Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Matemática Aplicada

Redes Neurais aplicadas às Séries Temporais

Eduardo Galvani Massino

Monografia Final

map2010 — Trabalho de Formatura

Orientador: José Coelho de Pina Junior
Coorientador: Alberto Ueda

São Paulo
Janeiro de 2021

Dedico esse trabalho à minha família, minha mãe e irmã, e em especial à minha futura esposa Camila, que me ajudou em muitos momentos difíceis neste ano, sobretudo durante a escrita desse texto. Por fim dedico esse trabalho a todos os andróides e robôs da ficção científica que sempre me inspiraram e me puseram no mundo da ciência e da computação desde criança.

Lieutenant Commander Data is a machine. Do we deny that? No, because it is not relevant: we, too, are machines, just machines of a different type.
— Captain Picard

Is Data a machine? Yes. Is he the property of Starfleet? No. We've all been dancing around the basic issue: does Data have a soul? I don't know that he has. I don't know that I have! But I have got to give him the freedom to explore that question himself. It is the ruling of this court that Lieutenant Commander Data has the freedom to choose.
— Captain Louvois em "The Measure Of A Man",
episódio da série *Star Trek The Next Generation*

Resumo

Eduardo Galvani Massino. **Redes Neurais aplicadas às Séries Temporais**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2021.

Elemento obrigatório, constituído de uma sequência de frases concisas e objetivas, em forma de texto. Deve apresentar os objetivos, métodos empregados, resultados e conclusões. O resumo deve ser redigido em parágrafo único, conter no máximo 500 palavras e ser seguido dos termos representativos do conteúdo do trabalho (palavras-chave). Deve ser precedido da referência do documento. Texto texto. Texto texto.

Palavras-chave: machine-learning. redes-neurais. perceptron. séries-temporais.

Abstract

Eduardo Galvani Massino. **Redes Neurais aplicadas às Séries Temporais**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2021.

Elemento obrigatório, elaborado com as mesmas características do resumo em língua portuguesa. De acordo com o Regimento da Pós-Graduação da USP (Artigo 99), deve ser redigido em inglês para fins de divulgação. É uma boa ideia usar o sítio www.grammarly.com na preparação de textos em inglês. Text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text. Text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text.

Keywords: machine-learning neural-nets. perceptron. temporal-series.

Lista de Figuras

1.1 Diagrama de Venn relacionando os três aspectos inerentes à ciência de dados. 2

2.1 Tarefa de classificação. 6
2.2 Tarefa de regressão. 7
2.3 Tarefa de clusterização. 7
2.4 Sobreajuste num modelo de classificação. 9
2.5 Modelo de regressão, ajuste linear. 9
2.6 Modelo de regressão, polinômio quadrático. 10
2.7 Modelo de regressão, polinômio de grau *n*. 10
2.8 Árvore de decisão classificando famílias de plantas. 14
2.9 Representação de um neurônio biológico. 18
2.10 Rede neural simples, o perceptron de camada única. 19
2.11 Rede neural mais simples ainda, apenas um neurônio oculto. 19
2.12 Representação de um neurônio artificial. 20
2.13 As redes neurais recorrentes: *perceptron*, *feedforward* e *deep feedforward*. 21
2.14 As redes neurais recorrentes. 22
2.15 As redes neurais convolucionais. 22
2.16 Redes neurais extremas. 23

3.1 Visão estrutural da rede perceptron. A linha tracejada destaca uma das camadas da rede. 29
3.2 Comparação entre as funções de ativação do tipo escada e a *sigmoid*. . . 31
3.3 Gráficos da função *sigmoid* e sua derivada. 32
3.4 Gráficos da função tangente hiperbólico e sua derivada. 33
3.5 Gráficos da função *RELU* e sua derivada. 34
3.6 Gráficos da função *Leaky RELU* e sua derivada. 35
3.7 Gráficos da função *ELU* e sua derivada. 36
3.8 Exemplos de fotos da base de dados MNIST de números manuscritos. . . 48

3.9 Matriz de confusão e acurácia do conjunto de treino da base MNIST 8 × 8 pixels. 50
3.10 Matriz de confusão e acurácia do conjunto de teste da base MNIST 8 × 8 pixels. 51
3.11 As 2 implementações da API Keras. *Multibackend* à esquerda e *TensorFlow* à direita. 52
3.12 Matriz de confusão, função de perda e acurácia do conjunto de teste da base MNIST 8 × 8 pixels, utilizando a API Keras. 54
3.13 Matriz de confusão, função de perda e acurácia do conjunto de teste da base MNIST 28 × 28 pixels, utilizando a API Keras. 55

4.1 Processo estocástico como uma família de trajetórias, isto é, de séries temporais. 59
4.2 Esquerda: Índices mensais do Ibovespa. Direita: Log-diferença do Ibovespa. 61
4.3 Função de autocorrelação (fac) de um ruído branco. 63
4.4 Autocorrelações e autocorrelações parciais amostrais de alguns modelos *ARIMA*. 73

A.1 Visualização do método do gradiente descendente com taxa de aprendizado única. Os pontos azuis representam candidatos a ponto mínimo em cada iteração do algoritmo. 78
A.2 Visualização do método do gradiente descendente com taxa de aprendizado única. Ilustração do uso de um valor de taxa de aprendizado muito grande. 79
A.3 Visualização do método do gradiente descendente com taxa de aprendizado variável que vai diminuindo passo-a-passo do algoritmo. 80
A.4 Comportamento de diferentes taxas de aprendizado nos valores candidatos a mínimo. 80

B.1 Gráficos da amplitude por média de subconjuntos de Z_t, com os correspondentes valores de λ. 82

