBLER DUFLOTH

APLICAÇÃO DE MACHINE LEARNING NA CADEIA DE SUPRIMENTOS: UTILIZANDO APRENDIZAGEM DE LONGO PRAZO PARA PREVISÃO DE DEMANDA

ERECHIM - RS
2020

VINICIUS LOEBLER DUFLOTH

APLICAÇÃO DE MACHINE LEARNING NA CADEIA DE SUPRIMENTOS: UTILIZANDO APRENDIZAGEM DE LONGO PRAZO PARA PREVISÃO DE DEMANDA

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do grau de Bacharel, Departamento de Engenharias e Ciência da Computação da Universidade Regional Integrada do Alto Uruguai e das Missões Campus de Erechim.

Orientador: Prof. Me. Marcos André Lucas

ERECHIM - RS
2020

RESUMO

No mercado conectado em que vivemos, as cadeias de suprimentos possuem um papel essencial e, para garantir sua produtividade, é muito importante manter um planejamento constante. Uma informação fundamental para esse planejamento é o conhecimento da demanda esperada para o curto e médio-prazo. O presente trabalho apresenta uma abordagem de previsão utilizando *Machine Learning* para buscar uma maior acurácia nessas previsões. Foram aplicados sobre o mesmo conjunto de dados alguns dos métodos mais populares utilizados hoje no meio logístico e o modelo de Rede Neural construído utilizando a arquitetura de *Long Short-Term Memory*. Após aplicados sobre o conjunto de dados, foi medida a acurácia com as métricas de Erro Médio Absoluto e Raiz do Erro Quadrático Médio. Foram apresentado gráficos, códigos e resultados dessas previsões, levando a acreditar que a Regressão Linear Multivariada é o método que traz resultados mais satisfatórios na resolução deste problema.

Palavras-chave: Aprendizagem de Máquina. Cadeia de Suprimentos. Previsão de Demanda. Arquitetura de Memória de Longo Prazo. Regressão Linear.

ABSTRACT

In the connected market we live in, supply chains play an essential role, and to guarantee their high productivity it's very important to keep constant planning. A key information for this planning is the knowledge of demand forecast for the short and medium term. This work presents a forecasting approach using Machine Learning to seek greater accuracy in these predictions. Under the same data set, some purely common methods used today in logistics and the Neural Network model built using the Long Short-Term Memory architecture were be applied. After being applied to the dataset, the accuracy of each method was then measured using the Mean Absolute Error and the Rooted Mean Squared Error. Plots, codes and results of those predictions where presented, and leading to believe that Multivariate Linear Regression is the method that brings the most satisfactory results on the solution to this problem.

Key-words: Machine Learning. Supply Chain. Demand Forecasting. Long Short-Term Memory. Linear Regression.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Existe uma grande importância das cadeias de suprimentos em nossa economia, pois todo negócio necessita de produtos que são produzidos por certos fornecedores, que por sua vez, obtém suas matérias primas de outros fornecedores, gerando assim uma vasta cadeia de suprimentos que culmina em um produto final para atender a demanda de um cliente.

Planejar decisões futuras sendo qualquer uma dessas entidades é uma tarefa complexa que só pode ser bem executada quando se tem uma boa perspectiva da demanda futura. Sendo assim, entendemos que a acuracidade na previsão de demanda futura é crucial para qualquer entidade envolvida em uma cadeia, pois quanto mais próximo da realidade estiver esta previsão, mais otimizado será o planejamento e o atendimento da demanda do cliente, rendendo também maiores lucros à entidade.

Embora já existam técnicas comummente usadas que podem fazer essa previsão, é possível que utilizando *Machine Learning* seria possível obter resultados melhores devido ao modo como operam. Como elas criam os próprios modelos estatísticos a partir dos conjuntos de dados reais que são aplicados a elas durante treinamento, é plausível que uma Inteligência Artificial (IA) crie um modelo que mais se aproxima da realidade em comparação aos existentes. Além disso, as empresas do ramo logístico, em sua maioria, já possuem grandes coleções de dados armazenados, fato que facilita o treinamento nesse setor. Portanto, este trabalho busca entender se o uso de Aprendizagem de Máquina na previsão de demanda futura traz resultados satisfatórios e pode trazer ganhos ao setor.

O trabalho estrutura-se da seguinte forma. No Capítulo 2 é apresentado um panorama da previsão de demanda no meio logístico e o método escolhido dentre os populares, assim como sua fórmula. No Capítulo 3 é realizado um panorama geral do campo de Inteligência Artificial, afunilando para as áreas pertinentes a este trabalho. Nele, passamos pelo método de Regressão Linear, sua teoria e sua fórmula, passamos pelas Redes Neurais, entrando em detalhes sobre suas arquiteturas, com foco na LSTM e falamos sobre todos os hiper-parâmetros envolvidos com a criação de um modelo, e como escolher os valores corretos.

Por fim, no Capítulo 4 falamos sobre uma importante área da Aprendizagem de Máquina, a Engenharia de Dados, que mostra como preparar os dados para serem utilizados como entrada em seus modelos. No Capítulo 5 abordamos todas as bibliotecas e ferramentas utilizadas com exemplos práticos de código. E por fim, no Capítulo 6 colocamos toda a teoria e ferramentas vistos até então na prática, juntando dados e gráficos que podem nos ajudar a responder a pergunta levantada nessa introdução. Essa resposta está no Capítulo 7, onde é feito um recapitulamento do trabalho e conclue-se o que foi levantado.

2 Previsão de Demanda na Cadeia de Suprimentos

Uma boa definição para uma cadeia de suprimentos pode ser encontrada no trabalho de Bousqaoui, Achchab e Tikito (2019), que a definem como um conjunto de entidades (fornecedores, fabricantes, distribuidores, armazéns e etc.) em que existe fluxo de materiais, informações e finanças entre si, possuindo como objetivo final atender a demanda de certo cliente.

Pensando assim, pode-se compreender porque Ballou (2006) chegou a dizer que torna-se fundamental para a estrutura geral de qualquer negócio neste ramo, possuir algum método de previsão de demanda futura. Isso ocorre pois esse dado se torna a entrada básica de informação para o planejamento estratégico e controle de áreas vitais da empresa, como a Logística, o Marketing, a Produção e o Financeiro. Diz também que as previsões de curto prazo auxiliam no controle de estoque, programação de embarques, planejamento de carregamento de armazéns e semelhantes.

Embora um grande avanço na previsão de demanda poderia ocorrer pelo compartilhamento de informações entre compradores e fornecedores, por diversos motivos isso na prática acaba pouco ocorrendo. Muitas vezes existem desconfianças em compartilhar essas informações pelo medo de outras empresas rivais conseguirem alguma vantagem competitiva caso encontrem informações atreladas a planos de crescimento futuro, mudança de ramo, criação de novos produtos, entre outros.

Foi assim que em seu trabalho, Carbonneau, Vahidov e Laframboise (2007) justificam a busca por outros métodos que possam trazer previsões mais exatas para cada entidade presente na cadeia de suprimentos, mesmo sem precisar diretamente dos dados de outra empresa, e que isso pode ocorrer utilizando métodos de Aprendizagem de Máquina.

2.1 Estratégias Clássicas de Previsão de Demanda

Segundo Ballou (2006) as técnicas mais utilizadas hoje na previsão de demanda no meio logístico, são a Média Móvel Exponencial (MME, também conhecida como Ponderação Exponencial), a Decomposição das Séries de Tempo e a Regressão Linear Múltipla. Essas são citadas como as mais comuns, mas dentre várias outras, Redes Neurais também são citadas por ele como já utilizadas.

Como em seu trabalho, Carbonneau, Vahidov e Laframboise (2007) demonstraram que as previsões com Médias Móveis e Regressão Linear foram mais acertadas do que utilizando o método *ARIMA* (um tipo de Decomposição de Séries de Tempo), e sendo seu conjunto de dados semelhante ao deste trabalho, os métodos de decomposição aqui foram deixados de lado, buscando dar prioridade às previsões com Redes Neurais que serão abordadas no Capítulo 3.

Mais detalhes sobre a MME estão descritos na Subseção 2.1.1 e sobre a Regressão Linear na Seção 3.2. Também será utilizado como *benchmark* o método de Previsão Ingênua

(do inglês, *Naïve Method*) que consiste em apenas utilizar o mês anterior, sem nenhum tipo de tratamento ou cálculo sobre os dados. Esse método é utilizado como *benchmark* pois se algo mais complexo resulta em previsões iguais ou semelhantes, deve ser descartado, pois sempre se mantém o método mais simples dadas performances semelhantes.

2.1.1 Média Móvel Exponencial

É um tipo de média móvel, em que observações mais recentes recebem um peso maior do que as mais antigas. Ela é dada pela fórmula

$$F_{t+1} = \alpha A_t + (1 - \alpha) F_t$$

onde

t = período de tempo atual.

α = constante da ponderada exponencial

A_t = demanda no período t

F_t = previsão do período anterior

F_{t+1} = previsão para o próximo período

O valor de α geralmente fica entre 0,01 a 0,3 que são valores já demonstrados como baixos o suficiente para proporcionarem previsões estáveis, não muito influenciadas por aleatoriedades. A escolha de um valor dentro deste intervalo deve ser determinado por alguém com alto grau de conhecimento sobre o modelo sendo desenvolvido e o momento econômico em que se vive, já que de acordo com Ballou (2006), esta variável é que define a velocidade com a qual se quer que o modelo reaja a mudanças, e isso pode variar de acordo com o período. Sendo assim, existe um certo grau de dependência de conhecimento de experiência humana necessária para que esse método tenha sucesso.

Porém, essa versão da Ponderação Exponencial não se corrige para a tendência, necessitando de mais algumas adições que tornam ela mais completa. Essa nova versão é representada por um conjunto de três equações, sendo elas

$$S_{t+1} = \alpha A_t + (1 - \alpha)(S_t + T_t)$$

$$T_{t+1} = \beta(S_{t+1} - S_t) + (1 - \beta) T_t$$

$$F_{t+1} = S_{t+1} + T_{t+1}$$

onde os símbolos não definidos anteriormente são

S_t = previsão inicial para o período t

T_t = tendência para o período t

β = constante ponderada da tendência

Novamente, a constante β precisa ser definida por alguém com conhecimento da demanda específica e varia entre 0,01 e 0,3.

Existe ainda uma terceira versão da Ponderação Exponencial, mas ela apenas é indicada quando os picos e vales no padrão da demanda tem um motivo conhecido e ocorrem sempre na mesma época. Além disso, deve ser utilizada quando os picos causados por fatores desconhecidos não é maior do que os causados pelos pico sazonais conhecidos. Sendo que os conjuntos de dados abordados neste trabalho não apresentam essa característica, a fórmula utilizada será esta última apresentada.

2.2 Vantagens De Previsões com Inteligência Artificial

Como afirmado por Mueller e Massaron (2006), o uso de técnicas de Inteligência Artificial (que serão abordadas no Capítulo 3), tornam-se interessante por sua capacidade de criar modelos estatísticos que são formados a partir de exemplos reais aplicados a ela. Sendo assim, é plausível que um modelo de Rede Neural, por exemplo, crie relações estatísticas sobre um modelo que se aproxime mais da realidade do que os modelos clássicos apresentados na Seção 2.1.

Além disso, como demonstrado no trabalho de Carbonneau, Vahidov e Laframboise (2008), os modelos clássicos podem introduzir o Efeito Chicote (do inglês, *bullwhip effect*, BWE) nas previsões de uma cadeia de suprimentos, enquanto que previsões lineares regressivas, como uma Rede Neural Recorrente, isso não costuma ocorrer.

O Efeito Chicote é causado quando cada membro de uma cadeia de suprimentos deriva seu padrão de demanda dos pedidos de seu parceiro imediato adicionando um índice de segurança próprio. Ao chegar no final da cadeia, pode-se ter um resultado extremamente distorcidos quando comparados com o que realmente seria necessário para suprir a necessidade dos clientes, causando incertezas, mau planejamento, e altos custos operacionais (BALLOU, 2006).

3 Inteligência Artificial

Inteligência Artificial (IA) de acordo com Russel e Norvig (2013), é um campo que busca compreender e construir entidades inteligentes, abrangendo uma grande variedade de campos da ciência, podendo ser considerado como um campo universal.

Russel e Norvig (2013) ainda dizem que existem várias definições e formas de se abordar IA dependendo da definição de sucesso que o pesquisador tem nesse campo para o que ele procura. Pode-se enxergar o trabalho nessa área como uma busca pela fiel réplica do desempenho humano ou então como uma busca pelo conceito ideal de inteligência, uma busca pela racionalidade. Racionalidade aqui sendo a ideia de que dadas as informações existentes em determinado sistema que essa inteligência atua, ela tomará a melhor decisão que conseguir.

Neste trabalho, foram utilizados algoritmos e sistemas de IA no sentido de busca dessa racionalidade. Ou seja, buscamos um sistema que dadas certas informações iniciais, conseguisse fornecer uma correta previsão para o futuro.

3.1 Aprendizagem de Máquina

Aprendizagem de Máquina (do inglês, *Machine Learning*, *ML*) teve seu início com os trabalhos de Samuel (1959) que buscava um projeto para criar uma máquina autônoma que aprendesse sozinha as regras do jogo de tabuleiro de damas. Dele originou-se a ideia de que *ML* é o campo de estudo que da aos computadores a habilidade de aprenderem sem serem explicitamente programados.

Em outras palavras, Burkov (2019) diz que esse ramo busca construir algoritmos que agem em uma coleção de exemplos de algum fenômeno, seja ele natural ou intencionalmente gerado, buscando resolver um problema prático através da construção de um modelo estatístico auto criado com base no conjunto de dados fornecido.

Essa tecnologia já é comum em várias áreas do nosso dia a dia. Todos grandes buscadores de *sites*, assistentes pessoais, e aplicações de reconhecimento de voz são exemplos de uso diários que fazemos dessas tecnologias. Para exemplificar porque essas técnicas são preferíveis, podemos utilizar o exemplo fornecido por Géron (2019) do uso de *ML* nos filtros de *spam* de *emails*.

No exemplo dele, utilizando um algoritmo programado com regras pré-definidas para categorizar um *email* como *spam* ou não, todas as regras utilizadas na identificação e categorização precisariam ser pensadas e codificadas pelos programadores desse algoritmo.

Isso seria possível e poderia funcionar bem, porém logo os criadores de emails de *spam* identificariam as regras que os bloqueavam e se adaptariam com novos modelos de *emails* que burlassem essa categorização. Os programadores então receberiam reclamações dos usuários de seu filtro, teriam de identificar esses novos padrões, realizar uma correção no programa e gerar

uma nova versão.

Em contraste, um filtro de *spam* baseado em Aprendizagem de Máquina seria desenvolvido e treinado em um conjunto de dados composto por *emails* já categorizados como *spam* ou não. Conforme os próprios usuários categorizassem novos *emails* como spam ou não, esse conjunto de dados iria aumentando dinamicamente.

Ao ser retreinado periodicamente, caso um novo tipo de *email* tivesse passado pelo seu filtro existente, ele teria sido categorizado como *spam* pelos usuários e esse novo padrão seria aprendido em novos treinos sem intervenção direta dos programadores. Da mesma forma, quando utilizado para previsão de demanda futura, é possível que novas tendências e alterações no conjunto de dados sejam adaptadas pelo algoritmo de *ML* sem necessidade de interferência direta.

Aprendizagem de Máquina é um campo muito amplo, e pode ser dividido em muitas categorias. Nas próximas seções iremos abordar essas subdivisões até chegarmos no modelo escolhido para este trabalho. Iniciaremos falando sobre seu tipo de treinamento, que pode ser categorizado em aprendizagens por reforço, supervisionadas ou não supervisionadas.

É importante ressaltar que na prática as distinções dentre essas categorias podem não ser tão nítidas, sendo necessário em alguns problemas a união de tipos diferentes de aprendizagem em um só agente em diferentes momentos.

3.1.1 Aprendizagem Por Reforço

Russel e Norvig (2013) diz que na aprendizagem por reforço o algoritmo de aprendizagem de máquina (também chamado de agente) aprende a partir de uma série de recompensas ou punições.

Geralmente esse tipo de agente identifica o ambiente em que está inserido, ou seja quais ações pode tomar, e após realizar certa ação recebe uma penalidade ou recompensa que vai dizer à ele quão eficaz ou não foi a sua ação segundo Géron (2019). Algoritmos utilizados para jogos de Xadrez ou *Go* são exemplos de uso destes agentes.

3.1.2 Aprendizagem Não Supervisionada

Na aprendizagem não supervisionada, o agente aprende padrões a partir de uma certa entrada, mas não é fornecido nenhum *feedback* explícito sobre seus acertos ou erros conforme emite seus retornos. (RUSSEL; NORVIG, 2013)

Segundo Géron (2019), Na aprendizagem não supervisionado os dados de treinamento não contém legendas para classificação ou verificação de quão próximo se chegou. Alguns importantes algoritmos de aprendizagem não supervisionada são listados e divididos abaixo:

- Agrupamento
  - K-Means
  - DBSCAN

  - Análise Hierárquica de Clusteres (do inglês, *Hierarchical Cluster Analysis*, (HCA))
- Detecção de Anomalias e Novidades
  - One-class SVM
  - Árvore de Isolamento
- Visualização e Redução de Dimensões
  - Análise do Componente Principal (do inglês, *Principal Component Analysis*, (PCA))
  - Kernel PCA
  - Locally-Linear Embedding (LLE)
  - t-distributed Stochastic Neighbor Embedding (t-SNE)
- Regras de Associação
  - Apriori
  - Eclat

A tarefa mais comum para uso de aprendizagem não supervisionada é o agrupamento de certo conjunto de dados em grupos semelhantes com base em uma ou mais características, seja para classificação ou visualização destes dados.

3.1.3 Aprendizagem Supervisionada

Segundo Russel e Norvig (2013), na aprendizagem supervisionada, o agente observa alguns exemplos de pares de entrada e saída, e aprende uma função que faz o mapeamento desta entrada para a saída.

Também é dito por Géron (2019) que na aprendizagem supervisionada os dados esperados para previsão estão relacionados com os dados iniciais do treinamento, sendo muito utilizada para casos de classificação ou de previsão numérica de um valor, dadas certas características (do inglês, *features*) como é o caso deste trabalho. Alguns exemplos de algoritmos de Aprendizagem Supervisionada são:

- K vizinhos mais próximos (do inglês, *K-Nearest Neighbors*, KNN)
- Regressão Linear
- Regressão Logística
- Máquinas de Vetores de Suporte (do inglês, *Support Vector Machines*, SVMs)
- Árvores de Decisão e Floresta Aleatória
- Redes Neurais

Neste trabalho, temos um conjunto de entrada definido, buscamos mapear uma função que defina sua saída e temos como definir quão acertado foi esse mapeamento. Sendo assim, iremos buscar algoritmos na categoria de Aprendizagem Supervisionada. Dentro desta categoria, existem diversos métodos que podem ser utilizados para solucionar o problema alvo deste trabalho, mas foram escolhidos para implementação e análise os algoritmos de Regressão Linear e Redes Neurais.