Lista de Programas

3.1 Trecho da classe Neuron 29
3.2 Trecho do script util.py 31
3.3 Trecho da classe *Layer* 37
3.4 Trecho da classe *Layer* 38
3.5 Trecho da classe *Layer* 38
3.6 Trecho da classe *Network* 39
3.7 Trecho da classe *Network* 39
3.8 Trecho da classe *Network* 39
3.9 Trecho da classe *Network* 40
3.10 Trecho da classe *Network* 41
3.11 Trecho da classe *Network* 41
3.12 Trecho da classe *Network* 41
3.13 Trecho da classe *Perceptron* 43
3.14 Trecho da classe *Perceptron* 44
3.15 Trecho da classe *Perceptron* 45
3.16 Trecho da classe *Perceptron* 46
3.17 Trecho da classe *Perceptron* 46
3.18 Trecho da classe *Perceptron* 46
3.19 Trecho da classe *Perceptron* 47
3.20 Trecho da classe *Perceptron* 47
3.21 Trecho do script *mnist_test.py* 49
3.22 Trecho do script *mnist_test.py* 49
3.23 Trecho do script *mnist_keras.py* 52
3.24 Trecho do script *mnist_keras.py* 52
3.25 Trecho do script *mnist_keras.py* 53
3.26 Trecho do script *mnist_keras.py* 54

Sumário

1 Introdução **1**

2 Redes neurais no contexto de aprendizado de máquina **5**
2.1 Tipos de aprendizagem . . . 5
2.1.1 O problema do sobreajuste dos modelos . . . 8
2.2 Alguns algoritmos de aprendizagem . . . 11
2.2.1 Regressão Linear . . . 11
2.2.2 Regressão Logística . . . 12
2.2.3 Árvores de decisão . . . 14
2.2.4 *K*-médias . . . 16
2.3 As redes neurais e o *perceptron* . . . 17
2.4 Outras arquiteturas de redes neurais . . . 21

3 Perceptron multi-camadas **25**
3.1 Matemática do algoritmo de retropropagação . . . 25
3.2 Implementação do algoritmo de retropropagação . . . 28
3.2.1 O neurônio . . . 29
3.2.2 A função de ativação . . . 30
3.2.3 As camadas . . . 36
3.2.4 A rede . . . 38
3.2.5 A classe *Perceptron* . . . 43
3.3 Exemplo de utilização do *perceptron* . . . 48
3.4 Utilizando a API Keras . . . 51

4 Séries temporais **57**
4.1 Processos estocásticos . . . 58
4.1.1 Definições . . . 59
4.1.2 Processos estacionários . . . 60
4.1.3 Função de autocorrelação . . . 62

4.1.4 Exemplos de processos estocásticos 62
4.2 Os modelos ARIMA 64
4.2.1 Processo linear geral 66
4.2.2 Modelos autorregressivos (AR) 67
4.2.3 Modelos de médias móveis (MA) 68
4.2.4 Modelos autorregressivos e de médias móveis (ARMA) 69
4.2.5 Os modelos integrados não-estacionários (ARIMA) 69
4.2.6 Identificação dos modelos utilizando a função de autocorrelação . 70

5 Procedimentos de comparação e resultados **75**
5.1 Conhecendo a série temporal de interesse 75
5.2 Estimando um modelo ARIMA 75
5.3 Construção do modelo com uma rede neural 75

Apêndices

A O gradiente descendente **77**

B Transformação de Box-Cox **81**

C Função de autocorrelação parcial **83**

Anexos

Referências **85**

Capítulo 1

Introdução

De tempos pra cá, ler e ouvir falar de ciência de dados tornou-se muito comum, tanto nos meios profissionais e científicos quanto na mídia. Existem atualmente aplicações em praticamente todas as áreas do conhecimento humano, da agricultura à indústria e ao entretenimento.

Joel Grus (Grus, 2016) define genericamente **ciência de dados** como sendo a extração de conhecimento a partir de dados desorganizados. Os dados podem ser números, textos, áudio, vídeo, entre outros quaisquer que podem ser úteis na tomada de decisões de negócios.

Pedro A. Morettin e Julio M. Singer (Morettin e Motta Singer, 2020) afirmam que este termo, embora usado como se fosse um conceito novo, não está separado dos conceitos já históricos da *estatística*. Eles apontam que o trabalho dos *cientistas de dados* difere do trabalho dos *estatísticos* apenas quando eles usam dados multimídia como áudio, vídeo, ou textos. Mas que, uma vez que esses dados são processados e tornam-se números, as técnicas e conceitos utilizados por ambos passam a ser basicamente os mesmos.

Na verdade, Morettin e Singer (Morettin e Motta Singer, 2020) citam que na década de 80 houve uma primeira tentativa de aplicar o rótulo *ciência de dados*, (*Data Science*), ao trabalho feito pelos estatísticos aplicados da época, como uma forma de dar-lhes mais visibilidade. Curiosamente, fato mencionado pelos autores, existem atualmente cursos específicos de ciência de dados em universidades ao redor do mundo, mas a maioria deles situada em institutos de áreas aplicadas como engenharia e economia, e raramente nos institutos de estatística propriamente ditos.

Para entender um pouco mais de seu escopo, David M. Blei e Padhraic Smyth (Blei e Smyth, 2017) discutem ciência de dados sob as visões estatística, computacional e humana. Eles argumentam que é a combinação desses três componentes que formam a essência do que ela é e, assim como, do conhecimento que ela é capaz de produzir.

Pode-se extender e observar essa ideia de uma visão em conjunto dos aspectos estatísticos, computacionais e humanos com um diagrama de Venn. Por exemplo, seja o diagrama criado por Andrew Silver (Silver, 2018) e mostrado a seguir na figura 1.1.

Dessa forma vemos que no aspecto estatístico, seja univariado ou multivariado, está a

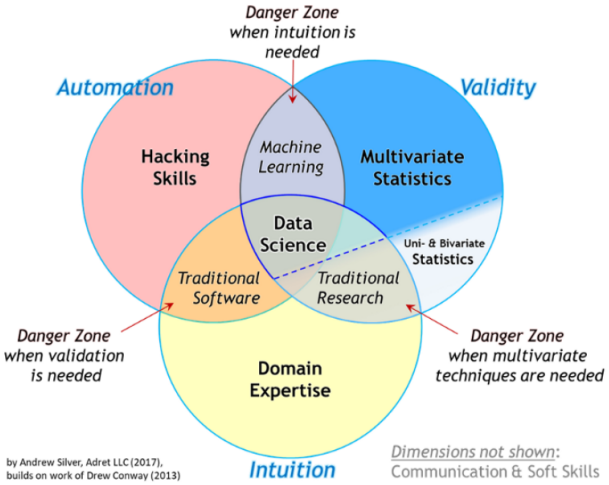

Figura 1.1: *Diagrama de Venn relacionando os três aspectos inerentes à ciência de dados.*[a]

[a]Extraído de (Silver, 2018)

criação e validação dos modelos utilizados. No aspecto computacional estão as habilidades computacionais, de criação e de eficiência dos modelos. E que no aspecto humano, está a intuição e o domínio do assunto que está sendo estudado.

Além disso, podemos ver, nas intersecções entre os aspectos, as definições de outros conceitos-chave. O **aprendizado de máquina** está na intersecção entre a estatística e a computação, já que o objetivo é criar modelos *computacionais* a partir de modelos estatísticos.

Uma pesquisa tradicional utiliza o conhecimento de um profissional da área aliado com habilidades e ferramentas estatísticas. Um software tradicional, por sua vez, é criado por profissionais de tecnologia para o uso de profissionais de outras áreas.

A ciência de dados, portanto, é a junção desses aspectos, aliando modelagem estatística, automação computacional, validade, e intuição. As áreas perigosas mostradas na figura 1.1, refletem o que pode ocorrer quando algum dos três aspectos é negligenciado numa empreitada de ciência de dados.

Algoritmos de *aprendizado de máquina* vem sendo utilizados em grande parte dos modelos de ciência de dados. Mas o que é aprendizado de máquina? Ou então, o que significa dizer que o computador, neste caso a "máquina", está *aprendendo*?

Aurélien Géron (Géron, 2019) nos dá uma ideia geral lembrando que uma das primeiras aplicações de sucesso de aprendizado de máquina foi o filtro de *spam*, criado na década de 90. Uma das fases de seu desenvolvimento foi aquela em que os usuários assinalavam que certos e-mails eram *spams* e outros não eram. Hoje em dia, raramente temos que marcar ou desmarcar e-mails, pois a maioria dos filtros já "aprenderam" a fazer seu trabalho de forma muito eficiente, não temos mais nada a "ensiná-lo".

O conceito de aprendizado de máquina está intimamente ligado à ciência da computação. Porém, no contexto de ciência de dados, é definido por Grus (Grus, 2016) como a "criação e o uso de modelos que são ajustados a partir dos dados". Seu objetivo é usar dados

existentes para desenvolver modelos que possamos usar para *prever* possíveis respostas à consultas.

Exemplos, além do filtro de *spams* podem ser: detectar transações de crédito fraudulentas, calcular a chance de um cliente clicar em uma propaganda ou então prever qual time de futebol irá vencer o Campeonato Brasileiro.

Como ficará claro ao longo deste trabalho, o aprendizado consiste na utilização de dados já conhecidos para ajustar parâmetros de modelos. Uma vez ajustados os parâmetros, o algoritmo que descreve o modelo passa a ser usado para responder às consultas. Essa fase de ajuste de parâmetros é chamada de aprendizado ou treinamento.

Uma **rede neural** é um exemplo de modelo preditivo de aprendizado de máquina que foi criado com inspiração no funcionamento do cérebro biológico. David Kopec (Kopec, 2019) descreve que apesar de terem sido as primeiras a serem criadas, elas vem ganhando nova importância na última década, graças ao avanço computacional, uma vez que exigem muito processamento, e também porque podem ser usadas para resolver problemas de aprendizagem dos mais variados tipos.

Atualmente existem vários tipos de redes neurais, porém este trabalho lida principalmente com aquele tipo que foi originalmente criado sob a inspiração do funcionamento do cérebro, chamado de **perceptron**, e que portanto tenta imitar o comportamento dos neurônios e suas conexões, aprendendo padrões a partir de dados existentes e tentando prever o comportamento de dados novos a partir do padrão aprendido.

Uma visão geral da arquitetura do aprendizado de máquina, situando a posição das redes neurais e do algoritmo *perceptron* em toda esta estrutura, assim como exemplos de aplicações em cada uma das suas ramificações, estão no Capítulo 2.

Neste trabalho é utilizada como base didática uma versão simples do algoritmo *perceptron* feita e explicada por Kopec (Kopec, 2019), e a partir desta base, foram criados novos métodos de treinamento e de validação com algumas estratégias, como uma tentativa de automatizar e otimizar o processo de treinamento do algoritmo, que é comumente feito de forma heurística. Detalhes dessa implementação estão no Capítulo 3.

São apresentados os conceitos e usos das séries temporais de dados não-lineares no Capítulo 4, assim como alguns exemplos de aplicações na área financeira. As técnicas tradicionais de análise e de previsão de séries temporais de dados serão brevemente apresentadas neste capítulo.

Para servir de validação e aplicação do algoritmo criado, no Capítulo 5 serão feitas comparações de desempenho entre o *perceptron* e os modelos tradicionais de previsões de séries temporais apresentados no capítulo anterior, como o **SARIMA** (*seasonal auto regressive integrated moving average*). Tais comparações usam como inspiração trabalhos similares como o de Alsmadi, Omar, Noah e Almarashdah (Alsmadi *et al.*, 2009).

Ao final concluo sobre os modelos de previsões aqui estudados e comparados, destacando a qualidade e eficiência dos métodos de redes neurais, tão bons e às vezes melhores do que os métodos tradicionais estatísticos.

Capítulo 2

Redes neurais no contexto de aprendizado de máquina

Neste capítulo são apresentados alguns conceitos básicos de ciência de dados e de aprendizado de máquina, alguns dentre os vários tipos e exemplos de algoritmos de aprendizagem, direcionando-os para aquele que é o foco do trabalho, ou seja, as redes neurais artificiais.

Neste texto os termos "algoritmo" e "técnica" serão usados livremente como sinônimos, pois uma técnica de aprendizado de máquina, no contexto atual, é um algoritmo executado no computador que tem por objetivo ajustar parâmetros de modelos estatísticos.