O algoritmo de Regressão Linear foi escolhido por ser já muito utilizado nas resoluções destes problemas de previsão de demanda, gerando excelentes resultados em trabalhos já

existentes como evidenciado no de Carbonneau, Vahidov e Laframboise (2008).

Já as Redes Neurais, sua escolha está baseada no fato de que algoritmos dessa classe estão sendo os mais utilizados na literatura para problemas com foco na logística, principalmente envolvendo planejamento, como demostrado por Bousqaoui, Achchab e Tikito (2019). Além disso, as Redes Neurais são aproximadores universais, fazendo com que sejam interessantes para aplicações de problemas de previsão (SHAHRABI; MOUSAVI; HEYDAR, 2009). Seus detalhes serão abordados na Seção 3.3.

3.2 Regressão Linear

Regressão Linear é um dos algoritmos mais simples de aprendizagem supervisionada. De acordo com Albon (2018) embora muitas vezes ela nem seja considerada Aprendizagem de Máquina pela sua simplicidade, ela é um método comum e muito útil para realizar previsões quando a informação que se deseja prever é um valor quantitativo, que é o caso deste trabalho.

A Regressão Linear pode ser utilizada tanto com modelos univariados quanto modelos multivariados, que serão apresentados na Subseção 3.2.1 e Subseção 3.2.2.

3.2.1 Regressão Linear Univariada

Russel e Norvig (2013) definem Regressão Linear como uma função linear univariada com entrada x e saída y, tendo a forma $y = w_1x + w_0$ sendo w_0 e w_1 coeficientes reais de valores a serem aprendidos.

É utilizada a letra w porque imagina-se os coeficientes como pesos (em inglês, *weights*). O valor de y é alterado conforme a mudança do peso relativo de um termo para outro. Após definir w como o vetor $[w_0, w_1]$ é definida a fórmula:

$$h_w(x) = w_1x + w_0$$

O processo de se obter os coeficientes corretos é o que ocorre durante o processo de treino de um algoritmo de Regressão Linear sobre um certo conjunto de dados. Para isso, é necessário ter alguma métrica que defina o quão bom é ou não um coeficiente. Segundo Géron (2019), a função mais utilizada para este propósito é o Erro Quadrático Médio (do inglês, *Mean Squared Error*, MSE) sendo a escolhida para este trabalho.

3.2.2 Regressão Linear Multivariada

Tomando como base a Regressão Linear Univariada apresentada na Subseção 3.2.1, é possível estendê-la para que contemple mais de uma variável de entrada (RUSSEL; NORVIG, 2013). Isso é feito considerando-se que cada exemplo x_jé um vetor de n elementos, gerando a fórmula:

$$h_{sw}(x_j) = w_0 + w_1 x_{j,1} + ... + w_n x_{j,n}$$

Ou então:

$$x_j = \sum_i w_i x_{j,i}$$

Da mesma forma que na Regressão Linear Univariada, o processo de treinamento irá definir quais os coeficientes corretos para cada uma das posições do vetor de pesos. Porém, como lembrado por Russel e Norvig (2013), no modelo multivariado precisamos nos preocupar com a superadaptação.

Superadaptação ocorre quando uma dimensão que pode ser pouco útil na prática, seja considerada como tendo um por não estar no mesmo intervalo das outras dimensões. Para corrigir isso, é necessário regularizar e normalizar todos parâmetros para que esteja no mesmo intervalo e não sofram essas distorções.

3.3 Redes Neurais Artificiais

Russel e Norvig (2013) dizem que Redes Neurais Artificiais (RNAs), também chamadas de *Multilayer Perceptrons* (MLPs) são uma arquitetura de Aprendizagem de Máquina que baseia-se no funcionamento do cérebro, tendo neurônios conectados entre si, formando uma rede. Essas redes tem como seus predecessores os algoritmos lineares, como os já abordados na Seção 3.2.

De acordo com Goodfellow, Bengio e Courville (2016), essa ideia é inspirada no fato de que como o cérebro é um exemplo de comportamento inteligente, seria possível realizar uma "engenharia reversa" nele, buscando seus princípios mecânicos na tentativa de replicá-los, conseguindo assim ter comportamentos semelhantes à inteligência.

3.3.1 Estrutura das Redes Neurais Artificiais

Para Goodfellow, Bengio e Courville (2016), podemos nos referir às RNAs como operadores de uma sequência que contém vetores $x^{(t)}$ com o índice temporal t indo de 1 até τ. Em ambientes reais, essas redes operam com lotes de entradas com uma diferente sequencia de tamanho τ para cada um dos membros do lote.

Falando-se mais especificamente da sua estrutura em rede, RNAs são compostas por nós ou unidades (do inglês, *units*) que são conectados por ligações direcionais cada uma com um peso específico. (RUSSEL; NORVIG, 2013)

Para exemplificar, dizemos ter duas unidades i e j conectadas entre si que visam propagar a ativação a_i da unidade i para a j. Consideramos também que cada ligação em uma rede tem um peso numérico $w_{i,j}$ (interpreta-se: peso de i para j) que determina a força e o sinal dessa conexão.

Para definir a próxima propagação então, a unidade j primeiramente calcula uma soma ponderada de todas suas entradas:

$$in_j = \sum_{i=0}^{n} w_{i,j} a_i$$

Importante salientar que para computar isso, cada unidade possui uma entrada "fictícia"$a_0 = 1$ com seu peso $w_{0,j}$ associado.

Após calcular a soma ponderada de todas as suas entradas, é necessário aplicar-se a função de ativação, representada por g, sobre este valor:

$$a_j = g(in_j)$$

Ou então, juntando-se as duas em uma só expressão:

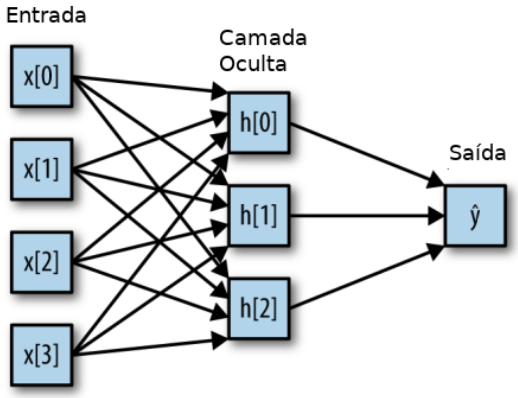
$$a_j = g\left(\sum_{i=0}^{n} w_{i,j} a_i\right)$$

Após termos compreendido a forma que estrutura-se uma unidade, devemos compreender as formas como podemos arranjar estas unidades dentro de uma rede. Uma das formas é a *Feed Forward Network*, rede que tem seu fluxo de informação apenas em uma direção. Uma rede assim não possui um estado interno adicional além dos próprios pesos.

Outra forma é através de Redes Recorrentes que veremos na Subseção 3.3.2, em que além dos pesos, temos o estado atual de cada célula se alterando com base nas informações que passaram anteriormente por aquela camada. No entanto, ambas as formas são semelhantes quanto à estrutura geral de unidades dispostas em forma de camadas, que ligam-se em sequência com a camada anterior e a próxima.

Müller e Guido (2016) explicam que após as entradas do vetor x chegarem na camada de entrada, as transformações são realizadas nas unidades da camada mais interna, chamadas de unidades ocultas (do inglês, *hidden units*). Em seguida, é calculado o valor final $\hat{y}$ resultado das transformações sobre aquele determinado conjunto x. Uma ilustração de um modelo assim é ilustrado na Figura 1.

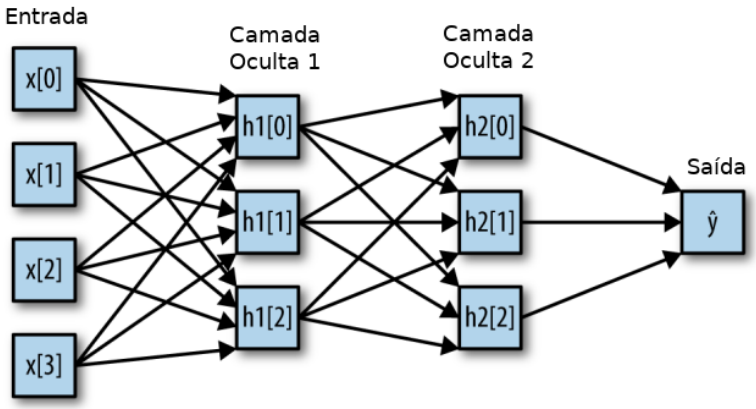
Figura 1 – Rede Neural Com Uma Camada Oculta



Fonte: Adaptado do autor Müller e Guido (2016)

As ANNs podem ter diversas camadas, o que faz aumentar o número de transformações aplicadas ao conjunto de entrada. Um modelo com duas camadas ocultas é ilustrado na Figura 2. Müller e Guido (2016) afirmam que quantidade de camadas e também de unidades por camada são parâmetros importantes a serem definidos e devem variar de acordo com a complexidade dos dados de entrada.

Figura 2 – Rede Neural Com Duas Camadas Ocultas



Fonte: Adaptado do autor Müller e Guido (2016)

Burkov (2019) diz que todo aprendizado de RNA precisa ter uma *loss function* (função de perda), um critério de otimização baseado nessa mesma *loss function* que geralmente é chamada de *cost function* (função de custo), uma *activation function* (função de ativação) responsável por realizar as operações dentro das unidades, e uma *optimization function* (função de otimização) que se apoia nos dados do treinamento buscando melhorar os pesos de cada conexão.

Existem diversos tipos de algoritmos e técnicas para cada um destes componentes da estrutura. Apenas para citar alguns dos algoritmos de otimização, temos:

- Stochastic Gradient Descent
- Adagrad
- Momentum
- RMSprop
- Adam

Veremos mais sobre como definir a quantidade de camadas, neurônios e algoritmos na Seção 3.4.

Utilizando essas funções, podemos construir o processo de treino de uma Rede Neural, sendo que primeiramente ela recebe o vetor de todas entradas. Em sequência, ela organizará todas essas entradas em diferentes lotes de igual tamanho. Ao final de todos os lotes, a RNA passou por todos dados do treino, caracterizando o fim de uma época (do inglês, *epoch*) para Burkov (2019).

Tipicamente são necessárias diversas épocas até que os valores dos pesos não mais se alterem de uma época para outra, demonstrando que a rede atingiu o melhor modelo de pesos possível para aquele conjunto com a sua estrutura.

A cada final de época, a rede atribuirá uma pontuação para estes pesos utilizando a *loss function*. Em sequência, rebalanceará os pesos de acordo com a função de otimização num processo chamado *back-propagation* em que, segundo Al-Masri (2019), através da função de otimização a Rede neural recalcula os pesos das suas unidades na direção contrária ao fluxo de informação normal da rede.

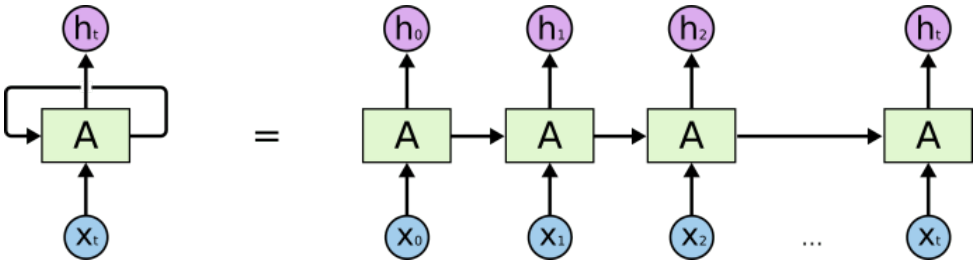
Na seção Subseção 3.3.2 veremos mais especificamente sobre as Redes Neurais Recorrentes, que se estruturam de forma diferente das *Feed Forward Networks*.

3.3.2 Estrutura das Redes Neurais Recorrentes

Para Goodfellow, Bengio e Courville (2016), Redes Neurais Recorrentes (RNR) são uma família dentro das RNAs muito utilizadas para processamento de dados sequenciais. Ou seja, são utilizadas para o processamento de dados que tenham uma relação de sequência entre si. Elas estendem as RNAs para incluir ciclos dentro da rede, criando uma representação da influência de valores já passados por determinado momento na rede nos valores que lá estarão no momento.

Estruturalmente, essas redes podem ser formadas como "cópias" de uma RNA, em que cada cópia passa uma mensagem de sua transformação a sua sucessora. Essa recorrência é chamada de *loop* e é feito realizando um *unflod* (do inglês, *desenrolar*) nas unidades da camada recorrente, como ilustrado na Figura 3, sendo A uma camada recorrente de uma rede, que recebe a entrada x_t e emite o valor de saída h_t (DATA SCIENCE ACADEMY, 2019).

Figura 3 – *Unfold* de uma Rede Neural Recorrente

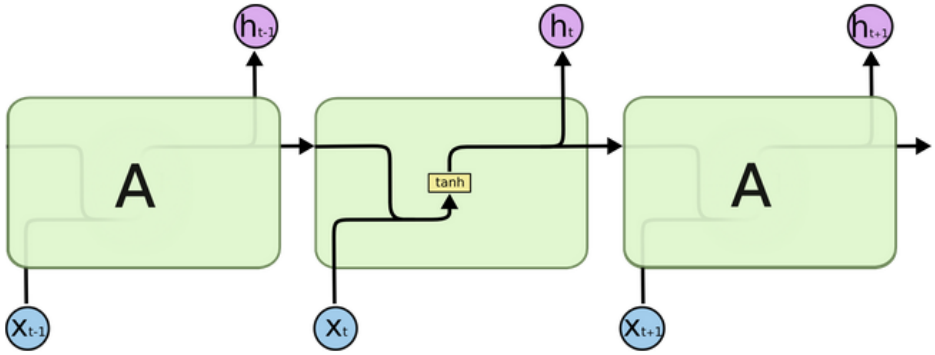


Fonte: DATA SCIENCE ACADEMY (2019)

Nesta rede, suas saídas são alimentadas de volta às suas entradas, fazendo com que os níveis de ativação da rede formem um estado interno dinâmico e adaptável a mudanças.

Para Phi (2018), por causa dessa funcionalidade recorrente, as RNR também possuem uma certa memória de curto prazo (do inglês, *short-term memory*), já que suas funções atuais pode ser alteradas de acordo com os valores anteriores que estão interferindo recursivamente nas novas entradas. Isso as torna inclusive, ainda mais semelhantes ao cérebro humano.

Figura 4 – Por Dentro de uma Unidade de RNR



Fonte: Olah (2015)

Na Figura 4 observamos como é estruturada uma unidade dentro de uma camada de rede neural recorrente. Como explicado por Phi (2018), uma combinação da entrada atual x_t combinado com a saída anterior h_{t-1} passam por uma ativação $\tanh$ que faz os valores ficarem entre -1 e 1 gerando a saída da célula, h_t.

No entanto, existe um problema com o uso de Redes Neurais Recorrentes já demonstrado por vários autores, como por exemplo Bengio, Simard e Frasconi (1994), que demonstram a dificuldade das RNR lidarem com dependências de longo prazo devido a perda dessa informação durante o rebalanceamento dos pesos das unidades.

Phi (2018) Explica que isso ocorre durante a *back-propagation*, onde os pesos são recalculados do final da rede em direção ao começo com base no gradiente obtido através da função de otimização.

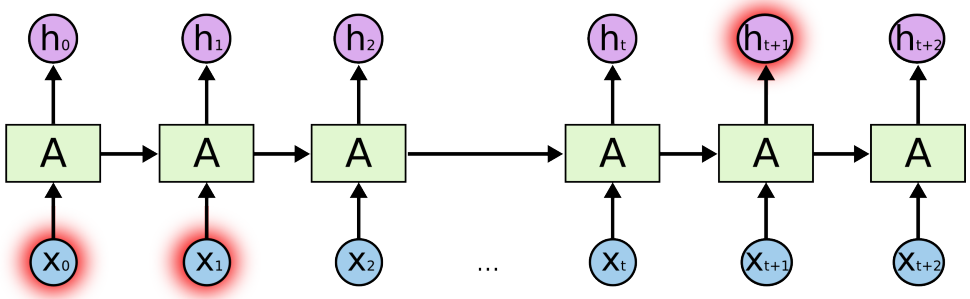
O problema é que cada célula na camada calcula o seu gradiente tomando como base o gradiente já calculado na camada anterior. Então, caso os ajustes na camada anterior foram pequenos, os ajustes nessa camada serão ainda menores.

Isso faz com que o gradiente decresça exponencialmente, desaparecendo enquanto se propaga na rede. Por causa disso, as primeiras camadas falham em ter algum aprendizado. Esse é o chamado problema do Gradiente Decrescente (do inglês, *Vanishing Gradient*).

Sendo assim, podemos ter situações como ilustrado na Figura 5, em que é representado um *unfold* de uma unidade em que um valor que já passou por essa camada em estados anteriores, representado por h_{t+1}, teria de ter um peso alto e ser combinado com a entrada x_0 e x_1 para prever corretamente essa entrada.

Para criarmos um modelo que previsse corretamente a situação ilustrada na Figura 5, teríamos de ter uma rede que atribuísse um peso alto às unidades iniciais e finais e não tanto para as medianas. Porém, devido ao problema do gradiente decrescente, as unidades iniciais só terão pequenas alterações a cada otimização, não tendo a relevância necessária.

Figura 5 – O problema do gradiente decrescente das RNRs



Fonte: Olah (2015)

Esse problema é resolvido com a arquitetura de *Long Short-Term Memory*, que será abordado na Subseção 3.3.3.

3.3.3 Estrutura da *Long Short-Term Memory* (LSTM)

A arquitetura de Memória de Longo Prazo (do inglês, *Long Short-Term memory*, LSTM) foi introduzida no cenário de IA por Hochreiter e Schmidhuber (1997) e é a arquitetura de RNN mais efetiva para ser usada na prática segundo Burkov (2019), já que ela resolve o problema do gradiente decrescente visto na Subseção 3.3.2.

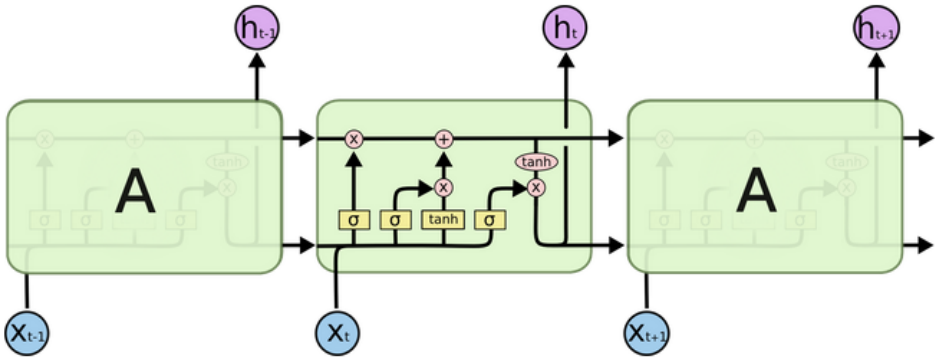
Além deste fato, de acordo com TENSORFLOW (2020), essa rede pode ser muito bem utilizada para processar uma série de dados temporais grande pois mantém um estado interno que "resume" a informação vista até o momento. Isso faz com que ela tenha um maior sucesso na previsão de dados em série, sendo o motivo principal pelo qual foi a escolhida para este trabalho.