Analogamente à definição dada na Introdução (1), citamos outras duas. Prince Barpaga (Barpaga, 2019) define aprendizado de máquina como sendo um ramo da inteligência artificial, em que computadores são treinados a partir de dados conhecidos para realizar alguma tarefa específica, ao invés de ser explicitamente programado para exibir uma resposta fixa.

Similarmente, Robbie Allen (Allen, 2020) descreve que um conjunto de dados é usado para treinar um modelo estatístico de forma que, ao ser deparado com dados similares aos dados usados para o treino, saberá como tratá-los. Geralmente, dados são usados como entradas para esses modelos que irão fornecer como saída predições de interesse.

Concluindo, pode-se utilizar o grau de supervisão humana durante o aprendizado para classificá-los em diferentes tipos, como é descrito por Géron (Géron, 2019). Durante o aprendizado podem ser fornecidos um conjunto de consultas e de respostas já conhecidas. Tais respostas foram dadas por humanos, ao menos neste momento, e daí o termo "supervisão humana".

2.1 Tipos de aprendizagem

Um algoritmo de **aprendizado supervisionado** é usado quando conhecemos características dos dados que estamos utilizando. De modo geral já temos de antemão as respostas às consultas para os dados utilizados no treinamento. Por exemplo, se estamos classificando

fotos de animais, possuímos um conjunto de fotos para as quais já sabemos quais são de gatos, cachorros, etc.

O ato de rotular previamente os dados que usamos no treinamento é o que designamos de supervisão humana. Uma vez *treinado*, o algoritmo recebe uma foto, ou seja, uma nova consulta e então fornece a resposta, neste caso se essa é a foto de um gato, ou cachorro, ou qualquer outra resposta dentre aquelas que foram dadas como exemplos durante o treinamento.

Dentro do aprendizado supervisionado temos duas técnicas principais. A primeira é a **classificação**, usada para rotular ou dividir os dados em classes pré-determinadas, a partir de exemplos, que é exatamente o caso dos exemplos descritos nos parágrafos anteriores, e ilustrada na figura 2.1.

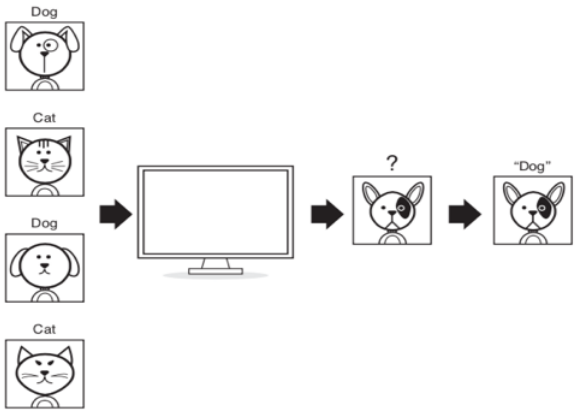

Figura 2.1: *Tarefa de classificação.*[a]

[a]Extraído de (Allen, 2020)

A segunda técnica é a **regressão**, usada para prever valores, ou seja, fornecer respostas a consultas ainda inéditas, sejam dados do futuro ou valores de funções em pontos do domínio para os quais ainda não existem valores. Podemos entender a diferença, com a ajuda de Allen (Allen, 2020) se percebermos que na classificação as respostas são valores discretos, isto é, um 'cachorro' ou um 'gato'.

Enquanto isso, na regressão as respostas são valores contínuos, um intervalo real de possibilidades. Como exemplo, Allen (Allen, 2020) ilustra na figura 2.2, dado uma imagem radiológica, um modelo poderia prever em quantos anos uma pessoa poderia desenvolver alguma doença.

No **aprendizado não-supervisionado** não sabemos os rótulos dos dados que estamos lidando, isto é, não sabemos previamente respostas aos dados conhecidos, assim o algoritmo poderá agrupar os dados de forma automática, por exemplo, se estivermos lidando com problemas de classificação. Aqui, as consultas podem ser coisas como "quantos são os perfis dos clientes" ou "quantas espécies de flores existem nestas fotos", e assim por diante.

Alguns métodos não-supervisionados de aprendizado foram enumeradas por Géron (Géron, 2019). O **agrupamento** de dados similares sob uma inspiração geométrica. Nesse caso os dados são agrupados conforme suas posições num determinado espaço e utiliza-

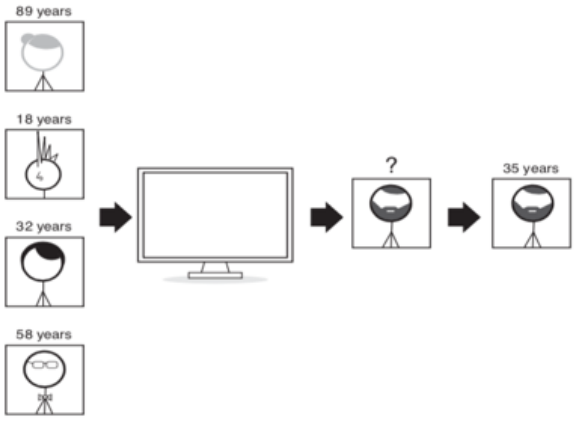

Figura 2.2: *Tarefa de regressão.*[a]

[a]Extraído de (Allen, 2020)

se algoritmos como k-vizinhos, k-means, k-medians, etc. Exemplos de aplicações são agrupamento de produtos em supermercados, interesses comuns de clientes em sites de conteúdo digital, etc.