O ponto fundamental em uma LSTM são os *gates* (portões). Burkov (2019) explica que LSTMs utilizam esses portões para que possam manipular as informações armazenadas nas células através de funções. Assim, a LSTM pode "decidir" em algum momento guardar, ler, ou deletar uma informação. Com isso, ela consegue gerar um modelo que utiliza de forma recorrente apenas as informações que de fato são úteis ao modelo.

Segundo DATA SCIENCE ACADEMY (2019), existem três portões para realizar a manipulação na memória nessa arquitetura: O *Forget Gate* (portão de esquecimento), o *Input Gate* (portão de entrada) e o *Output Gate* (portão de saída).

Esses portões garantem a chave para o sucesso da arquitetura LSTM, pois conseguem gerar um estado da célula único, representado por c na linha superior do diagrama da Figura 6. Olah (2015) exemplifica que pode-se pensar nessa linha como uma linha de produção em que será "fabricado"a saída de dentro dessa célula, que pode ter seu fluxo alterado ou não dependendo dos resultados dos portões internos.

Figura 6 – Por Dentro de uma Unidade de LSTM

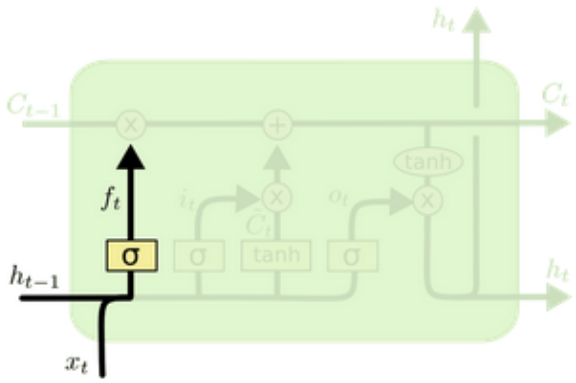


Fonte: Olah (2015)

Tecnicamente, esses portões são redes internas à uma camada de *LSTM* ativados por uma função sigmóide, que transforma valores recebidos em 0 e 1 que simbolizarão o quanto de informação será passada, sendo 0 nada e 1 a informação completa sem alteração.

Olah (2015) ilustra a ação dos portões em uma unidade LSTM em quatro passos distintos. Em um primeiro passo, é necessário decidir qual informação será descartada do estado da célula. Essa decisão é feita com uma camada de ativação sigmóide denominada *forget gate* que recebe h_{t-1} e x_t e emite um valor entre 0 e 1 para cada entrada no estádo de célula c_{t-1}.

Figura 7 – Por dentro de uma unidade LSTM: *Forget Gate*



Fonte: Olah (2015)

No *Forget Gate*, as informações que não são mais úteis em uma célula são removidas. As entradas x_t e $h_{(t-1)}$ são alimentadas a esse *gate* e multiplicadas por matrizes de peso, e em seguida tem a adição do viés (peso) interno. Isso pode ser descrito pela fórmula

$$f_t = \sigma\left(W_f . [h_{t-1}, x_t] + b_f\right)$$

Este resultado é passado para uma função de ativação que fornece uma saída binária, que caso seja 0, a informação não é armazenada. Caso a saída seja 1, a informação é armazenada para uso futuro, como ilustrado na Figura 7.

O próximo passo é decidir qual informação será armazenada no estado da célula, e podemos quebrar ele em duas etapas. Primeiramente, o *input gate* decide qual o peso i_t dos valores que serão atualizados utilizando uma função sigmóide com a expressão

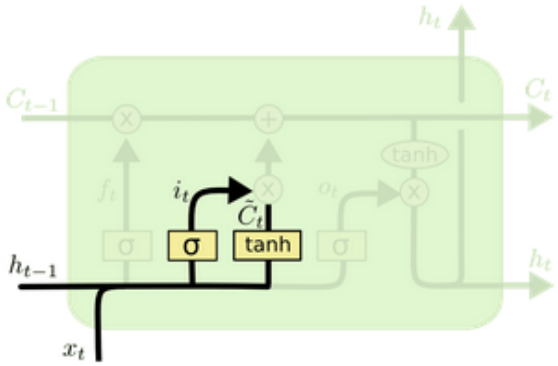
$$i_t = \sigma\left(W_i . [h_{t-1}, x_t] + b_i\right)$$

Em seguida, uma camada de ativação $\tanh$ cria um vetor dos novos valores candidatos $\tilde{C}_t$ combinando x_t e $h_{(t-1)}$

$$\tilde{C}_t = \tanh\left(W_c . [h_{t-1}, x_t] + b_c\right)$$

Os resultados dessas duas ativações serão combinados criando uma atualização ao estado, ilustrado na Figura 8.

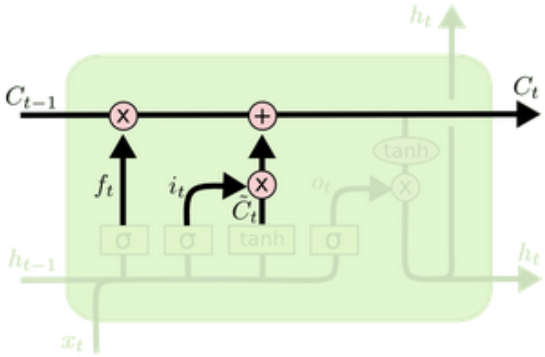
Figura 8 – Por dentro de uma unidade LSTM: *Input Gate*



Fonte: Olah (2015)

Em sequência, o estado de célula antigo C_{t-1} será atualizado recebendo seu novo valor C_t, ilustrado na Figura 9, através da junção dos resultados dos passos anteriores, em que f_t é multiplicado por i_t e $\tilde{C}_t$.

Figura 9 – Por dentro de uma unidade LSTM: alteração do estado da célula



Fonte: Olah (2015)

Ou seja, através da expressão

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

estamos escalando os valores candidatos por quanto foi decidido atualizar cada um dos valores do estado.

Finalmente, será decidido o que irá ser gerado na saída h_t dessa unidade baseado-se no estado final da célula, porém com mais algumas operações. É realizada uma ativação sigmóide

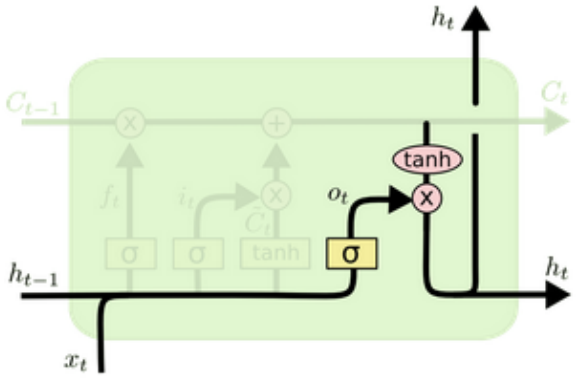
$$o_t = \sigma\left(W_o.\left[h_{t-1}, x_t\right] + b_o\right)$$

e então o novo estado da célula passa por uma última ativação $\tanh$ colocando os valores entre -1 e 1 e multiplicando esse estado pela saída da ativação sigmóide para apenas transmitir as partes necessárias

$$h_t = o_t * \tanh(C_t)$$

A saída h_t é então enviada para a saída da camada e como entrada para a célula adjacente, servindo como valor anterior h_{t-1} na sua própria ativação, como ilustrado na Figura 10.

Figura 10 – Por dentro de uma unidade LSTM: *Output Gate*



Fonte: Olah (2015)

3.4 Hiperparâmetros

Existem diversos parâmetros envolvidos na construção de uma rede neural que podem alterar o seu processo de aprendizagem e portanto, sua performance. Nesse Capítulo, iremos abordá-los assim como o que foi utilizado para decidirmos o que seria configurado em nossa própria rede para prevenir *underfitting* e *overfitting*.

Underftiting ocorre quando um modelo não tem boas avaliações métricas ao prever um conjunto de dados. Podem existir inúmeras razões para isso, mas as mais comuns são que o modelo é muito simples para os dados propostos, ou que as entradas informadas não são informativas o suficiente (BURKOV, 2019).

Overfitting também chamado de Alta Variância (*high variance* ocorre quando a rede prevê muito bem o conjunto de treino, porém ao ser utilizado em um conjunto novo, como por exemplo o de validação, ele prevê muito pior, o que significa que ela se tornou muito boa em prever apenas o conjunto de treino, e não conseguiu criar um modelo abstrato o suficiente para ser usado em novos valores.

Burkov (2019) ainda diz que da mesma forma que com *underfitting* podem ter inúmeras possibilidades do porque disso acontecer, mas geralmente ou o modelo é muito complexo

para um conjunto de dados relativamente simples, ou existem muitas entradas diferentes para poucos registros de treino.

Algumas destas soluções podem ser: utilizar um modelo mais simples com menos camadas e neurônios, reduzir a quantidade de variáveis de entrada, adicionar mais registros ao conjunto de treino, ou então regularizar o modelo.

3.4.1 Número de Camadas

Géron (2019) diz que para muitos problemas, pode-se obter resultados já muito significativos apenas com uma camada na rede neural, tendo já sido demonstrado que MLPs com apenas uma camada entregaram resultados satisfatórios em funções complexas, desde que tivesse o número suficiente de neurônios.

No entanto, ele continua dizendo que para objetivos complexos como categorização de imagens, por exemplo, não se deve ignorar a vantagem descoberta com Redes Neurais Profundas (do inglês, *Deep Neural Networks*) em que é possível treinar diferentes partes da rede para diferentes atividades envolvidas no processo de reconhecimento de imagem e depois, essas partes podem ser reutilizadas para outras formas de modelos e redes.

Moolayil (2019) concorda com isso, ao dizer que na maioria dos casos, adicionar diversas camadas irá apenas aumentar o custo computacional significamente enquanto que apenas marginalmente a performance.

3.4.2 Número de Neurônios

Para Moolayil (2019), uma boa regra geral para a escolha do número de neurônios da primeira camada, é multiplicar o número de dimensões do vetor de entrada x e pegar o valor mais próximo a este resultado que esteja na potência de 2.

Por exemplo, digamos que se tenha um vetor de entrada com 10 dimensões. Utilizando essa regra, escolheríamos o valor na escala de potências de dois mais próximas a 20, optando então por 16 neurônios.

Já Géron (2019), diz que essa prática de diminuir o número de neurônios por camada foi demonstrado em não trazer, benefícios mostrando ser mais dependente da forma do conjunto de dados. Ele diz o melhor é ir aumentando gradualmente o número de neurônios e que isso ainda é uma "arte"mais do que ciência. Porém, continua dizendo que uma abordagem mais simples seja colocar mais neurônios do que imagina necessário e utilizar de técnicas como uma camada de *dropout* ou *early stopping* para prevenir *overfitting*.

3.4.3 Tamanho do Lote de Treinamento

Moolayil (2019) diz que utilizar um tamanho de lote (do inglês, *batch size*) de 32 ou 64 vai na maioria dos casos entregar uma curva de aprendizagem crescente e satisfatória, no

entanto Géron (2019) afirma que raramente se terá benefícios utilizando mais do que 32, sendo vantajoso utilizar um lote menor para ter o aprendizado mais rapidamente.

3.4.4 Atribuição dos Pesos Iniciais dos Neurônios

É necessário algum algoritmo para inicializar os pesos das unidades em cada camada de uma rede neural para que estes depois sejam alterados pela *back-propagation* a cada época. Moolayil (2019) considera muito importante uma boa inicialização para a performance de uma rede e velocidade de treinamento.

Para Géron (2019), as mais comuns são a inicialização de Glorot, também chamada de inicialização de Xavier apresentada por Glorot e Bengio (2010), que pode ser expressa pela fórmula

$$fan_{avg} = \frac{fan_{in} + fan_{out}}{2}$$

Mas Géron (2019) também diz que existem outras ativações que se inspiram na estratégia utilizada por elas, mas que são otimizadas para outras funções de ativação. Enquanto que inicialização de Glorot deve ser utilizada em conjunto com funções de ativação como a Tangente Hiperbólica (do inglês, *Hyperbolic Tangent Function*, TanH), a Logística e a Softmax, a inicialização de *He* deve ser utilizada para ReLU e suas variantes e a inicialização de LeCun deve ser utilizada para a função de ativação SELU.

3.4.5 Funções de Ativação

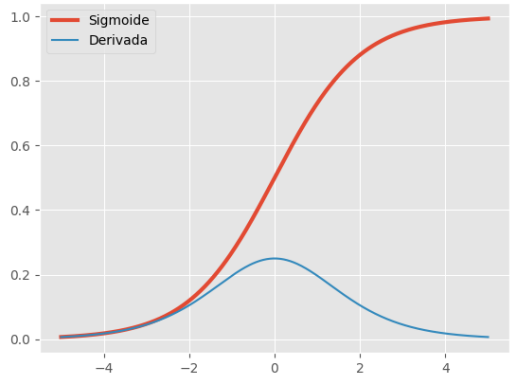
De acordo com Burkov (2019), no passado era muito utilizada a função de ativação sigmóide, mas hoje em dia as funções de ativação mais utilizadas são a TanH, que é é similar a função logística mas no intervalo entre -1 a 1. E a Unidade Linear Retificada (do inglês, *Rectified Linear Unit*, ReLU) que iguala a 0 quando tem uma entrada negativa e a própria entrada em outro caso.

Facure (2017) diz que no início era comum utilizar-se a ativação sigmóide para replicar o comportamento do cérebro biológico, porém conforme outras funções foram apresentadas e demonstrando maiores performances, a sigmóide passou a ser bem menos utilizada. Esta função assim como sua derivada são descritas por

$$\sigma(x) = \frac{1}{1 + e^x}$$

$$\sigma'(x) = \sigma(x).(1 - \sigma(x))$$

Figura 11 – Ativação Sigmóide



Fonte: Facure (2017)

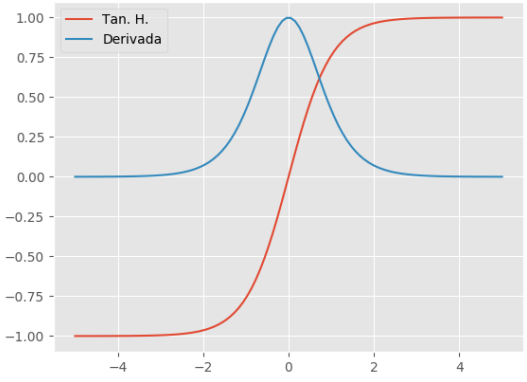
Quanto à função TanH, podemos expressar ela e sua derivada por

$$\tanh(x) = 2\sigma(2x) - 1$$

$$\tanh'(x) = 1 - \tanh^2(x)$$

Tanto esta função quanto a Sigmóide possuem um característico formato de "S" como podemos notar na Figura 11 e na Figura 12.

Figura 12 – Ativação TanH



Fonte: Facure (2017)

Na grande maioria das vezes a função ReLU é favorecida no lugar da TanH. Na verdade, para Moolayil (2019) embora se tenha uma grande opção de funções a serem utilizadas além destas, na grande maioria dos casos a *ReLU* funciona perfeitamente bem e nos casos em

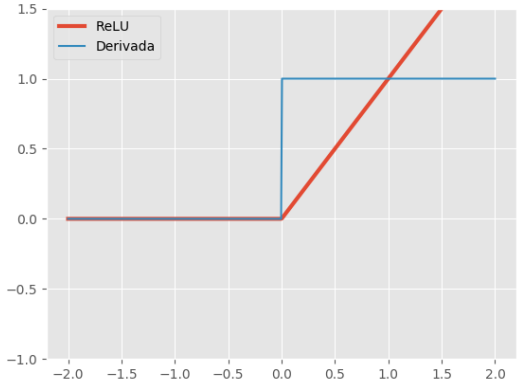
que ela possa não entregar a performance desejada, pode-se alterar para sua variação PReLU (do inglês, *Parametric Leaky Rectified Linear Unit*) ou a *leaky* ReLU.

Para Facure (2017) a ativação ReLU é muito mais eficiente do que as funções Sigmoidais e é uma das descobertas que contribuiu de forma significativa para a recente popularidade de aprendizagem profunda (em inglês, *Deep Learning*). Podemos descrever a função ReLU e sua derivada respectivamente por

$$ReLU(x) = \max\{0,x\}$$

$$ReLU'(x) = \begin{cases} 1 \text{ se } x \geqslant 0 \\ 0 \text{ se } x < 0 \end{cases}$$

Figura 13 – Ativação ReLU



Fonte: Facure (2017)

Moolayil (2019) também explica que a *leaky* ReLU pode ser melhores pois apesar da função ReLU ser muito eficiente, ela sofre algumas vezes com o problema de "morte de neurônios" em que em alguns momentos do treino, alguns neurônios nessa ativação podem passar a registrar apenas 0 como saída, especialmente quando se usa um alto coeficiente de aprendizagem.

Podemos expressar a função *leaky* ReLU e sua derivada respectivamente pelas expressões

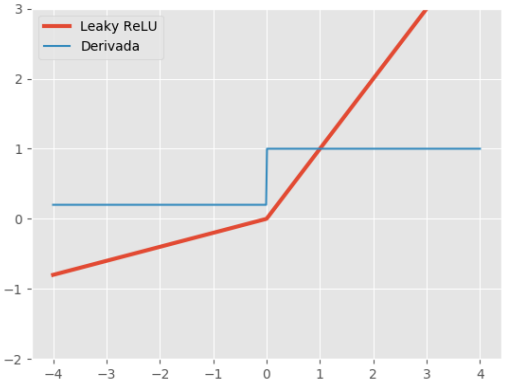
$$leakyReLU(x,\alpha) = \max\{\alpha x,x\}$$

e

$$leakyReLU'(x,\alpha) = \begin{cases} 1 \text{ se } x \geqslant 0 \\ \alpha \text{ se } x < 0 \end{cases}$$

e podemos comparar visualmente as diferenças entre essas duas ativações na Figura 13 e na Figura 14.

Figura 14 – Ativação *leaky* ReLU



Fonte: Facure (2017)

Além destas, como mais nova temos a ativação de Unidade Linear Exponencial (do inglês, *Exponential Linear Units*, ELU) introduzida por Clevert, Unterthiner e Hochreiter (2016) e pode ser visualizada na Figura 15 e ela e sua derivada podem ser representadas pelas expressões

$$ELU(x,\alpha) = \begin{cases} x \text{ se } x \geqslant 0 \\ \alpha(e^x - 1) \text{ se } x < 0 \end{cases}$$

e

$$ELU'(x,\alpha) = \begin{cases} 1, \text{ se } x \geqslant 0 \\ ELU(x,\alpha) + \alpha \text{ se } x < 0 \end{cases}$$

Para Facure (2017) assim como a *leaky* ReLU, a ELU resolve o problema das unidades mortas apresentado pelas ReLUs e Géron (2019) diz que esta consegue ser mais eficaz que aquela, chegando a melhores modelos mais rapidamente durante treino.

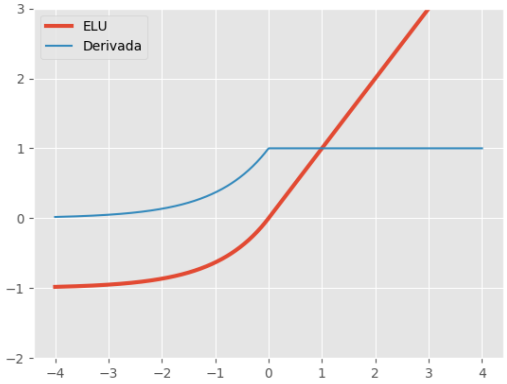
Ainda segundo Géron (2019) o único problema com a ativação ELU é que ela exige maior poder computacional, demorando mais para fazer previsões. Durante o treino isso não é tão perceptível pois ela também chega no modelo ideal mais rápido, compensando essa falha. Mas durante validação pode-se perceber essa demora.

Por fim, expandido encima da ELU, temos a SELU que é apenas a versão escalada da ELU mas que foi demonstrada por Klambauer et al. (2017) que uma rede que utiliza essa função e apresenta mais alguns requisitos pode se auto-normalizar.

Hansen (2019) diz que para ocorrer essa auto-normalização, é preciso ser utilizada a inicialização normal de LeCun e caso seja utilizada uma camada de Dropout deve ser utilizado

o Alpha Dropout, que é um Dropout que mantém a média e a variância das entradas, garantido uma auto-normalização.

Figura 15 – Ativação ELU



Fonte: Facure (2017)

3.4.6 Funções de Otimização e Coeficiente de Aprendizagem

Assim como com as funções de ativação, também temos um número grande de escolhas referente às funções de otimização. Para Moolayil (2019) a mais recomendada na grande maioria dos cenários é a ativação Adam (do inglês, Adaptive Moment Estimation) e em cenários em que essa função possa não estar entregando os melhores resultados, é recomendado explorar os otimizadores Adamax ou Nadam que são versões trabalhadas a partir do Adam.

Géron (2019) diz que o otimizador Adam se baseia em dois otimizadores anteriores, o Momentum e o RMSProp tendo como diferença a troca da soma exponencial decrescente pela média exponencial decrescente. Podemos definir a fórmula do otimizador Adam na seguinte sequência de fórmulas:

$$m \leftarrow \beta_1 m - (1 - \beta_1) \bigtriangledown_\theta J(\theta)$$

$$s \leftarrow \beta_2 s + (1 - \beta_2) \bigtriangledown_\theta J(\theta) \otimes \bigtriangledown_\theta J(\theta)$$

$$\hat{m} \leftarrow \frac{m}{1 - \beta_1^t}$$

$$\hat{s} \leftarrow \frac{s}{1 - \beta_2^t}$$

$$\theta \leftarrow \theta + \eta \hat{m} \oslash \sqrt{\hat{s} + \epsilon}$$

Sendo que o hiperparâmetro B_1 é tipicamente inicializado com 0.9, o B_2 com 0.999, o η com 0.001 e ϵ com um pequeno valor como 10^{-7} e t o valor de iteração atual, que inicia em 1. Estes valores podem ser deixados nestes padrões pois o otimizador Adam é um algoritmo de aprendizagem adaptativa.

Quanto ao coeficiente de aprendizagem, segundo Moolayil (2019) ele define quão frequentes serão as atualizações aos pesos da rede. Para o caso do Adam, seu padrão é 0.001 e essa é considerado um bom padrão que raramente precisa ser alterado.

Existe ainda uma variação preferível ao Adam que é o Nadam, que possui a adição do NAG (do inglês, *Nesterov momentum*, NAG) ao Adam.

Segundo Schmidt (2018), o NAG é uma mudança no otimizador Adam em que ao invés do gradiente ser computado da posição de θ_t ele é computado de $\theta_{intermedirio} = \theta_t + \mu\upsilon_t$ Isso faz com que o gradiente sempre aponte na direção correta, ganhando tempo de treino.

3.4.7 Número de Épocas

Quanto ao número de épocas que se deve treinar um modelo, não existem respostas definitivas. Moolayil (2019) diz que depende do modelo, e que geralmente quanto mais épocas melhor irá ser o resultado final, porém também muito mais custo computacional, as vezes por um ganho marginal de performance.

3.4.8 *Dropout*

Burkov (2019) explica o conceito de *dropout* como a exclusão de alguns neurônios aleatoriamente e temporariamente da rede. Quanto maior a porcentagem de unidades excluídas, maior é o efeito de regularização. Isso pode ser feito na prática como um parâmetro passado a criação de uma rede ou ainda como uma camada específica entre duas outras. Em ambas formas, o parâmetro passado fica entre 0 e 1.

Moolayil (2019) diz que uma camada *dropout* ajuda a reduzir *overfitting* introduzindo capacidades de regularização e generalização a própria rede. Isso funciona pois com certos neurônios faltando, a rede se força a encontrar novas padrões com os neurônios restantes, tornando mais propensa a abstrair o modelo estatístico em sí, e não apenas aos dados do treinamento.

Géron (2019) complementa essa explicação dizendo que utilizando *dropout* cada neurônio em uma rede neural é forçado a atingir seu nível máximo individual de acuracidade, pois não pode depender dos outros da rede.

Além disso, Géron (2019) também afirma que o parâmetro indicado geralmente é de 0.5 ou seja, a cada época metade aleatória dos neurônios estarão desligados e que o modelo já treinado usado em produção não possuirá nada de *dropout*, sendo usado apenas em treinamento.

3.4.9 *Early Stopping*

Early Stopping (tradução livre, parar cedo) é uma forma de regularização que pode ser usada para parar o treinamento assim que a métrica de validação chegar em um mínimo aceitável, fazendo com que sejam ganho tempo de treinamento de acordo com Géron (2019).

3.4.10 Regularizações L1 e L2

Outra forma de reduzir *overfitting* é incluir processos de regularização L1 ou L2 na rede neural. Moolayil (2019) Explica que os pesos dos neurônios são atualizados a cada iteração. Quando o modelo encontra uma amostra "barulhenta" o modelo acaba tentando acomodar todos os pesos para essa amostra irregular.

O processo de regularização adiciona os pesos das extremidades da rede para a função de perda, representando uma maior perda caso isso aconteça. Isso que faz com que a rede neural faça a próxima iteração na direção correta.

Na regularização L1, os pesos absolutos são adicionados a função de perda, reduzindo os pesos a 0 e tornando a computação mais rápida. na regularização L2 o quadrado dos pesos são adicionados a função de perda, fazendo os pesos serem reduzidos a 0, mas nunca totalmente a 0. Na maioria dos casos, a regularização L2 é mais recomendada, e recebendo o parâmetro padrão 0.01.

4 Engenharia de Dados

Burkov (2019) diz que Engenharia de Dados é o processo necessário de transformar dados crus em um conjunto de dados (do inglês, *dataset*) propriamente dito, com um vetor de entrada relacionado com suas saídas. Isso geralmente é um processo trabalhoso que requer conhecimento da área e criatividade, já que os dados crus podem estar nos mais variados formatos e geralmente precisam ser tratados, padronizados e normalizados. Existem diversas técnicas dentro de Engenharia de Dados. Iremos falar nesse Capítulo apenas daqueles que foram relevantes no desenvolvimento deste trabalho.

4.1 Normalização

Para Burkov (2019), normalização é o processo de converter um conjunto de dados de uma determinada série, para um outro conjunto padronizado entre todas característica, geralmente os intervalos $[-1, 1]$ e $[0, 1]$. De forma geral, a fórmula de normalização é a seguinte:

$$\overline{x}^{(j)} = \frac{x^{(j)} - min^{(j)}}{max^{(j)} - min^{(j)}}$$

Normalização não é necessariamente um requerimento, porém pode aumentar consideravelmente a velocidade do treinamento de acordo com Moolayil (2019). Isso porque a normalização irá reduzir a computação e melhorar o aprendizado ao garantir que todas as entradas estão no mesmo intervalo, evitando que o processador tenha de trabalhar com valores muito pequenos ou muito grandes e acabe tendo erros de arredondamento.

4.2 Divisão do Conjunto de Dados

Para Burkov (2019), durante todo o processo de criação de uma rede divide-se o conjunto de dados em três partes: treinamento, validação e testes. A porcentagem de tamanho dos dados varia, geralmente sendo maior do que 70% para os dados de treino e dividida igualmente entre os dois conjuntos restantes.

Os conjuntos de validação e testes são geralmente do mesmo tamanho, muito menores do que o de treino e podem até ser o mesmo, não necessariamente precisando haver esta divisão. De qualquer foma, ele também lembra que as redes nunca devem ter acesso durante o treinamento à esses dados para não interferir na análise de sua real performance, já que é possível que uma rede fique *Underfitted* ou *Overfitted*.

4.3 Avaliação da Performance das Previsões

Para verificar se nosso modelo generalizou bem os dados de treino, é preciso definir métodos para testar sua performance e acurácia. Em um problema de regressão numérica, como é o caso deste trabalho, quanto mais próximas forem as previsões comparadas ao teste, melhor ele performou. Para verificar isso, geralmente é utilizado a métrica do Erro Quadrático Médio (EQM) segundo Burkov (2019). Outra métrica também utilizada é o Erro Médio Absoluto (EMA).

A vantagem do EQM, segundo Müller e Guido (2016), é a de que ela adiciona mais peso à diferença dos erros, pois eles são elevados ao quadrado antes da média ser calculada. Ou seja, em um cenário em que a maioria das previsões foi correta mas uma delas existiu um erro muito grande, EMA poderia ser pouco afetada enquanto que a EQM mostraria esse erro aumentado seu valor consideravelmente. Como a EQM é elevada ao quadrado, geralmente os resultados são demonstrados após se fazer a raiz quadrada dela.

5 Bibliotecas e Ferramentas Utilizadas

Para entendermos o ecossistema de aprendizagem de máquina em Python, é interessante olharmos pela ótica da analogia que Moolayil (2019) faz com *frameworks* de softwares. Ele diz que se analisarmos a história de desenvolvimento de *Softwares*, fica evidente que quanto mais o tempo passa, mais fácil fica construir sistemas de alto nível e complexidade. Isso se deve às ferramentas que automatizaram ou abstraíram complexos problemas em formas fáceis de serem utilizadas.

Essa evolução também ocorreu em Aprendizagem de Máquina. Conforme a comunidade percebeu que certas funções eram repetidas em diversos projetos e poderiam ser transformadas em APIs. Isso fez com que hoje existam vários blocos de construção prontos que podem ser integrados e utilizados nessa área, e alguns deles serão apresentados neste capítulo.

5.1 Python

De acordo com Müller e Guido (2016), Python acabou se tornando a "língua franca" para muitos estudantes de ciência de dados. Para eles, isso se dá pois essa linguagem combina o poder de uma linguagem de propósito geral com uma facilidade de uso comparável com linguagens de domínio específico como *MATLAB* ou *R*. Além de que, por ser uma linguagem interpretada, pode-se interagir com ela em tempo real em terminal ou até mesmo algo que se assemelha a um caderno de anotações como com o Jupyter Notebook.

Além desses motivos (ou talvez por causa deles), foram criadas e desenvolvidas as mais diversas bibliotecas que auxiliam operações necessários nesse campo, como carregamento de dados, visualização, estatísticas, processamento de imagens e mais, provendo aos seus usuários um vasto arsenal de funcionalidades, tanto específicas como gerais. Ou seja, nesse ecossistema é possível realizar a visualização e preparação dos dados, criação de redes neurais, teste e análise e ainda disponibilizar o resultado dem forma de API web ou interface gráfica, tudo com a mesma linguagem. Por todos estes motivos, essa foi a linguagem escolhida neste trabalho e as bibliotecas utilizadas serão apresentadas nessa seção.

5.1.1 NumPy

Albon (2018) Considera NumPy a fundação do ambiente de Aprendizagem de Máquina em Python, permitindo operações eficientes em estruturas de dados muito utilizadas nesse segmento, como vetores, matrizes e tensores. Além disso, NumPy contém as mais diversas funcionalidades para vetores unidimensionais e multidimensionais para execução funções matemáticas de alto nível como álgebra linear, transformação de Fourier e geradores de números pseudo-aleatórios.

NumPy também é de código aberto e um bloco fundamental de outras bibliotecas desse ecossistema, sendo por exemplo, o objeto de entrada da maioria das funções da biblioteca Scikit-Learn segundo Müller e Guido (2016). Para Harris et al. (2020) a funcionalidade principal desta biblioteca é a classe *ndarray* que nada mais é do que um vetor de N dimensões.

Figura 16 – Exemplo de uso do NumPy para criar vetor 2D

```
import numpy as np
np_vetor_2D = np.array([[11, 12, 13, 14], [21, 22, 23, 24]])
print(np_vetor_2D)
np.shape(np_vetor_2D)
```

```
[[11 12 13 14]
 [21 22 23 24]]
(2, 4)
```

Fonte: O autor.

Para exemplificar sua facilidade de uso, na Figura 16 temos um trecho de código que demonstra a importação desta biblioteca sendo apelidada de *np* e a criação de um vetor de duas dimensões a partir da chamada do método *array* recebendo como parâmetro uma lista da linguagem Python que contém mais outras duas listas de inteiros. A partir disso, é retornado à variável *np_vetor_2D* um objeto *ndarray* de duas dimensões. Nesse mesmo trecho também foi chamado o método *shape* desta biblioteca sobre este vetor, que mostra o formato do objeto. Neste caso, o resultado (2, 4) demonstra um vetor de 2 dimensões, cada uma com 4 elementos.

Figura 17 – Exemplo de uso do NumPy para transformar vetor 2D em 1D

```
np_vetor_1D = np_vetor_2D.ravel()
print(np_vetor_1D)
np.shape(np_vetor_1D)
```

```
[11 12 13 14 21 22 23 24]
(8,)
```

Fonte: O autor.

Ainda trabalhando com esse vetor 2D, podemos exemplificar na Figura 17 como também é possível transformar um vetor de várias dimensões em só uma com o método *ravel*. Após transformar o vetor criado na Figura 16, imprimimos no console o novo vetor e executamos novamente o método *shape* mostrando que agora resulta em (8,) significando um vetor de uma dimensão e 8 elementos.

Figura 18 – Exemplo de uso do NumPy para criar 2 vetores aleatórios e somá-los

```
np_vetor_aleatorio = np.random.random(4)
print(np_vetor_aleatorio)
np_vetor_aleatorio2 = np.random.random(4)
print(np_vetor_aleatorio2)
novo_vetor = np_vetor_aleatorio2 + np_vetor_aleatorio
print(novo_vetor)
```

```
[0.87576662 0.41910988 0.95039651 0.17727285]
[0.59326713 0.86522517 0.99247722 0.47187114]
[1.46903374 1.28433505 1.94287373 0.64914399]
```

Fonte: O autor.

Outra funcionalidade interessante da biblioteca NumPy é a facilidade de trabalho com números aleatórios e operações encima de vetores. Na Figura 18 temos um trecho de código que demonstra a criação de dois vetores aleatórios seguidos de sua soma. Podemos perceber ao analisar os resultados, que como os dois vetores apresentam o mesmo número de elementos, a operação normal de soma entre eles automaticamente executa uma soma entre cada um dos elementos.

Figura 19 – Exemplo de uso do NumPy para obter informações sobre vetores

```
ultimos2 = novo_vetor[-2:]
media = np.average(novo_vetor)
soma = np.sum(novo_vetor)
maior = np.max(novo_vetor)
menor = np.min(novo_vetor)

ultimos2, media, soma, maior, menor
```

```
(array([1.94287373, 0.64914399]),
 1.3363466282731629,
 5.3453865130926514,
 1.9428737288675175,
 0.6491439920844602)
```

Fonte: O autor.

Com o vetor obtido na Figura 18, podemos demonstrar várias operações que podem ser realizadas. Na Figura 19 é demonstrado como é possível obter vetores menores a partir de um mesmo vetor com o uso de índices. O índice [-2:] passado ao vetor significa duas posições antes do final até o final. Além disso, nesta mesma figura podemos notar como é simples obter a média, soma, maior e menor valores de um objeto *ndarray* NumPy.

5.1.2 Pandas

Pandas é uma biblioteca em que na definição do próprio Time de Desenvolvimento pandas (2020) é rápida, poderosa, flexível, de código aberto e de fácil uso voltada para análise e manipulação de dados e construída para a linguagem de programação Python. No artigo de sua criação, McKinney (2010) diz que esta biblioteca foi criada com o intuito de facilitar o trabalho com diversos conjuntos de dados mesmo que em formatos diferentes e prover um conjunto de blocos fundamentais para a construção de modelos estatísticos. Albon (2018) corrobora com isso ao dizer que esta biblioteca permite criar um conjunto de observações claras e bem estruturadas para serem processadas.

Na prática, essa biblioteca é útil pois pode importar ou exportar um conjunto de dados de uma série de origens, destinos e formatos, alguns deles sendo: *csv, xls, parquet, html, hdf5, json, gbq* e *sql* (Time de Desenvolvimento pandas, 2020). Independente da fonte de importação, ou do tipo de dados em cada coluna, é gerado um objeto de uma classe chamada DataFrame que estrutura as informações em linhas e colunas, como em uma planilha ou tabela de banco de dados, e nela podem ser realizadas diversas operações. De acordo com Müller e Guido (2016), a biblioteca pandas é toda construída em torno desta estrutura de dados, que foi modelada tomando como base o modelo DataFrame da linguagem R.

Como exemplo disso, na Figura 20 tempos um trecho de código em que foi carregado um conjunto de dados em formato *csv* para um objeto DataFrame chamado de *df*. O método *head* mostra os primeiros *n* resultados desse DataFrame sendo neste caso os primeiros 5. A montagem gráfica da tabela a partir deste resultado é feito automaticamente pelo Jupyter Notebook utilizado e que será abordado na Subseção 5.1.5.

Figura 20 – Exemplo de importação de dados utilizando biblioteca Pandas

```
import pandas as pd
df = pd.read_csv('./datasets/EXEMPLO.csv')
df.head(5)
```

	DEPOSITO	ANO	MES	DIA	PEDIDOOID	PRODUTOOID	QUANTIDADE	TIPOESTRUTURAOID
0	1	2016	8	5	278	188409	1.0	NaN
1	1	2016	8	5	278	188411	1.0	NaN
2	1	2016	8	5	314	139603	100.0	7938926.0
3	1	2016	8	26	804	141181	5.0	7938926.0
4	1	2016	8	26	984	141181	5.0	7938926.0

Fonte: O autor.