Para ilustar, Allen (Allen, 2020) sugere que imaginemos um conjunto de artigos de texto que gostaríamos de organizar. Alguns poderiam ser sobre esportes, outros sobre história, outros sobre arte, etc. O objetivo seria identificar e classificar automaticamente os textos dentre alguns assuntos possíveis. Imaginando cada assunto provável como uma figura geométrica diferente, a tarefa seria realizada idealmente como na figura 2.3.[1]

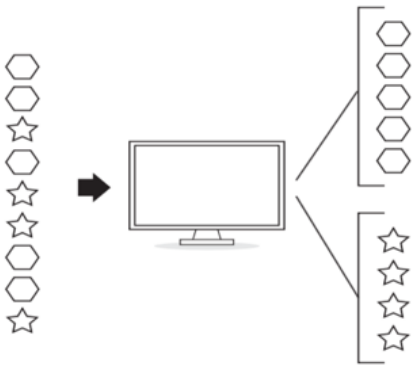

Figura 2.3: *Tarefa de clusterização.*[a]

[a]Extraído de (Allen, 2020)

Existe também o **aprendizado semi-supervisionado** em que combinam-se vantagens de ambos os tipos anteriores. Um modelo utiliza dados sem rótulos para descobrir uma estrutura geral dos dados, enquanto usa alguns poucos rótulos conhecidos para ajudar na organização inicial dessa estrutura, o que auxilia na partida do agrupamento, que é um problema das técnicas não-supervisionadas.

[1]Vale ressaltar que não sabemos *a priori* os rótulos aqui representados a favor do entendimento.

Essa técnica é também conhecida por *aprendizagem fraca*, e conforme aponta Allen (Allen, 2020) possui a vantagem de precisar de menor quantidade de dados rotulados para que se alcançe um bom resultado em termos de qualidade do modelo, já que aprende parte do padrão dos dados a partir dos dados sem rótulos.

Uma técnica já bem diferente das anteriores, é o **aprendizado por reforço**. Allen (Allen, 2020) nos explica o conceito principal dessa abordagem, que consiste na existência de um "agente" que interage executando ações num "ambiente", que por sua vez dá um retorno (*feedback*) a esse agente, usualmente na forma de uma "recompensa".

Tal recompensa pode ser entendida especificamente como um contador. O objetivo do agente é maximizar esse contador. Nenhuma informação é dita ao agente sobre como ele aumenta esse contador, ou porquê ele conseguiu aumentar, ele irá definir suas ações de acordo com as respostas dadas pelo ambiente.

Assim, tudo que ele sabe é se houve a recompensa ou não, e irá preferir as ações que fizeram o contador aumentar e preterir aquelas que fizeram ele diminuir, sem nunca existir rótulos ou respostas esperadas.

Outra técnica é a **detecção de anomalias**, cujo objetivo é ter uma descrição de como os dados considerados "normais" se parecem, e usa-se esse agrupamento para detectar se novos dados estariam "fora" desse padrão. Um exemplo é a detecção de fraudes.

Também pode-se citar a técnica de **estimação de densidades**, que tem como objetivo a estimação da função densidade de probabilidade de um conjunto de dados gerados por algum processo aleatório.

2.1.1 O problema do sobreajuste dos modelos

O sobreajuste (*overfitting*) é o primeiro desafio que deve ser enfrentado quando realizados uma tarefa de aprendizado de máquina. Um modelo de aprendizado de máquina só é considerado válido ou útil, se ele chega a um ponto de existir nenhum ou muito pouco sobreajuste.

O sobreajuste ocorre quando um modelo foi treinado exageradamente para o conjunto de dados conhecidos, ou seja, o conjunto utilizado para o treino. De forma que, ao se deparar com dados novos, desconhecidos, perde a capacidade de saber o que fazer, ou seja, erra muito nas previsões.

Isto pode ocorrer tanto nas tarefas de classificação, quanto nas tarefas de regressão. Utilizando o exemplo dado por Allen (Allen, 2020), imaginemos um modelo de classificação de fotos de animais. Um modelo 100% sobreajustado irá *decorar* as cores de cada um dos pixels dessa imagem, de forma que todos eles serão necessários para ele identificar se tal foto é um gato, ou um cachorro, etc.

Mudando apenas a cor ou a posição de um dos pixels de uma das imagens, e supondo que essa imagem modificada não fazia parte do conjunto de imagens do treinamento, o modelo não será capaz de fornecer uma resposta válida ou confiável. A figura 2.4 ilustra esse exemplo.

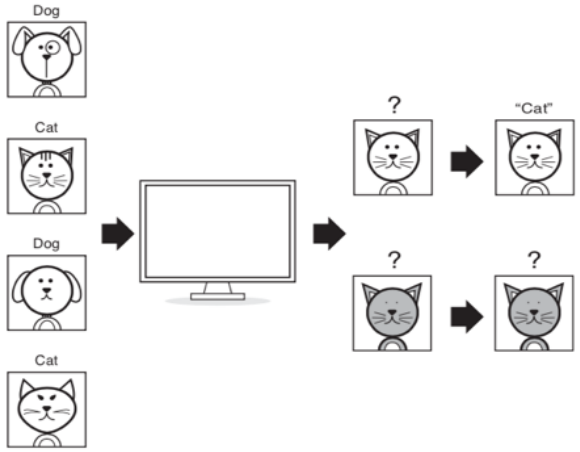

Figura 2.4: *Sobreajuste num modelo de classificação.*[a]

[a]Extraído de (Allen, 2020)

Isto é o sobreajuste, o modelo fica perfeito para os dados de treino, mas fica totalmente cego para os dados do mundo real, desconhecidos. Ele também está presente nos modelos de regressão. Consideremos o exemplo dado por Allen (Allen, 2020).

Seja um gráfico que relaciona a metragem ao quadrado de terrenos, no eixo X com o valor pago por eventuais compradores, no eixo Y. É um típico problema de regressão, já que queremos saber, para metragens de terrenos ainda desconhecidos, qual um valor esperado para sua compra ou venda.

O modelo mais simples seria o de uma regressão linear, explicado em detalhes na próxima seção, em que basicamente, traçamos uma reta do tipo $y = bx + a$, cujo valor y irá servir de valor esperado para nosso modelo, para qualquer x ainda desconhecido. Uma ilustração desse modelo hipotético está na figura 2.5.