Dentre algumas das operações que podem ser realizadas em um DataFrame, é possível dividir colunas ou linhas a partir de um DataFrame em outros DataFrames, ou o inverso, jun-

tando vários em um só. Também é possível até mesmo realizar operações familiares a linguagem SQL e bancos relacionais, como *SELECT*, *GROUP BY* e *JOIN*. Além disso, é extremamente fácil realizar operações em escala, criando novas colunas a partir de uma transformação em uma das colunas. Tudo isso cria uma base muito prática e dinâmica para se trabalhar com conjuntos de dados em Python.

Figura 21 – Exemplo de seleção e agrupamento de dados utilizando biblioteca Pandas

```
df2 = df.loc[df['DEPOSITO'].isin(["1"])].loc[df['ANO'] >= 2018].loc[df['ANO'] < 2020]
df2 = df2.groupby(['ANO', 'MES', 'DIA'])['QUANTIDADE'].sum().reset_index()
df2.head(5)
```

	ANO	MES	DIA	QUANTIDADE
0	2018	1	2	18035.0
1	2018	1	3	13537.0
2	2018	1	4	14765.0
3	2018	1	5	8895.0
4	2018	1	8	14321.0

Fonte: O autor.

Como exemplo de uma dessas transformações, com base no DataFrame carregado na Figura 20, na Figura 21 realizamos três operações *loc* que funcionam de forma similar a um *SELECT* em SQL. Neste caso, buscamos apenas os registros da coluna DEPOSITO que tenham o valor 1 e registros entre os anos de 2018 e 2020. Na próxima linha, agrupamos os registros que tinham o mesmo valor para as colunas ANO MES e DIA somando a QUANTIDADE no registro agrupado. Isso resultou em um conjunto de dados da quantidade total diária e sem as colunas não utilizadas no agrupamento. Essas operações demonstram alguns usos dessa biblioteca e sua praticidade.

5.1.3 Scikit-Learn

Scikit-Learn é um projeto de código aberto, constantemente desenvolvido e melhorado com uma comunidade bastante ativa. Contém diversos algoritmos de ponta para Aprendizagem de Máquina acompanhados de uma boa documentação, sendo muito utilizado tanto na indústria quanto na academia, segundo Müller e Guido (2016).

Segundo Pedregosa et al. (2011), essa biblioteca pode ser utilizada para redimensionamento de vetores utilizando algoritmos como K-Means, selecionar modelos diferentes fazendo comparação cruzada de métricas, pré-processamento de dados, como extração de características e normalização, classificação de dados e regressões lineares, entre outros, demonstrando ser uma biblioteca extremamente completa e robusta.

5.1.4 MatPlotLib

Segundo Hunter (2007), MatPlotLib é uma biblioteca para criação de gráficos de duas dimensões para Python que produz figuras com qualidade suficiente para serem publicadas em pesquisa disponibilizando uma grande variedade de formas para demonstrar dados, buscando ter uma API e visuais similares ao *MATLAB*. Müller e Guido (2016) classifica esta como a principal biblioteca para plotagem científica em Python.

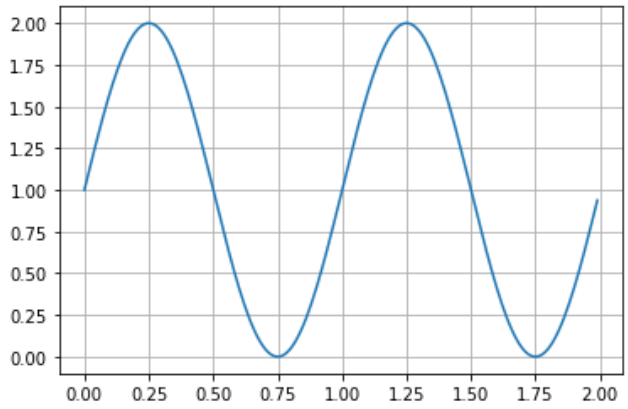
Figura 22 – Gerando gráfico com MatPlotLib

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 2.0, 0.01)
s = 1 + np.sin(2 * np.pi * t)

fig, ax = plt.subplots()
ax.plot(t, s)
ax.grid()

fig.savefig("teste.png")
plt.show()
```



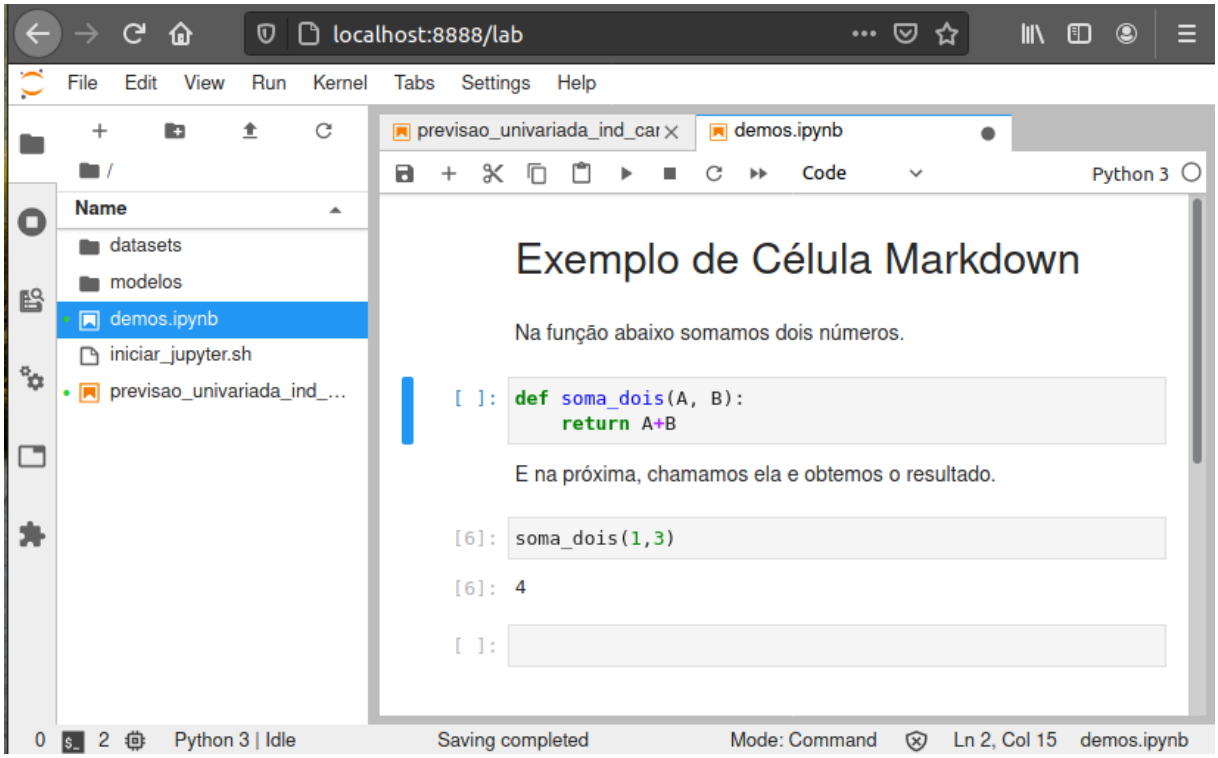
Fonte: O autor.

Podemos ter uma ideia de seu uso ao analisarmos o código e gráfico presentes na Figura 22. Nela, utilizando também a biblioteca NumPy, foi criado linhas t e s que representam uma onda senoidal. A biblioteca MatPlotLib se baseia na ideia de termos uma figura com *plots* e nesses *plots* temos *axis* (eixos) em que podemos gerar gráficos através de vetores ou listas de dados. Ainda é possível definir linhas como feito nesta Figura através do método *grid*, salvar a imagem com *savefig* e mostrar a imagem com *show*. Esta foi a biblioteca utilizada para a geração dos gráficos apresentados neste trabalho.

5.1.5 Jupyter Notebook

Jupyter Notebook é uma interface visual baseada em web voltado ao público de pesquisas que torna possível em uma mesma ferramenta, interagir com códigos Python em tempo real, e criar uma documentação do que está ocorrendo, usando inclusive recursos da linguagem *Markdown*. Também é de fácil compartilhamento desses cadernos quando exportados para formato JSON, suportando inclusive o fluxo de desenvolvimento de algoritmos de Aprendizagem de Máquina, segundo Kluyver et al. (2016). Além disso, possui adaptações para algumas bibliotecas especificas, mostrando resultados de um DataFrame de pandas em forma visual de tabela e mostra os gráficos criados com MatPlotLib diretamente na página web.

Figura 23 – Exemplo de uso do Jupyter Notebooks



Fonte: O autor.

É possível instalá-lo diretamente pelo gerenciador de programas do Python, o *pip* e ao ser inciado ele cria servidor web local. Ao acessá-lo através de um navegador, pode-se criar cadernos de anotações e em cada caderno são adicionadas células que podem ser Markdown ou código Python, que neste caso se comporta como um terminal Python próprio.Podemos ver o uso destas células e como funciona esse ambiente na Figura 23 Moolayil (2019) recomenda fortemente o uso de Jupyter Notebooks pela conveniência que ele provê na exploração, análise e reprodutibilidade durante pesquisas em Aprendizagem de Máquina. Esta foi a ferramente utilizada para criação das imagens de código demonstradas ao longo deste trabalho.

5.2 TensorFlow e Keras

Moolayil (2019) afirma que podemos classificar o nível de abstração de um *framework* de Aprendizagem de Máquina em baixo nível e alto nível. Embora a linha que separa essas declarações seja tênue, podemos ter uma ideia do que esperar de um *framework* de acordo com qual destas ele mais se relaciona. Quando for alto nível, espera-se maior abstração e facilidade de uso, enquanto que quando baixo nível, espera-se mais capacidade de customização mas requerendo maior conhecimento para seu uso.

Dentro dos *frameworks* de baixo nível para construção de Redes Neurais os mais utilizados são Theano, Torch (e PyThorch), MxNet e Tensorflow. Dentre estes, Tensorflow é o mais conhecido e adotados na indústria para Redes Neurais, de acordo com Moolayil (2019), sendo inclusive utilizada por gigantes da tecnologia como Google, Twitter, Intel, Deepmind entre outros.

Abadi et al. (2015) definem Tensorflow como uma plataforma de código aberto que possui um grande ecossistema de ferramentas e recursos e que pode ser utilizada em mais de uma linguagem, uma delas sendo Python. Foi desenvolvida inicialmente por pesquisadores da *Google Machine Intelligence Research Organization* para uso interno da empresa e então tornada pública em 9 de Novembro de 2015 sob a licença Apache 2.0.

Possui uma excelente documentação e comunidade e é muito completa e robusta, permitindo opções avançadas como processamento paralelo em GPUs, que podem ser críticas para aumentar velocidade de treinamento com grandes conjuntos de dados.

Já Keras, é uma API de alto nível, código aberto, inicialmente desenvolvida pelo projeto ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System). Esta API abstrai e facilita o uso de vários *framworks* de baixo nível aqui abordados, sendo um destes o Tensorflow. Porém, é importante dizer que essa API tem uma integração especial com o Tensorflow, sendo tão forte que a partir da versão 2.0 do Tensorflow, passou a fazer parte dele próprio, sendo importada junto em código e servindo como uma API oficial.

Sua utilização facilita todo desenvolvimento e uso dos métodos e algoritmos do Tensorflow, pois é possível focar na arquitetura da rede em sí, deixando as partes mais específicas e técnicas abstraídas. É extremamente utilizada, inclusive por gigantes como CERN, NASA, NIH (National Institutes of Health) e até no colisor de partículas, LHC de acordo com Chollet et al. (2015).

A forma mais comum de se trabalhar com Keras é utilizando o modelo Sequencial de sua biblioteca, em que podemos empilhar camadas e definir hiperparâmetros, dessa forma formando uma rede neural. Esse modelo se torna um objeto no código, que pode ser utilizado para ser treinado sobre certo conjunto de dados e depois realizar previsões. Também é possível salvar a estrutura inteira em arquivo, até mesmo em formato JSON, tornando simples sua passagem de ambiente de testes e treinos para um ambiente de produção.

Para Chollet et al. (2015) só não se deve utilizar o modelo Sequencial quando o modelo

em sí tem múltiplas entradas ou saídas, quando uma ou mais camadas tem múltiplas entradas ou saídas, ou é necessário fazer compartilhamento de camadas. Todos esses casos são usos mais avançados para modelos grandes e complexos em que é interessante dividir uma rede em várias diferentes partes que podem ser reaproveitadas, de acordo com Géron (2019). Como a rede neural deste trabalho não necessita de nenhuma dessas características, o modelo Sequencial pode ser utilizado perfeitamente.

Figura 24 – Exemplo de criação de modelo de RNR com Keras

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(350, input_dim = 14, activation="relu"))
model.add(Dense(350, activation="relu"))
model.add(Dense(350, activation="relu"))
model.add(Dense(350, activation="relu"))
model.add(Dense( 1,  activation="linear"))

# Cria o modelo
model.compile(optimizer='adam', loss="mean_squared_error")
# Treina o modelo
model.fit(x, y, validation_data=(x_val, y_val),
          epochs=20, batch_size=64)
```

Fonte: O autor.

Para demonstrar o seu uso, podemos observar o trecho de código presente na Figura 24. Nela, após ser importadas as classes *Sequential* e *Dense* da biblioteca Keras, que como dito agora faz parte do próprio TensorFlow, instanciamos a classe *Sequential* para um objeto chamado modelo. À este modelo, adicionamos camadas, neste caso densas, e definimos as dimensões da entrada, que neste caso são 14, ou seja, seria para entrada de um vetor de 14 elementos. Também é definida a cada camada a quantidade de neurônios pelo parâmetro *units* e a função de ativação pelo parâmetro *activation*.

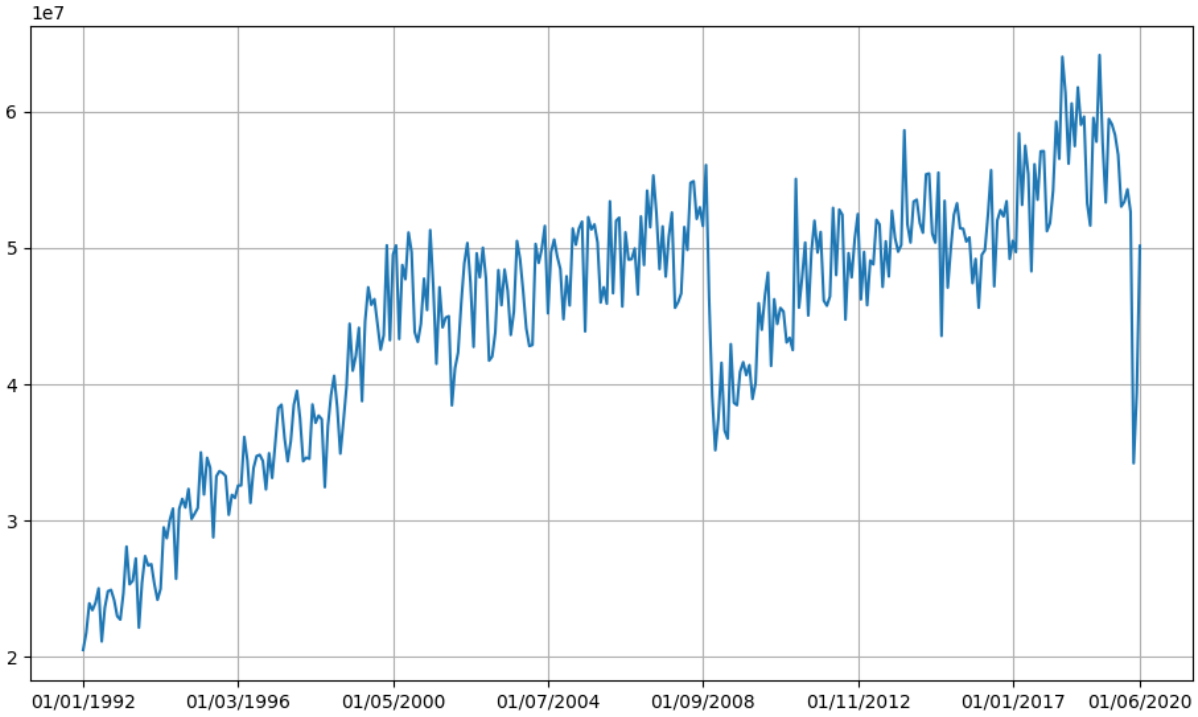
Após termos definido o modelo, juntamos todas as camadas através do método *compile* onde também definidos a função de otimização e a função de perda. Por fim, utilizando o método *fit* treinamos o modelo passando as variáveis de treino e validação, número de épocas e tamanho de lote. Devido ao ecossistema gigante de ferramentas e documentação, grande uso na indústria e integração forte entre um *framework* de alto nível e uma biblioteca de baixo nível, a combinação de Tensorflow e Keras foi a utilizada para construção das Redes Neurais apresentadas neste trabalho.

6 CONJUNTOS DE DADOS DAS INDÚSTRIAS CANADENSES

Para validação das técnicas de previsão de demanda deste trabalho, buscou-se um conjunto de dados público, presente na cadeia logística, e que já tivesse sido utilizado em outros trabalhos acadêmicos. Para atingir todos esses critérios, foi escolhido o conjunto de dados de novos pedidos mensais das indústrias Canadenses.

Este conjunto é conhecido como *Manufacturers' monthly new orders by North American Industry Classification System* e são mantidos pela instituição Canadense Monthly Survey of Manufacturing (MSM). Esses dados foram obtidos através de seu site oficial em STATISTICS CANADA (2020) buscando-se pela tabela 16-10-0047-01.

Figura 25 – Demanda da Indústria Canadense (01/1992 - 06/2020)



Fonte: O autor.

Neste conjunto está presente a demanda agrupada dos mais diversos setores da indústria daquele país com dados disponíveis desde Janeiro de 1992. A coleta destes dados começa aproximadamente 7 dias úteis após o término do mês de referência e de acordo com o próprio MSM seus dados são utilizadas tanto pelo setor público quanto o privado, incluindo departamentos do governo federal Canadense, o Bank of Canada (Banco central Canadense), consultores e casas de pesquisa de todo mundo.

Figura 26 – Importação dos dados da Indústria Canadense

```
import pandas as pd
df = pd.read_csv("./datasets/NAICS_number_of_orders_not_seasonally_ajusted.csv")
df.head(3)
```

	DATA	DEMANDA
0	01/01/1992	20532164
1	01/02/1992	21792493
2	01/03/1992	23948450

```
datas = df['DATA'].values
dados = df.drop(columns=['DATA'])
dados = dados['DEMANDA'].tolist()
```

Fonte: O autor.

Como primeiro passo de análise, foram importados os dados do conjunto em formato *csv* utilizando a biblioteca pandas, através do código da Figura 26. Em seguida, foi feito sua plotagem através da biblioteca MatPlotLib. Podemos observar na Figura 27 este processo, em que primeiro definimos um vetor para legenda dos pontos que irão aparecer no eixo x e em que momento irão aparecer. Isso foi feito no laço de repetição *for*. Em sequência, foi montada uma figura em que passamos os dados carregados do conjunto definindo parâmetros estéticos para mais fácil visualização, como fonte e tamanho da figura.

Ao analisarmos estes dados na Figura 25, notou-se uma tendência de subida permeada por intervalos alternados de subidas e quedas, que podem ser identificados como sazonalidades. Além dessa tendência, pode ser claramente notado dois períodos de queda brusca na demanda. O primeiro deles, representa o impacto causado na indústria durante a recessão que iniciou-se em 2008. O segundo, em início de 2020, registra o impacto causado pela pandemia da COVID-19.

Figura 27 – Plotagem da Demanda da Indústria Canadense

```
import matplotlib.pyplot as plt


ticks = []
indice_grafico = []
for i in (range(0, len(datas), 50)):
    indice_grafico.append(datas[i])
    ticks.append(i)
indice_grafico.append(datas[len(datas)-1])
ticks.append(len(datas)-1)

fig, axs = plt.subplots(1, figsize=(11, 5.5))
axs.plot(dados)
axs.set_xticks(ticks)
axs.set_xticklabels(indice_grafico)
axs.yaxis.grid(True)
axs.xaxis.grid(True)
plt.subplots_adjust(left=0.1, right=0.9, top=0.95, bottom=0.05)
plt.rcParams.update({'font.size': 10})
fig.savefig("./figuras/dados_ind_canad.png")
plt.show()
```

Fonte: O autor.

6.1 Preparação dos Dados

Para realizar as previsões, foram necessárias algumas adaptações aos dados, principalmente para os métodos de I.A. Primeiramente, foi realizada a normalização dos dados utilizando o método MinMaxScaler da biblioteca Scikit-Learn no intervalo $[0,1]$. Isso foi extremamente importante para a previsão do modelo LSTM que frente a testes realizados, após a normalização este modelo obteve um Erro Quadrático Médio muito menor.

Figura 28 – Normalização dos dados

```
from sklearn.preprocessing import MinMaxScaler
import numpy as np

dados_escalados = np.array(dados).reshape(-1, 1)
escalador = MinMaxScaler(feature_range=(0, 1))
dados_escalados = escalador.fit_transform(dados_escalados)
dados_escalados = dados_escalados.ravel()
print(dados_escalados[0:3])
```

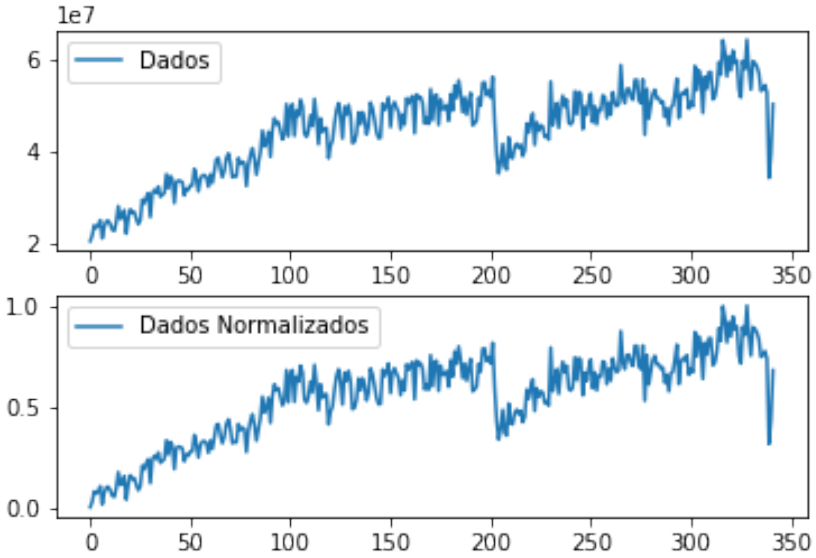
```
[0.         0.02889657 0.07832792]
```

Fonte: O autor.

O código para esta normalização pode ser encontrado na Figura 28. Como o método MinMaxScaler precisa receber um vetor NumPy de duas dimensões, nesse trecho também existe

essa transformação. Após isso, utilizando o método *reshape*, definirmos o intervalo dos dados criando um objeto *escalador* e utilizando o método *fit_transform* criamos nosso novo vetor normalizado. O último passo foi apenas re-transformar esse vetor em unidimensional através do método *ravel*. Foi também plotado uma comparação entre os dois vetores, o normalizado e o não normalizado e fica evidente ao compararmos ambos na Figura 29 que apesar do intervalo ter sido alterado, as proporções continuam as mesmas.

Figura 29 – Comparação dos dados normalizados com não normalizados



Fonte: O autor.

Após essa normalização, o conjunto foi divido utilizando as proporções dos 80% iniciais para testes e os 20% restantes para validação. É importante realizar essa divisão e manter os dados de validação separados dos dados de treino para testar a real acurácia quando o modelo se depara com novos dados, podendo assim percebermos indícios de *overfitting*. Este código e uma visualização da divisão podem ser observados na Figura 30.

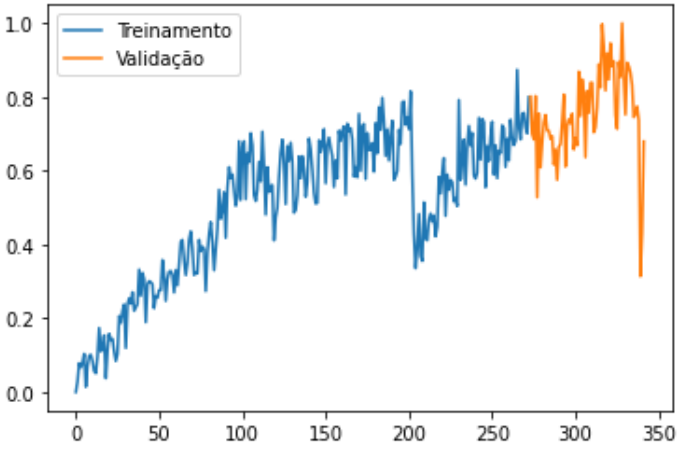
Figura 30 – Divisão dos dados entre conjunto e treinamento

```
percent_treino = 0.8 # ex: 0.8 = linha do 80%
n_linhas = int(percent_treino * len(dados_escalados))

x_treinar = dados_escalados[:n_linhas]
x_validar = dados_escalados[n_linhas:]

indice_grafico_treinar = datas[:n_linhas]
indice_grafico_validar = datas[n_linhas:]

x_treinar_index = range(0, len(x_treinar))
x_validar_index = range(len(x_treinar), len(x_treinar)+len(x_validar))
plt.plot(x_treinar_index, x_treinar, label='Treinamento')
plt.plot(x_validar_index, x_validar, label='Validação')
plt.legend()
plt.show()
```



Fonte: O autor.

Em sequência, foi necessário montarmos os vetores utilizados como entradas para os métodos de I.A. Essa montagem foi realizada utilizando o método *concatenar_tempos_anteriores* presente na Figura 31. Nele, verificamos quantos conjuntos completos de entrada teremos a partir do vetor recebido é então são criados dois novos vetores, um deles com os *t* tempos anteriores e outro com o próximo tempo.

Figura 31 – Método para concatenar tempos anteriores.

```
def concatenar_tempos_anteriores(t, concat_dados):
    x_concatenado = []
    y_concatenado = []
    diferenca = len(concat_dados)%t
    for i in range(t+diferenca, len(concat_dados)):
        x_concatenado.append(concat_dados[i-t:i])
        y_concatenado.append(concat_dados[i])
    return x_concatenado, y_concatenado
```

Fonte: O autor.

Chamando este método, criamos então os conjuntos de entrada para a Regressão Linear e Rede Neural, também diminuindo o índice de legendas para ficar no novo tamanho, que depois utilizamos na plotagem de gráficos como na Figura 32. Alterando a variável *tempos_anteriores* para 1 teremos um conjunto univariado, e aumentando para no mínimo 2 já temos um conjunto multivariado. O valor de *tempos_anteriores* representa o número de meses de entrada ao modelo.

Figura 32 – Utilizando método para concatenar tempos anteriores.

```
tempos_anteriores = 6

x_treinar_concat, y_treinar_concat = concatenar_tempos_anteriores(tempos_anteriores, x_treinar)
x_validar_concat, y_validar_concat = concatenar_tempos_anteriores(tempos_anteriores, x_validar)

indice_grafico_treinar = indice_grafico_treinar[tempos_anteriores-1:]
indice_grafico_validar = indice_grafico_validar[tempos_anteriores-1:]
```

Fonte: O autor.

Para análise dos dados nas tabelas, devemos lembrar que esse conjunto em específico está na casa das dezenas de milhões. Sendo assim, os dados apresentados nas tabelas e gráficos aqui presentes serão todos tratados como milhões.

Além disso, o EQM representa o valor da métrica em sí, porém a Raiz do Erro Quadrático Médio (REQM) nos mostra em valores mais palpáveis de quanto foi o erro, removendo o quadrado do EQM. Isso significa que um REQM de 5 nesse conjunto representa um erro médio do período de 5 milhões.

Tanto a Regressão Linear quanto a rede neural receberam como parâmetros para seu treino o conjunto de dados de entrada x e o conjunto de dados y sendo que o primeiro contém os dados disponíveis para previsão e o segundo quais os dados reais que originaram do primeiro. Realizando suas computações encima destes dois conjuntos, ambas buscam encontrar a função $f(x) = y$ sobre os dados de treino para aplicar sobre os dados de validação.

Uma amostragem do conjunto de entrada é mostrado na Tabela 1. As colunas Demanda t_{-1} e Demanda t representam, respectivamente, a demanda do mês anterior ao mês a ser previsto e a demanda do próprio mês a ser prevista. Já as colunas X e Y representam o mesmo, porém escaladas para o intervalo $[0,1]$ e são elas que foram passadas às redes para treinamento.

Tabela 1 – Entrada Univariada das Indústrias Canadenses

Registro	Data	Demanda t_{-1}	Demanda t	X	Y
0	01/02/1992	20,5322	21,7925	0	0,0288
1	01/03/1992	21,7925	23,9485	0,0288	0,0783
2	01/04/1992	23,9485	23,4467	0,0783	0,0668
3	01/05/1992	23,4467	23,9918	0,0668	0,0793
...	...	...	...	...	...
268	01/06/2014	53,4080	53,5470	0,7537	0,7569
269	01/07/2014	53,5470	51,8331	0,7569	0,7176
270	01/08/2014	51,8331	51,1185	0,7176	0,7012
271	01/09/2014	51,1185	55,4018	0,7012	0,7994

Fonte: O autor.

6.2 Previsões Univariadas

Após realizada a preparação dos dados, foram realizadas as previsões Ingênua, Média Móvel Exponencial, Regressão Linear e LSTM. Neste Capítulo está presente a metodologia usada em cada uma destas assim como seus resultados.

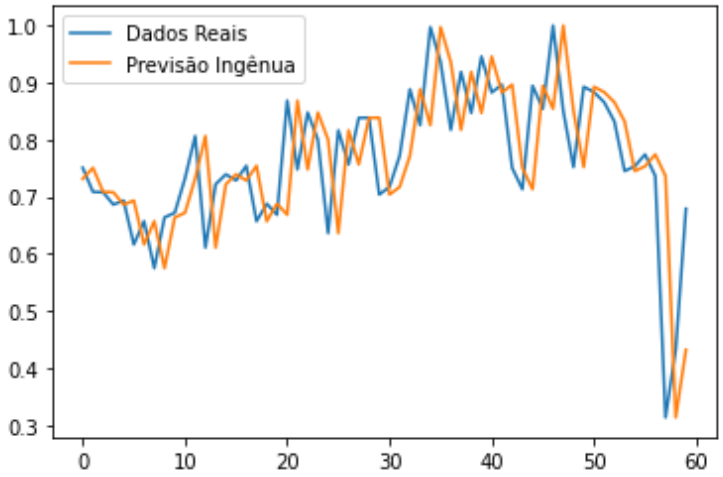
6.2.1 Previsão Ingênua

Para realizar a mais simples das previsões e que serviu como nosso *benchmark*, a previsão ingênua, realizou-se um "deslizamento" dos dados em uma casa no vetor através do código presente na Figura 33 assim como sua visualização em gráfico. Além de "deslizar" o registro em um mês, foi reduzido o vetor ao tamanho do utilizado na validação dos métodos de I.A. para garantir uma métrica justa. Foram testadas previsões ingênuas de outros períodos, uma delas sendo o valor do mesmo ano no mês anterior, porém todas tiveram uma performance inferior, sendo a aqui apresentada a utilizada como *benchmark*.

Figura 33 – Previsão Ingênua das Indústrias Canadenses.

```
previsao_ingenua = dados_escalados[:-1]
indice_previsoes_classicas = (len(y_validar_concat))*-1
previsao_ingenua = previsao_ingenua[indice_previsoes_classicas:]

plt.plot(y_validar_concat, label='Dados Reais')
plt.plot(previsao_ingenua, label='Previsão Ingênua')
plt.legend()
plt.show()
```



Fonte: O autor.

6.2.2 Previsão da Média Móvel Exponencial

Para realizar a previsão da Média Móvel Exponencial (MME) sobre o conjunto de dados foi utilizada uma função de autoria própria chamada *media_movel_exponencial*, seguindo o que foi apresentado na Subseção 2.1.1 e que pode ser verificada na Figura 34.

Figura 34 – Método de Previsão da Média Móvel Exponencial

```
def media_movel_exponencial(x, S_inicial, alpha, beta):
    S = S_inicial # Media simples do periodo anterior deve ser o inicial de S
    T = 0 # Tendência para primeira previsão é 0
    y = []

    for i in range (1, len(x)+1):
        A = x[i-1] # Valor real do tempo anterior
        S_anterior = S

        S = (alpha * A) + ((1 - alpha) * (S_anterior + T))
        S = round(S, 4)

        T = beta * (S - S_anterior) + (1 - beta) * T
        T = round(T, 4)

        F = S + T
        F = round(F, 4)

        y.append(F)

    return y
```

Fonte: O autor.

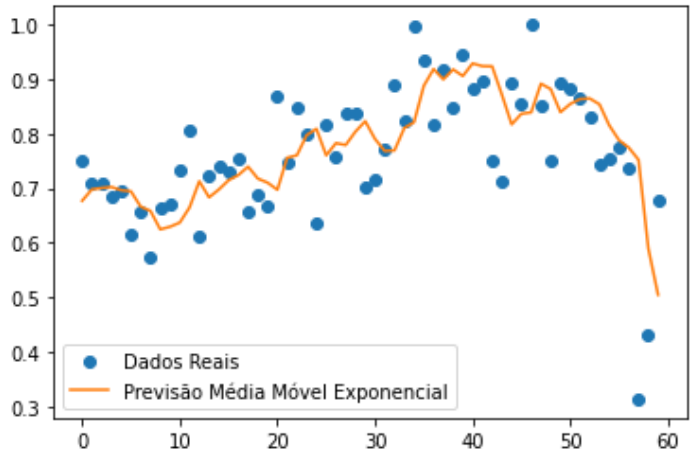
Foi considerado o primeiro ano dos dados de entrada como a primeira sazonalidade desse modelo e a média simples destes primeiros doze registros do conjunto tornou-se o valor passado ao parâmetro *S_inicial*. A chamada deste método pode ser verificada na Figura 35, assim como a criação da média do primeiro ano com o método *average* da biblioteca NumPy e o encurtamento do vetor para o mesmo tamanho do utilizado para validação das previsões de I.A.

Como foi utilizado o primeiro ano como parâmetro inicial para a média, só teremos previsões válidas nesse modelo a partir do décimo terceiro registro. Isso não atrapalha a sua métrica porque validamos os dados com os 20% finais do conjunto, que não inclui o ano inicial.

Figura 35 – Método de Previsão da Média Móvel Exponencial

```
# Envia ultimo registro do periodo anterior mais todos a serem previstos
x_media_movel = dados_escalados[12:-1]
# Média do primeiro ano
s_inicial = np.average(dados_escalados[:12])
previsao_mme = media_movel_exponencial(x_media_movel, s_inicial, 0.3, 0.15)
previsao_mme = previsao_mme[indice_previsoes_classicas:]

plt.plot(y_validar_concat, 'o', label='Dados Reais')
plt.plot(previsao_mme, label='Previsão Média Móvel Exponencial')
plt.legend()
plt.show()
```



Fonte: O autor.

Parâmetros importantes para a acurácia desse modelo são os valores de *alpha* e *beta*. Buscando encontrar os melhores valores, foi realizada uma sequência de previsões alterando-se os valores de *alpha* e *beta* até ser escolhido o par que teve a menor Raiz do Erro Quadrático Médio (REQM) frente aos dados reais. Os resultados estão demonstrados na Tabela 2 estando ordenados pela menor REQM, sendo os valores $\alpha = 0.3$ e $\beta = 0.15$ os que obtiveram melhor desempenho.

Tabela 2 – Resultados para Média Ponderada Exponencial conforme *alpha* e *beta*

alpha	*beta*	REQM	EMA
0,30	0,15	4,4343	3,2211
0,20	0,20	4,4487	3,1978
0,30	0,20	4,4567	3,2600
0,20	0,30	4,4635	3,2568
0,15	0,30	4,4729	3,2197
0,20	0,15	4,4708	3,1927
0,15	0,20	4,5277	3,1936
0,30	0,30	4,5405	3,3485
0,15	0,10	4,7024	3,2696
0,10	0,15	4,9265	3,4403

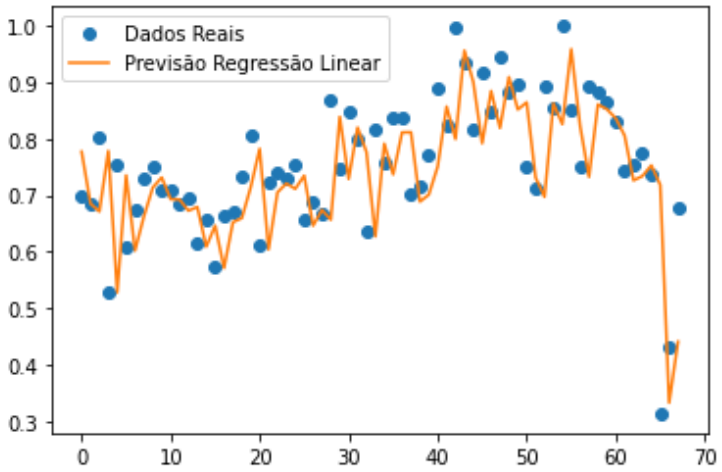
Fonte: O autor.

6.2.3 Previsão da Regressão Linear

Figura 36 – Método de Previsão da Regressão Linear Univariada

```
from sklearn.linear_model import LinearRegression
modelo_regressao = LinearRegression()
modelo_regressao.fit(x_treinar_concat, y_treinar_concat)
previsao_reg_lin = modelo_regressao.predict(x_validar_concat)

plt.plot(y_validar_concat, 'o', label='Dados Reais')
plt.plot(previsao_reg_lin, label='Previsão Regressão Linear')
plt.legend()
plt.show()
```



Fonte: O autor.