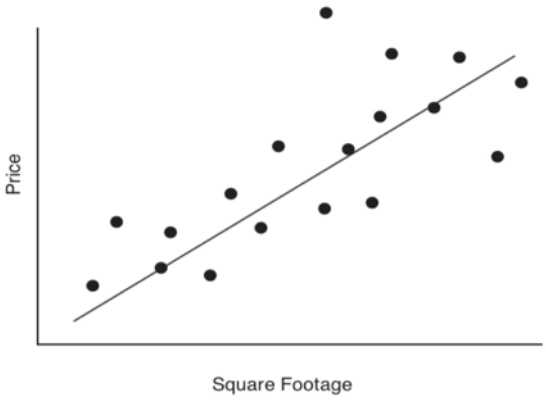

Figura 2.5: *Modelo de regressão, ajuste linear.*[a]

[a]Extraído de (Allen, 2020)

Podemos não ficar satisfeitos com esse ajuste linear, já que praticamente nenhum ponto está contido perfeitamente na reta ajustada. E assim, supomos um novo modelo, dessa vez

de um polinômio quadrático, isto é, um modelo do tipo $y = cx^2 + bx + a$.

A diferença é que antes tínhamos 2 parâmetros para ajustar, a e b, e agora temos um terceiro parâmetro, c.[2] Um ajuste possível, é ilustrado na figura 2.6.

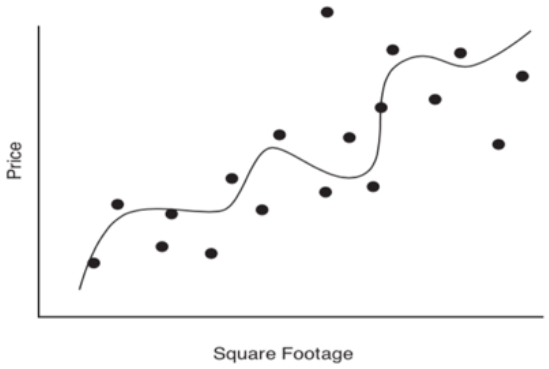

Figura 2.6: *Modelo de regressão, polinômio quadrático.*[a]

[a]Extraído de (Allen, 2020)

Aparentemente, esse modelo é melhor do que o anterior, pois a função está melhor ajustada aos dados conhecidos. Poderíamos pensar que quanto mais parâmetros, isto é, quanto maior o grau do polinômio a ser ajustado, mais adequado o modelo estará aos dados.

Isso pode chegar até o extremo em que, possuindo n+1 dados, tentamos ajustar um polinômio de grau n. Tal ajuste está ilustrado na figura 2.7.

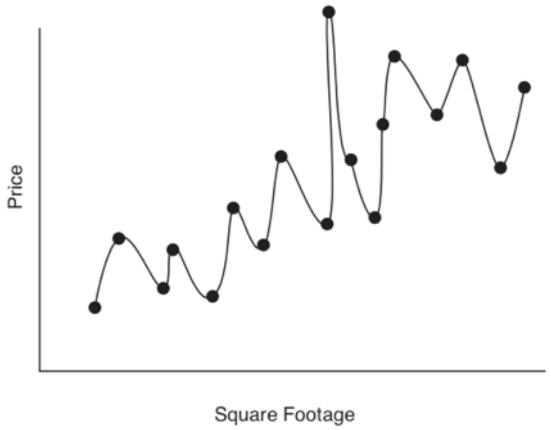

Figura 2.7: *Modelo de regressão, polinômio de grau n.*[a]

[a]Extraído de (Allen, 2020)

Com isso, teremos o caso extremo de um modelo de regressão 100% sobreajustado. Ele está perfeitamente ajustado aos dados conhecidos, mas retornará resultados espúrios para

[2]É importante mencionar que, em outros contextos, o número de parâmetros livres de um modelo é conhecido como o número de *graus de liberdade* desse modelo.

qualquer outro x que não pertença a esse conjunto, perdendo totalmente a capacidade de *generalização*, conforme explicado por Allen (Allen, 2020).

O objetivo dessa discussão é apontar que devemos ser parcimoniosos na escolha do nosso modelo, tentando buscar o menor número possível de parâmetros a serem corretamente ajustados, de modo a dar ao mesmo tempo uma boa perfomance no conjunto de treinamento (mas não perfeita), e também uma capacidade igualmente boa de generalização para os dados desconhecidos.

2.2 Alguns algoritmos de aprendizagem

Conhecendo os tipos mais comuns de aprendizagem, o próximo passo nesse caminho da ciência de dados é conhecer os diferentes algoritmos de aprendizagem, alguns sendo mais usados em tarefas supervisionadas, outros em tarefas não-supervisionadas e alguns podendo ser utilizados em ambos os tipos de tarefas.

É importante também conhecer para quais tarefas do mundo real eles foram concebidos, pois essa etapa é muito importante, dentro do contexto humano da ciência de dados, para se adquirir intuição e auxiliar na escolha do algoritmo ou dos algoritmos mais úteis que possam modelar algum novo problema.

2.2.1 Regressão Linear

Um dos pioneiros e mais simples problemas de aprendizagem de máquina supervisionada. É, aliás, um método nascido e desenvolvido na *estatística*, possuindo soluções fechadas, como por exemplo o método dos mínimos quadrados.

Porém, o uso de técnicas computacionais é muito bem-vinda, se estamos lidando com volumes de dados muito grandes, e que exigem manipulações aritméticas de matrizes, como multiplicação, inversão, diagonalização, etc.

Formalmente, supomos um modelo de regressão linear múltipla, associando um conjunto de variáveis independentes $X_1, X_2, \ldots, X_p$, que representam os dados, a uma variável dependente Y, que representa a resposta esperada, escrevemos o seguinte modelo.

$$Y = \mathrm{E}(Y|X_1 = x_1, \ldots, X_p = x_p) + \epsilon = \beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p + \epsilon \tag{2.1}$$

onde ϵ é o erro aleatório, para o qual é assumido uma distribuição normal de média 0.

Temos portanto p+1 parâmetros a serem ajustados em nosso modelo. Uma solução conhecida para o caso univariado, retirada de Magalhães e Lima (Magalhães e Lima, 2002) (páginas 332–336), e que pode ser generalizada, é dada pelo método dos mínimos quadrados.