Utilizando a classe *LinearRegression* da biblioteca Sklearn foi criado um modelo de Regressão Linear conforme a Figura 36. Após ser treinada com o método *fit*, esta classe retorna

um objeto modelo já adaptado à regressão do conjunto de dados de treino. Utilizando esse objeto, chamamos o método *predict* para realizar a previsão. Para previsão, foi passado apenas o conjunto de entrada dos 20% destinados à validação.

6.2.4 Previsão da LSTM

Dentro da construção de uma rede neural, existem diversas formas diferentes que poderíamos tela estruturado, e portanto, escolhemos testar estes modelos. Diferente das outras previsões vistas até agora nesse capítulo, as redes neurais não são determinísticas, pois utilizam geradores aleatórios para definir os pesos iniciais e os gradientes de otimizações da rede, o que afetam o resultado final a cada treinamento. Além disso, diferentes dos outros, as LSTMs precisam ter como entrada um vetor de 3 dimensões, correspondentes à quantidade de amostras, períodos temporais, e características.

Para garantir que estaríamos avaliando de fato alterações na performance do modelo e não só da aleatoriedade dos números selecionados naquela vez, no momento da seleção da melhor arquitetura foi definida a semente de aleatoriedade 1234 fixa para bibliotecas Numpy e Tensorflow, garantindo a replicabilidade do treinamento dos modelos durante avaliações. No código presente na Figura 37 foi definida esta semente aleatória, assim como transformados os vetores de 1 para 3 dimensões.

Figura 37 – Preparação dos vetores para treino na LSTM.

```
import tensorflow as tf
tf.compat.v1.disable_eager_execution()
# Define semente aleatória para poder replicar resutlados
np.random.seed(1234)
tf.random.set_seed(1234)
# Transforma em 3d
np3d_x_treinar_concat = np.array(x_treinar_concat)
n_aspectos = np3d_x_treinar_concat.shape[1]
np3d_x_treinar_concat = np.reshape(x_treinar_concat,
                                   newshape=(-1, n_aspectos, 1))

np_y_treinar_concat = np.array(y_treinar_concat)
```

Fonte: O autor.

Antes de debatermos quais foram as diferenças entre os métodos avaliados, é necessário falarmos sobre as semelhanças. Após alguns testes, algumas características ficaram claramente demonstradas como mais performáticas no nosso contexto e não foram mais alteradas para os testes finais apresentados. O otimizador escolhido para todos foi o Nadam e não existiu razão para ser trocado, a não ser o seu padrão para o coeficiente de aprendizagem. Este foi alterado do padrão de $0{,}001$ para $0{,}01$. Isso porque durante os testes dos hiperparâmetros inicias

ficou claro que para esse conjunto em específico era necessário um coeficiente de aprendizagem maior, que altera com mais intensidade os pesos a cada *back-propagation*.

Percebeu-se também uma melhora geral ao ser utilizado o regularizador de Kernel L2 com o parâmetro padrão $0{,}01$. Além deste, outra regularização utilizada foi a *Early Stopping* com nível de paciência de 3 monitorando a métrica de *loss*. Isso significa que caso o modelo passasse 3 épocas do treinamento sem melhorar sua métria o treinamento era parado abruptamente.

Foram realizados testes com camadas densas ao invés de camadas LSTM, mas sua performance foi bem inferior, nos levando a de fato manter por padrão a LSTM. Também foram testadas camadas de *Dropout* mas para este modelo sua performance sempre piorou ao serem utilizadas. Todos os treinos foram realizados com o parâmetro de 300 épocas e como métrica para coeficiente de perda foi utilizado o EQM.

Por fim, foram testados modelos com mais camadas, mas estes não apresentaram melhoras significativas, sendo mantida apenas uma. Também foram testados modelos com quantidade muito maior de neurônios do que os apresentados na Tabela 3 porém da mesma forma, não mostraram melhora significativa.

Tabela 3 – Métricas das Previsões Univariadas Sobre o Conjunto das Indústrias Canadenses

Neurônios	Ativação	Inicialização	Dropout	REQM
2	TanH	Glorot	Não	4,7988
2	SELU	LeCun Normal	Não	4,9441
4	TanH	Glorot	Não	4,9852
4	SELU	LeCun Normal	Não	5,9891
2	TanH	Glorot	0,1	7,8171
4	SELU	LeCun Normal	0,1	8,2072

Fonte: O autor.

Quanto aos hiperparâmetros que foram avaliados entre sí e estão demonstrados na Tabela 3, foram a função de ativação, quantidade de neurônios e inicializadores de kernel. Na imagem Figura 38 temos o código de criação de uma rede neural. Nele, podemos alterar os hiperparâmetros testados. Após cada alteração, precisamos retreinar o modelo conforme código na Figura 39 em que temos a chamada ao método *fit* e a demonstração do retorno que o método nos dá em console.

Analisando a Tabela 3 com os resultados das comparações, podemos perceber uma melhor performance da rede com dois neurônios, ativação TanH e inicializador de Kernel Glorot Uniforme. O número de neurônios vai de acordo com a regra demonstrada na Subseção 3.4.7 em que por termos apenas um parâmetro de entrada para a previsão univariada, um número bom para se iniciar o processo de treinos seriam 2 neurônios.

Figura 38 – Criação do modelo de LSTM univariada.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from tensorflow.keras import regularizers, callbacks, optimizers

# Cria o modelo
modelo = Sequential()
modelo.add(LSTM(units=2,
                input_shape=(n_aspectos, 1),
                activation="tanh",
                kernel_initializer="glorot_uniform",
                kernel_regularizer=regularizers.l2(0.01)))
modelo.add(Dense(1))
#Early Stopping
callback = callbacks.EarlyStopping(monitor='loss', patience=5)
# Nadam configs
nadam = optimizers.Nadam(learning_rate=0.01)
# Compila o modelo
modelo.compile(optimizer=nadam,
               loss="mean_squared_error")
```

Fonte: O autor.

Figura 39 – Treinamento de modelo de LSTM.

```
# Treina o modelo
modelo.fit(np3d_x_treinar_concat, np_y_treinar_concat,
           callbacks=[callback],
           epochs=300,
           verbose=1,
           batch_size=32)
```

```
Train on 272 samples
Epoch 1/300
272/272 [==============================] - 0s 71us/sample - loss: 0.2392
Epoch 2/300
272/272 [==============================] - 0s 60us/sample - loss: 0.1454
Epoch 3/300
272/272 [==============================] - 0s 60us/sample - loss: 0.0752
Epoch 4/300
272/272 [==============================] - 0s 55us/sample - loss: 0.0439
```

Fonte: O autor.

A função de ativação ReLU também foi testada, porém muitas vezes ocorria o problema das "unidades mortas" já abordado na Subseção 3.4.5. Por isso, como opção da família dos algoritmos ReLU foi optada pela SELU. É importante dizer que apesar deste parâmetros terem apresentado os melhores resultados no ambiente de semente controlada, ao ser testado

com a aleatoriedade própria e normal das redes neurais, ou seja, sem semente definida, houve variação na acurácia das previsões.

Figura 40 – Carregamento modelo de LSTM.

```
from tensorflow.keras.models import load_model
modelo = load_model('./modelos/lstm_uni.h5')
```

Fonte: O autor.

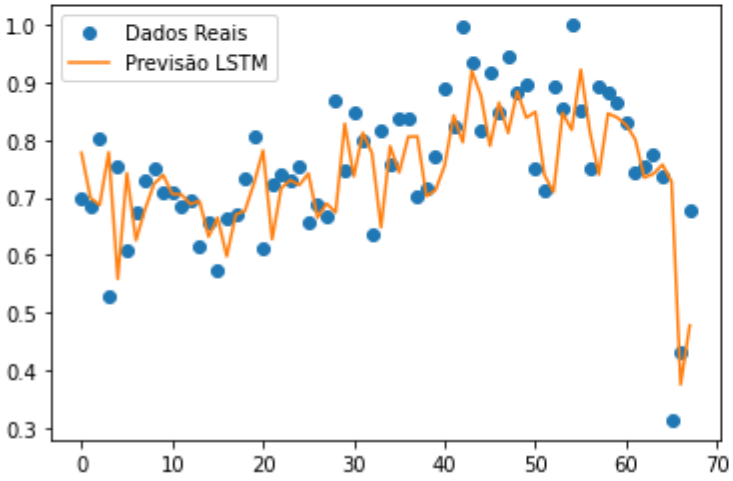
O resultado apresentado na Tabela 4 para a LSTM foi a melhor performance obtida após 10 treinamentos. Em média, a acurácia não ficou muito distante deste, porém em algumas dessas execuções sua acurácia equiparou-se a modelos inferiores, até mesmo com a previsão ingênua. Este modelo foi salvo e carregado com o código presente na Figura 40 e seus resultados, assim como a forma com a qual as previsões via LSTM são obtidas, podem ser observados na Figura 41.

Figura 41 – Previsão utilizando modelo de LSTM univariada.

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3' # Remove avisos de deprecated

np3d_x_validar_concat = np.array(x_validar_concat)
np3d_x_validar_concat = np.reshape(np3d_x_validar_concat,
                                   newshape=(-1,n_aspectos, 1))
previsao_lstm = modelo.predict(np3d_x_validar_concat)

plt.plot(y_validar_concat, 'o', label='Dados Reais')
plt.plot(previsao_lstm, label='Previsão LSTM')
plt.legend()
plt.show()
```



Fonte: O autor.

6.2.5 Comparação dos Resultados

Para compararmos todos resultados, foi criada uma função denominada *show_metricas* presente na Figura 42 que transforma os vetores das previsões em vetores de duas dimensões para serem desescalados pelo *inverse_transform* da biblioteca MinMaxScaler. Após isso, estando no seu valor original, é realizada duas previsões métricas, a EMA e a RQM utilizando os métodos *mean_squared_error* e *mean_absolute_error* da biblioteca Scikit-Learn.

Figura 42 – Obtenção das métricas dos modelos.

```
from sklearn.metrics import mean_squared_error, mean_absolute_error

def show_metricas(x, y):
    remq = mean_squared_error(escalador.inverse_transform(np.array(x).reshape(-1 ,1)),
                              escalador.inverse_transform(np.array(y).reshape(-1 ,1)),
                              squared=False)
    ema = mean_absolute_error(escalador.inverse_transform(np.array(x).reshape(-1 ,1)),
                              escalador.inverse_transform(np.array(y).reshape(-1 ,1)))
    print('REQM:', round(remq, 2))
    print(' EMA:', round( ema, 2))

print('Ingênua')
show_metricas(previsao_ingenua, y_validar_concat)
print('MME')
show_metricas(previsao_mme, y_validar_concat)
print('Reg. Lin.')
show_metricas(previsao_reg_lin, y_validar_concat)
print('LSTM')
show_metricas(previsao_lstm, y_validar_concat)
```

```
Ingênua
REQM: 5081008.5
 EMA: 3854053.62
MME
REQM: 4434373.5
 EMA: 3221124.79
Reg. Lin.
REQM: 4937368.37
 EMA: 3696213.44
LSTM
REQM: 4686508.74
 EMA: 3389028.76
```

Fonte: O autor.

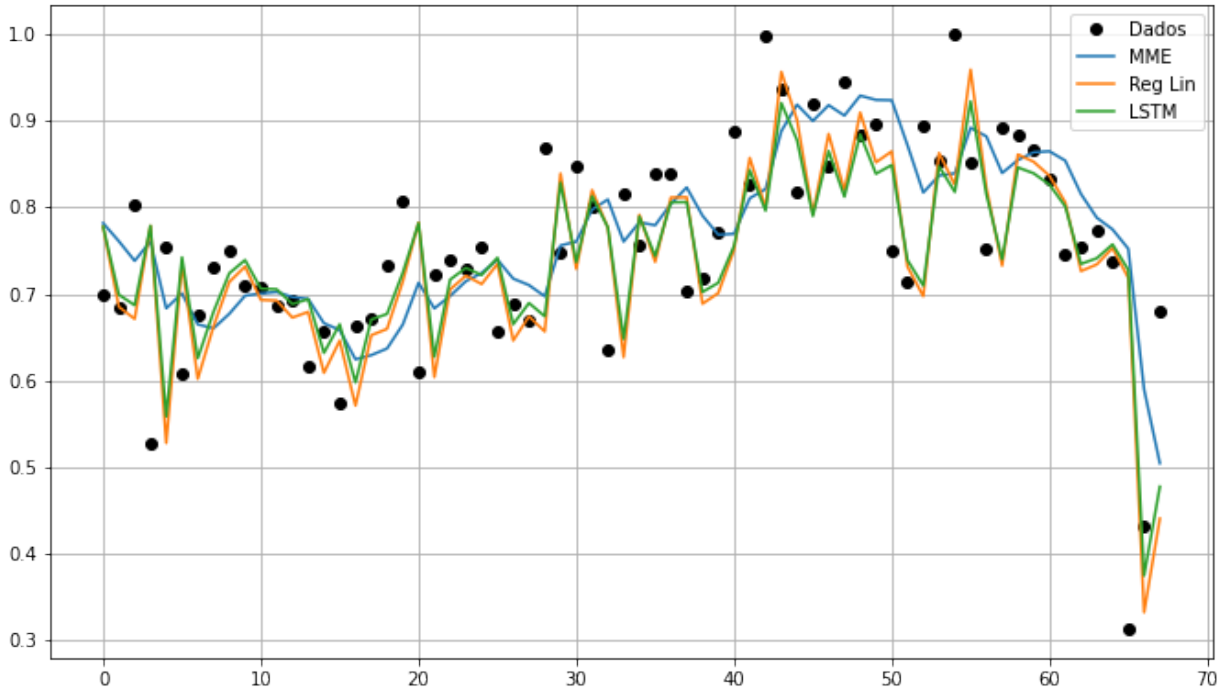
Essas métricas estão presentes na Tabela 4 para análise. Olhando para ela, percebemos que a Média Móvel Exponencial se saiu melhor, seguida da LSTM, da Regressão Linear e Previsão Ingênua, respectivamente. Todos métodos superaram o *benchamrk* e com uma boa margem. Trazendo para números reais, a diferença da métrica EMA entre a Previsão Ingênua e o menos performático dos métodos de aprendizagem de máquina neste caso, a Regressão Linear, temos uma diferença de 157.840. Como os valores neste conjunto está em dólares canadenses, vemos que um pequeno ganho em precisão pode significar uma grande mudança no planejamento de uma empresa.

Tabela 4 – Métricas das Previsões Univariadas Sobre o Conjunto das Indústrias Canadenses

Método de Previsão	EMA	REQM
Média Móvel Exponencial	3,2211	4,4343
LSTM Univ. 2 neur. TanH	3,3890	4,6865
Regressão Linear Univariada	3,6962	4,9373
Previsão Ingênua	3,8540	5,0810

Fonte: O autor.

Figura 43 – Previsões Univariadas Sobre o Conjunto de Dados das Indústrias Canadenses



Fonte: O autor.

As previsões dos 3 melhores métodos foram visualmente plotadas e estão presentes na Figura 43. Nela, podemos podemos observar que enquanto a LSTM e a Regressão Linear buscaram acompanhar os pontos, a Média Móvel Exponencial acompanhou mais como uma média destes, explicando talvez por isso sua melhor performance, sendo que quando as outras duas erravam, podiam errar por grande margem, enquanto que a MME se mantinha em uma média.

6.3 Previsões Multivariadas

Uma previsão multivariada significa que são passados como entrada ao modelo mais de uma variável, dando mais informações ao algoritmo. Para isto, assim como feito nos trabalhos de Carbonneau, Vahidov e Laframboise (2008) e Carbonneau, Vahidov e Laframboise (2007),

passamos como outras variáveis os meses anteriores ao qual queremos prever. Foram testadas entradas de 4, 6 e 12 meses anteriores e constatou-se que os melhores resultados obtidos foram com os vetores de entrada de 6 meses.

Podemos visualizar esse a forma como ficaram esses dados para entrada na Tabela 5. Fica claro observando esta tabela a necessidade de termos começado a considerar como um registro de entrada apenas a partir do mês 01/06/1992 pois apesar do conjunto iniciar em 01/01/1992, apenas nesta data tínhamos registros suficientes para formar o primeiro vetor de entrada completo ao modelo.

Tabela 5 – Deslizamento dos dados

Registro	Data	X_{t-6}	X_{t-5}	X_{t-4}	X_{t-3}	X_{t-2}	X_{t-1}	X_t
-	01/01/1992	-	-	-	-	-	-	20,5322
-	01/02/1992	-	-	-	-	-	20,5322	21,7925
-	01/03/1992	-	-	-	20,5322	21,7925	23,9485	23,4468
-	01/04/1992	-	-	20,5322	21,7925	23,9485	23,4468	23,9918
-	01/05/1992	-	20,5322	21,7925	23,9485	23,4468	23,9918	25,0477
0	01/06/1992	20,5322	21,7925	23,9485	23,4468	23,9918	25,0477	21,1522
1	01/07/1992	21,7925	23,9485	23,4468	23,9918	25,0477	21,1522	23,6199
2	01/08/1992	23,9485	23,4468	23,9918	25,0477	21,1522	23,6199	24,8329

Fonte: O autor.

Tabela 6 – Entrada Multivariada Escalada das Indústrias Canadenses

Registro	Data	X_{t-6}	X_{t-5}	X_{t-4}	X_{t-3}	X_{t-2}	X_{t-1}	y_t
0	01/06/1992	0,0000	0,0288	0,0783	0,0668	0,0793	0,1035	0,0142
1	01/07/1992	0,0288	0,0783	0,0668	0,0793	0,1035	0,0142	0,0707
...	...	...	...	...	...	...	...	...
265	01/07/2014	0,8736	0,7155	0,6848	0,7537	0,7569	0,7176	0,7012
266	01/08/2014	0,7155	0,6848	0,7537	0,7569	0,7176	0,7012	0,7994

Fonte: O autor.

Importante dizer que as previsões de Média Móvel Exponencial e Previsão Ingênua não sofreram alteração nos seus métodos, pois não tem a capacidade de previsões multivariadas. No entanto, suas previsões foram refeitas para esta seção, pois como o conjunto de dados ficou reduzido devido ao "deslizamento" dos dados, o valor correspondente aos 20% do treino ficou menor e com um registro inicial diferente.

Refazendo suas previsões no novo conjunto de validação, garantimos que todos métodos estão se apoiando sobre os mesmos dados e tivemos uma comparação justa entre os métodos. A previsão da Regressão Linear também ocorreu da mesma forma que na Seção 6.2, não precisando ter seus métodos registrados novamente, porém, agora se caracterizando como uma Regressão Linear Multivariada.

6.3.1 Previsão com LSTM

Quanto à LSTM, foi realizada novamente uma bateria de testes para verificarmos quais os melhores hiperparâmetros com esse novo modelo. Como diferença ao modelo univariado, a LSTM agora retornou melhores resultados com o learning rate $0{,}001$ padrão, explicado pelo fato de agora existirem mais dados e conexões a serem feitas, se beneficiando de uma aprendizagem mais lenta.