Se possuímos n observações disponíveis, tanto das variáveis independentes X_i quanto das variáveis dependentes Y_i, e denotando da seguinte maneira:

$$Y = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{bmatrix}, \; X = \begin{bmatrix} 1 & x_{11} & \dots & x_{1p} \\ 1 & x_{21} & \dots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \dots & x_{np} \end{bmatrix}, \; \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix}, \; \epsilon = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}$$

Usando essa notação, podemos reescrever a equação (2.1), para n observações:

$$Y = X\beta + \epsilon \tag{2.2}$$

Esse problema possui uma solução algébrica dada pelo método dos mínimos quadrados, que nos fornece uma estimativa do vetor β, que é:

$$\hat{\beta} = (X'X)^{-1}X'Y \tag{2.3}$$

Que envolvem as operações de matrizes mencionadas acima. E que assume uma definição baseada na minimização da função de erro quadrático, feita de forma algébrica.

As técnicas de aprendizado de máquina entram em jogo se queremos utilizar uma outra função de erro, ou se queremos uma outra abordagem para a minimização da função de erro escolhida, como por exemplo o método do gradiente descendente, explicado em detalhes no Apêndice A.

A utilização do gradiente descendente ou de outro método de otimização qualquer é motivado pela maior eficiência computacional desses métodos em comparação com a inversão de matrizes de ordem $p \times p$ necessária no cálculo direto de 2.3. A justificativa é facilmente verificada em problemas do mundo real, onde, na maioria dos casos, p>1.000.

Outras técnicas também podem ser utilizadas quando estamos no contexto de aprendizado de máquina. Daniil Korbut (Korbut, 2017) citam as técnicas de regularização, que são úteis para se evitar o sobreajuste nos modelos de regressão.

A ideia da regularização é adicionar parâmetros sem importância prática às somas dos quadrados na função de erro quadrático, com o objetivo de minimizar os valores dos parâmetros de interesse. Explicações mais detalhadas podem ser encontradas em Géron (Géron, 2019) (páginas 196–205).

Só é possível verificar a necessidade dessas e outras técnicas *sentindo na pele* quando estamos tentando resolver um problema de ciência de dados, e não é o objetivo deste trabalho esmiuçar os algoritmos de regressão linear em tais detalhes.

É possível utilizar a abordagem puramente estatística, lidando diretamente com as equações 2.2 e 2.3, a vantagem de se utilizar técnicas computacionais eficientes permanece imprescindível quando p é grande.

2.2.2 Regressão Logística

Suponhamos agora que queremos prever a probabilidade de que algo ocorra ou não. Por exemplo, queremos decidir, a partir de uma base de dados de características de clientes

conhecidos e de suas compras, se um novo cliente, dadas apenas suas características, irá ou não comprar um certo produto.

Poderíamos, a princípio, utilizar uma regressão linear, se usarmos características independentes nada nos impediria de usar a modelagem vista anteriormente. O resultado seria uma função que dada a lista de características de um cliente novo, retornaria um número. Porém esse número sofreria de problemas de interpretação.

Poderíamos supor que se esse número fosse grande, então há alta probabilidade do cliente comprar, ou se o número fosse muito pequeno, ou até mesmo negativo, que ele teria baixa probabilidade de comprar o produto. Mas essa linha de decisão seria ruim, pois, o que nos garantiria que essa interpretação representaria a realidade?

Uma alternativa é modelarmos em termos de probabilidades reais, ou seja, um modelo cuja resposta fosse um número entre 0 e 1, de forma que a interpretação probabilística fosse muito mais natural. Esta é a motivação do modelo de regressão logística.

Por causa da natureza da previsão, que irá retornar basicamente, sim ou não, este pode ser considerado um algoritmo de classificação, apesar do nome e do funcionamento interno que caracterizariam-na como uma regressão não-linear. Além disso, é um algoritmo supervisionado, já que nosso modelo é construído utilizando dados de compras de clientes já conhecidos.

É uma regressão-classificação não-linear, pois o modelo é dado, por Grus (Grus, 2016), para cada par de características (X_i) e resposta (Y_i), por:

$$Y_i = f(X_i\beta) + \epsilon \tag{2.4}$$

em que f é a função *sigmóide* ou *logística*, dada por:

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

O que queremos fazer é maximizar as probabilidades de quando $Y_i = 1$ ou $Y_i = 0$. Ou seja, o objetivo é maximizar a probabilidade de que os dados com valores conhecidos assumam os valores esperados de Y_i.

Esta probabilidade é definida, em Grus (Grus, 2016), por:

$$P(y_i|x_i, \beta) = f(x_i\beta)^{y_i}(1 - f(x_i\beta))^{1-y_i} \tag{2.5}$$

em que, $y_i = 1$ ou $y_i = 0$, para toda observação indexada por i.

Uma vez que queremos maximizar essa probabilidade, dada uma amostra, o parâmetro a ser ajustado é β, o que configura a expressão (2.5) como uma *verossimilhança*. Assim, desejamos um algoritmo que irá maximizar a verossimilhança, ou ainda a log-verossimilhança de nossa amostra, e irá retornar o $\hat{\beta}$ que o faça.

Novamente, no contexto de aprendizado de máquina, o mais eficiente será utilizar algum algoritmo de gradiente descendente e estocástico, para realizar esse trabalho, motivo

pelo qual é tratado como um algoritmo de aprendizagem, em oposição ao procedicimento do cálculo direto de $\hat{\beta}$.

A seguir, com o $\hat{\beta}$ em mãos, poderemos classificar novos clientes, aplicando suas características de volta na expressão (2.4), com o $\hat{\beta}$ ajustado. A resposta será uma probabilidade, para a qual poderemos definir um *corte*, sendo o mais simples definir que, se essa probabilidade for maior do que 0.5 diremos que o cliente irá comprar, se for menor, não irá comprar.