Tabela 7 – Métricas das Previsões Univariadas Sobre o Conjunto das Indústrias Canadenses

Neurônios	Ativação	Inicialização	Dropout	REQM
16	SELU	LeCun Normal	Não	4.3267
64	SELU	LeCun Normal	Não	4.3867
32	SELU	LeCun Normal	Não	4.4244
8	SELU	LeCun Normal	Não	4.6470
8	TanH	Glorot	Não	4.4968
16+8	SELU	LeCun Normal	Não	4.5575
16	TanH	Glorot	Não	4.5643
64+16	SELU	LeCun Normal	Não	4.5753
16+8	TanH	Glorot	Não	5.0074
16+8	SELU	LeCun Normal	0,1	6.0253
64+16	SELU	LeCun Normal	0,1	7.6977

Fonte: O autor.

Além deste, podemos observar os resultados com diversas alterações nos neurônios, função de ativação, inicialização e dropout na Tabela 7. Analisando estes resultados, percebemos que a regra geral para definição de neurônios continuou preponderante, sendo o modelo com melhor performance o de 16 neurônios.

Além dos neurônios, percebemos uma alteração pela preferência de função de ativação e inicialização, tendo melhor performance a SELU e LeCun Normal, respectivamente. Lembrando que, novamente, para estes testes foi definida a semente de aleatoriedade das bibliotecas Numpy e Tensorflow como 1234 e que como valor de comparação na Tabela 8 foi pego o melhor resultado após 10 seções de treinamento. A construção desta rede com essas alterações pode ser observada na Figura 44.

Figura 44 – Modelo LSTM Multivariado

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from tensorflow.keras import regularizers, callbacks, optimizers

# Cria o modelo
modelo = Sequential()
modelo.add(LSTM(units=16,
                input_shape=(n_aspectos, 1),
                activation="selu",
                kernel_initializer="lecun_normal",
                kernel_regularizer=regularizers.l2(0.01)))
modelo.add(Dense(1))
#Early Stopping
callback = callbacks.EarlyStopping(monitor='loss', patience=5)
# Nadam configs
nadam = optimizers.Nadam(learning_rate=0.001)
# Compila o modelo
modelo.compile(optimizer=nadam,
               loss="mean_squared_error")
```

Fonte: O autor.

6.3.2 Comparação dos Resultados

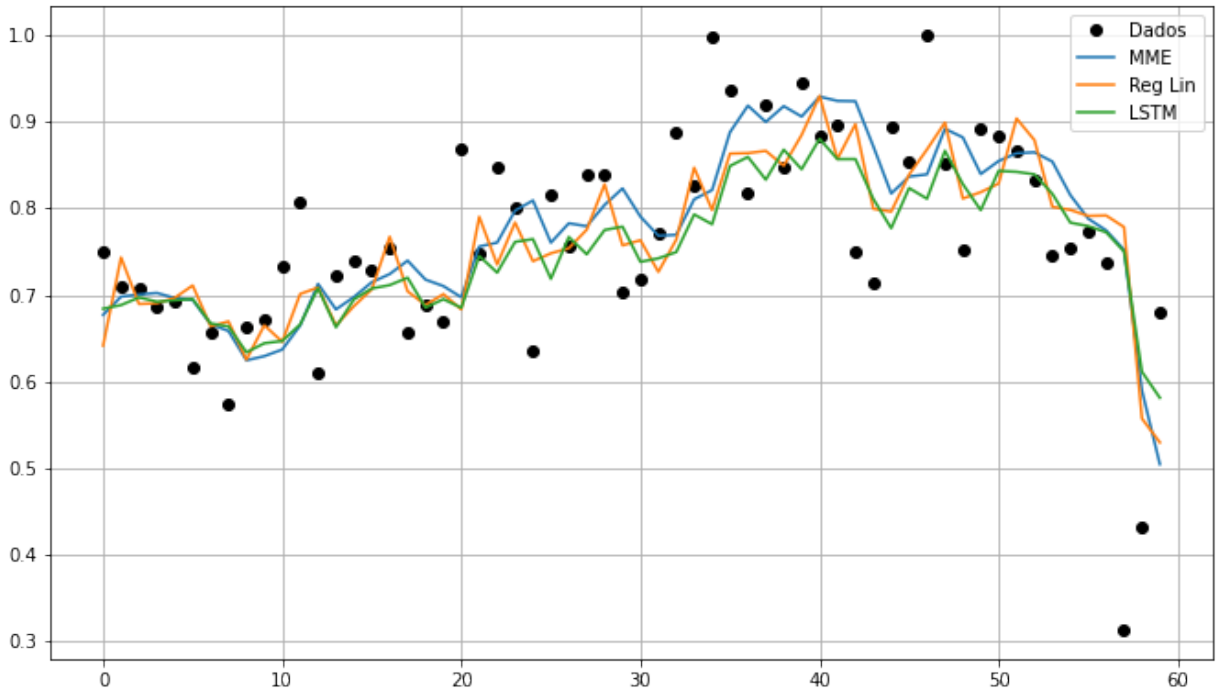
Analisando os resultados de todas previsões na Tabela 8, podemos notar que com mais informações por agora terem entradas multivariadas, os métodos de I.A. se saíram melhor do que antes, mudando inclusive de colocação quanto a performance. A Regressão Linear Multivariada obteve a previsão com maior acurácia, seguida do modelo LSTM, da Média Móvel Exponencial e da Previsão Ingênua. Novamente, trazendo para valores reais, percebemos que a diferença entre o melhor dos métodos anteriores, a MME, e o menos performático dos métodos de Aprendizagem de Máquina, a LSTM, temos uma diferença de C$86.777 e contra a Regressão Linear, de C$197.701.

Tabela 8 – Métricas das Previsões Multivariada Sobre o Conjunto das Indústrias Canadenses

Método de Previsão	EMA	REQM
Regressão Linear Multivariada	2,9442	4,1980
LSTM Multiv. 16 neur. SELU	3.0551	4.3187
Média Móvel Exponencial	3,1419	4,4096
Previsão Ingênua	3,6379	4,8524

Fonte: O autor.

Figura 45 – Previsões Multivariadas Sobre o Conjunto de Dados das Indústrias Canadenses



Fonte: O autor.

Plotando estas previsões e analisando o gráfico presente na Figura 45, podemos notar que no modelo multivariado, as linhas de previsão da LSTM e da Regressão Linear Multivariada acompanharam muito melhor as tendências dos dados reais quando comparadas às da Figura 43. Isso demonstra um significativo ganho de estabilidade ao receberem entradas com mais informações. Também fica claro, que apesar de acompanharem os dados, nos pontos de subidas e descidas elas conseguiram acompanhar mais os pontos reais do que a MME, mostrando uma agilidade à mudanças.

7 CONCLUSÃO

Neste trabalho, foi realizado um estudo teórico e prático sobre os métodos utilizados hoje na previsão de demanda da cadeia logística, buscando comparar os principais dentre estes com métodos do campo de Aprendizagem de Máquina. A previsão de demanda é muito importante por impactar o planejamento de uma empresa, estando diretamente relacionado com o sucesso de um projeto de médio a longo prazo. Qualquer ganho nesta previsão, é motivo de atenção e pesquisa principalmente quando falamos de grandes valores.

Aplicamos sobre um conjunto de dados público já utilizado por outros trabalhos acadêmicos, os métodos de: Média Móvel Exponencial, Regressão Linear Multivariada, *Long Short-Term Memory* e um método de *benchmark* em que apenas foram replicados os dados do mês anterior. Para entender estes métodos e como funcionam, foram pesquisados referenciais teóricos em Inteligência Artificial de diversos autores, e feito um apanhado dos pontos principais desta área, aprofundando-se nos assuntos que poderiam ser relevantes para a pergunta chave deste trabalho, que questiona se IA gera bons resultados na previsão de demanda.

Através da busca e execução destes conhecimentos, este trabalho também me trouxe grande desenvolvimento pessoal, me colocando dentro de uma nova área interessante e empolgante que traz muita inovação e desafios. Foi muito interessante conhecer todo o ecossistema em volta de Aprendizagem de Máquina, que é muito mais bem desenvolvido do que eu imaginava, tendo muitas ferramentas, documentação e comunidades ativas.

Antes de concluirmos com a análise das previsões, é importante comentarmos sobre questões mais técnicas a respeito do uso da Rede Neural Recorrente LSTM. Hoje, a parte de seleção de parâmetros ainda é praticamente mais arte do que Ciência, sendo prudente compartilharmos algumas percepções na tentativa de trazermos mais informações para este campo.

Concluiu-se que é possível utilizar como um bom ponto de partida para seleção da quantidade de neurônios em uma rede, o expoente dois mais próximo do dobro da quantidade de características no vetor de entrada. Mesmo realizando testes com várias outras alternativas, essa quantidade sempre foi a melhor encontrada. Além disso, ficou claro o benefício que algoritmos de Aprendizagem de Máquina obtém com vetores de entrada maiores. Outro ponto, que esse trabalho demonstra é que para problemas simples, é melhor o uso de redes simples, piorando os resultados ao se tentar adicionar complexidade.

Quanto à análise dos resultados, observou-se que quando fornecidos dados multivariados, com 6 meses anteriores para cada previsão, foi possível perceber um ganho significativo no uso dos métodos de Aprendizagem de Máquina. Como este conjunto está na casa das dezenas de milhões, esse ganho teria representado uma real mudança de planejamento na prática, já que a diferença de acuracidade entre um método e outro chegou a ser de mais de uma centena de milhar de dólares canadenses. No entanto, não tivemos ganho ao utilizarmos os métodos de Aprendizagem de Máquina com entradas simples e univariadas. Em ambos casos, a LSTM não

ficou na primeira colocação.

Isso nos leva a concluir que o campo de Aprendizagem de Máquina pode sim trazer resultados melhores que outros métodos, principalmente a Regressão Linear Multivariada. Quanto à LSTM, embora ela possa ser utilizada com bons ganhos para esse objetivo, não deixando em nada a desejar em comparação aos outros, por ter uma complexidade muito maior do que a Regressão Linear e não ter demonstrado melhor acurácia, para estes problemas ela não é recomendada.

7.1 TRABALHOS FUTUROS

Buscando aumentar o conhecimento da academia nessa área, trazendo mais testes com hiperparâmetros e conjuntos de dados, uma sugestão de trabalho futuro é realizar as mesmas previsões deste em outros conjuntos de dados, e com alterações nos parâmetros da Rede Neural.

Outro ponto interessante e que pode trazer melhores resultados, é adicionar aos vetores de entrada dados não relacionados diretamente ao conjunto, mas que podem exercer uma influência indireta. Alguns exemplos podem ser a quantidade geral de vendas da empresa, demonstrando o cenário atual desta. Ou então, representar o cenário econômico atual pela média da bolsa de valores. Seria possível utilizar qualquer indicador que possa ter influência no valor a ser previsto.

REFERÊNCIAS

ABADI, M. et al. **TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems**. 2015. Software available from tensorflow.org. Disponível em: <https://www.tensorflow.org/>. Citado na página 36.

AL-MASRI, A. **How Does Back-Propagation in Artificial Neural Networks Work?** 2019. Disponível em: <https://towardsdatascience.com/how-does-back-propagation-in-artificial-neural-networks-work-c7cad873ea7>. Acesso em: 18 de Novembro de 2020. Citado na página 12.

ALBON, C. **Python Machine Learning Cookbook**: Practical solutions from preprocessing to deep learning. 1. ed. Sebastopol, California: O'Reilly Media Inc, 2018. Citado 3 vezes nas páginas 8, 29 e 32.

BALLOU, R. H. **Gerenciamento da Cadeia de Suprimentos/Logística Empresarial**. 5. ed. Porto Alegre: Bookman, 2006. Citado 3 vezes nas páginas 2, 3 e 4.

BENGIO, Y.; SIMARD, P.; FRASCONI, P. Learning long-term dependencies with gradient descent is difficult. **IEEE TRANSACTIONS ON NEURAL NETWORKS**, v. 5, n. 2, March 1994. Citado na página 13.

BOUSQAOUI, H.; ACHCHAB, S.; TIKITO, K. Machine learning applications in supply chains: Long short-term memory for demand forecasting. **[Lecture Notes in Networks and Systems] Cloud Computing and Big Data: Technologies, Applications and Security**, v. 49, 2019. Citado 2 vezes nas páginas 2 e 8.

BURKOV, A. **The Hundred-Page Machine Learning Book**. 1. ed. [S.l.]: Andriy Burkov, 2019. Citado 10 vezes nas páginas 5, 11, 12, 14, 15, 18, 20, 25, 27 e 28.

CARBONNEAU, R.; VAHIDOV, R.; LAFRAMBOISE, K. Machine learning-based demand forecasting in supply chains. **International Journal of Intelligent Information Technologies**, v. 3, 2007. Citado 2 vezes nas páginas 2 e 54.

CARBONNEAU, R.; VAHIDOV, R.; LAFRAMBOISE, K. Application of machine learning techniques for supply chain demand forecasting. **European Journal of Operational Research**, v. 184, 2008. Citado 3 vezes nas páginas 4, 8 e 54.

CHOLLET, F. et al. **Keras**. 2015. <https://keras.io>. Citado na página 36.

CLEVERT, D.-A.; UNTERTHINER, T.; HOCHREITER, S. Fast and accurate deep network learning by exponential linear units (elus). **International Conference on Learning Representations**, 2016. Citado na página 23.

DATA SCIENCE ACADEMY. **Deep Learning Book**. 2019. Disponível em: <http://www.deeplearningbook.com.br>. Acesso em: 06 de Maio de 2020. Citado 3 vezes nas páginas 12, 13 e 15.

FACURE, M. **Funções de Ativação**: Entendendo a importância da ativação correta nas redes neurais. 2017. Disponível em: <https://matheusfacure.github.io/2017/07/12/activ-func/>. Acesso em: 15 de Novembro de 2020. Citado 5 vezes nas páginas 20, 21, 22, 23 e 24.

GLOROT, X.; BENGIO, Y. Understanding the difficulty of training deep feedforward neural networks. **Proceedings of the thirteenth international conference on artificial intelligence and statistics**, 3 2010. Citado na página 20.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. [S.l.]: MIT Press, 2016. <http://www.deeplearningbook.org>. Citado 2 vezes nas páginas 9 e 12.

GéRON, A. **Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow**. 2. ed. [S.l.]: O'Reilly Media, Inc, 2019. Citado 11 vezes nas páginas 5, 6, 7, 8, 19, 20, 23, 24, 25, 26 e 37.

HANSEN, C. **Activation Functions Explained - GELU, SELU, ELU, ReLU and more**. 2019. Disponível em: <https://mlfromscratch.com/activation-functions-explained/#/>. Acesso em: 23 de Julho de 2020. Citado na página 23.

HARRIS, C. R. et al. Array programming with NumPy. **Nature**, Springer Science and Business Media LLC, v. 585, n. 7825, p. 357–362, set. 2020. Disponível em: <https://doi.org/10.1038/s41586-020-2649-2>. Citado na página 30.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. **Neural Computation**, v. 9, 1997. Citado na página 14.

HUNTER, J. D. Matplotlib: A 2d graphics environment. **Computing in Science & Engineering**, IEEE COMPUTER SOC, v. 9, n. 3, p. 90–95, 2007. Citado na página 34.

KLAMBAUER, G. et al. Self-normalizing neural networks. **Advances in Neural Information Processing Systems**, v. 30, 2017. Citado na página 23.

KLUYVER, T. et al. Jupyter notebooks - a publishing format for reproducible computational workflows. In: LOIZIDES, F.; SCMIDT, B. (Ed.). **Positioning and Power in Academic Publishing: Players, Agents and Agendas**. Netherlands: IOS Press, 2016. p. 87–90. Disponível em: <https://eprints.soton.ac.uk/403913/>. Citado na página 35.

MCKINNEY Wes. Data Structures for Statistical Computing in Python. In: WALT Stéfan van der; MILLMAN Jarrod (Ed.). **Proceedings of the 9th Python in Science Conference**. [S.l.: s.n.], 2010. p. 56 – 61. Citado na página 32.

MOOLAYIL, J. **Learn Keras for Deep Neural Networks**. 1. ed. [S.l.]: Apress Media LLC, 2019. Citado 11 vezes nas páginas 19, 20, 21, 22, 24, 25, 26, 27, 29, 35 e 36.

MUELLER, J. P.; MASSARON, L. **Machine Learning For Dummies**. Hoboken, New Jersey: John Wiley and Sons Inc, 2006. Citado na página 4.

MüLLER, A. C.; GUIDO, S. **Introduction to Machine Learning with Python**: A guide for data scientists. 1. ed. 1005 Gravenstein Highway North, Sebastopol, CA: O'Reilly Media Inc, 2016. Citado 8 vezes nas páginas 10, 11, 28, 29, 30, 32, 33 e 34.

OLAH, C. **Understanding LSTM Networks**. 2015. Disponível em: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>. Acesso em: 15 de Novembro de 2020. Citado 6 vezes nas páginas 13, 14, 15, 16, 17 e 18.

PEDREGOSA, F. et al. Scikit-learn: Machine learning in Python. **Journal of Machine Learning Research**, v. 12, p. 2825–2830, 2011. Citado na página 33.

PHI, M. **Illustrated Guide to Recurrent Neural Networks**. 2018. Disponível em: <https://towardsdatascience.com/illustrated-guide-to-recurrent-neural-networks-79e5eb8049c9>. Acesso em: 15 de Novembro de 2020. Citado na página 13.

RUSSEL, S.; NORVIG, P. **Inteligência Artificial**. 3. ed. [S.l.]: Elsevier Editora Ltda., 2013. Citado 5 vezes nas páginas 5, 6, 7, 8 e 9.

SAMUEL, A. L. Some studies in machine learning using the game of checkers. **IBM J. Res. Dev.**, v. 3, p. 210–229, 1959. Citado na página 5.

SCHMIDT, D. **Understanding Nesterov Momentum (NAG)**. 2018. Disponível em: <https://dominikschmidt.xyz/nesterov-momentum/>. Acesso em: 23 de Julho de 2020. Citado na página 25.

SHAHRABI, J.; MOUSAVI, S. S.; HEYDAR, M. Supply chain demand forecasting: A comparison of machine learning. **Journal of Applied Sciences**, v. 9, 2009. Citado na página 8.

STATISTICS CANADA. **Manufacturers' sales, inventories, orders and inventory to sales ratios, by industry**. 2020. Disponível em: <https://www150.statcan.gc.ca/t1/tbl1/en/tv.action?pid=1610004701>. Acesso em: 23 de Julho de 2020. Citado na página 38.

TENSORFLOW. **TensorFlow Tutorial - Time Series Forecasting**. 2020. Disponível em: <https://www.tensorflow.org/tutorials/structured_data/time_series>. Acesso em: 06 de Maio de 2020. Citado na página 14.

Time de Desenvolvimento pandas. **pandas-dev/pandas: Pandas**. Zenodo, 2020. Disponível em: <https://doi.org/10.5281/zenodo.3509134>. Citado na página 32.