Alternativamente, podemos ter como resposta a probabilidade diretamente, sem preocuparmos com não um mas vários *cortes*, para então gerarmos estratégias distintas para clientes com *alta*, *média* ou *baixa* probabilidade de compra de certo produto, por exemplo.

2.2.3 Árvores de decisão

Uma **árvore de decisão** é outro exemplo de algoritmo supervisionado usado principalmente para classificação, mas que também podem ser usadas para regressão. É definida por Grus (Grus, 2016) como sendo uma estrutura que representa um número de possíveis caminhos de decisão e um resultado para cada caminho.

Apesar dessa definição genérica, um exemplo muito útil é lembrar da classificação das famílias pertencentes ao reino vegetal, vejamos na figura 2.8, sem nos preocuparmos com os detalhes biológicos.

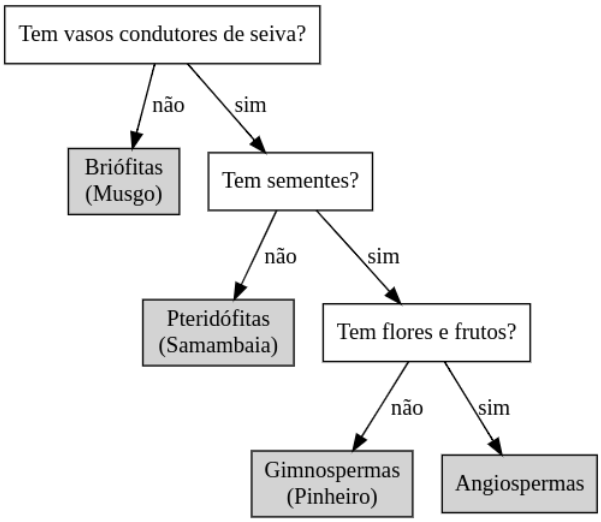

Figura 2.8: *Árvore de decisão classificando famílias de plantas.*

Pode-se notar a natureza didática das árvores de decisão, pois elas são de fato intuitivas. Dada uma planta, fizemos perguntas simples e diretas sobre suas características, que estão representadas pelos quadros de fundo branco de onde saem flechinhas indicando as respostas negativas e positivas, estes quadros são os **nós de decisão**.

Seguir as flechas, ou seja, as decisões de cada pergunta ou característica, nos permitirá classificar uma planta em alguma das famílias representadas pelos quadros de fundo cinza,

de onde não saem mais flechinhas. Estes são os **nós-folhas**, que são as respostas dadas por essa árvore de classificação.

Isto nos leva ao funcionamento do algoritmo de uma árvore de decisão. Segundo Grus (Grus, 2016), para construir uma árvore de decisão, precisamos decidir quais perguntas fazer e em qual ordem. Cada pergunta irá separar as possibilidades restantes de acordo com as respostas.

Outro aspecto importante será decidir os *cortes* de cada pergunta, ou seja, se estamos lidando com valores contínuos (por exemplo, o tamanho do caule), será preciso definir, como no caso da regressão logística, um valor que será o limiar entre uma resposta negativa e positiva para essa pergunta (por exemplo, *o caule é grande?*).

De acordo com Grus (Grus, 2016), seria útil escolher perguntas cujas respostas darão muita informação sobre o que a árvore deverá prever. Esta informação pode ser mensurada com o conceito de **entropia**, que neste contexto, representa a incerteza associada aos dados.

Seja um conjunto de dados S, para os quais pudéssemos rotular alguma dentre n classes, $C_1, \ldots, C_n$. Se todos os dados de S possuirem a mesma classe, a entropia H de S será zero. Se os pontos estiverem igualmente espalhados entre as classes, então a entropia será a máxima possível.

De modo geral, se p_i é a proporção de dados que pertencem à classe C_i, então a entropia de S será:

$$H(S) = -p_1 \log_2 p_1 - \ldots - p_n \log_2 p_n \tag{2.6}$$

onde Grus (Grus, 2016) utiliza a convenção: $0 \log 0 = 0$.

Se analisarmos rapidamente cada termo, como uma função $f(p) = -p \log p$, estendida com a convenção acima, vemos que é uma função que se aproxima de 0 quando p se aproxima de 0 ou de 1, e se afasta de 0, caso contrário, isto é, tem uma concavidade negativa, cruzando o eixo nos pontos $p = 0$ ou $p = 1$.

Assim, a entropia de S será maior quando os dados estiverem mais espalhados, o que significa que há maior incerteza na base, enquanto que será menor quando os dados estiverem mais concentrados em poucas classes, o que indicaria menor incerteza, o que configura o comportamento que gostaríamos de obter.

Sabendo computar a entropia do conjunto total S que representa uma *árvore*, o próximo passo é calcular a entropia de *ramos* da árvore, que serão obtidas a partir de perguntas que irão separar o conjunto S em dois ou mais subconjuntos.

Se estamos lidando com características categóricas, cada valor assumido por essa característica é uma das respostas possíveis, gerando um ramo para cada, ou seja, uma seta de decisão. Se uma característica é numérica e contínua, daí há uma liberdade na escolha da pergunta, podendo ser usada por exemplo a média ou a mediana para separar os dados entre abaixo ou acima desse corte.

Generalizando a mesma definição anterior, se dividirmos os dados de S em subconjuntos $S_1, \ldots, S_m$, para uma característica com m valores distintos possíveis, cada um contendo

proporções $q_1, \ldots, q_m$ respectivas, então a entropia da árvore será a soma ponderada das entropias de cada ramo:

$$H(S) = q_1 H(S_1) + \ldots + q_m H(S_m) \tag{2.7}$$

em que cada termo $H(S_i)$ é obtido normalmente por (2.6).

A partir dessa definição, já podemos construir um primeiro algoritmo de árvore de decisão. De acordo com Grus (Grus, 2016), um algoritmo *ganancioso*, pode ser construído como se segue. Dadas as características presentes em S, ou seja, as variáveis explicativas, computamos para cada uma o valor H dado por (2.7).

Escolhemos como nosso primeiro nó de decisão, a característica que nos der o menor valor de entropia H. Com isso, sabemos que essa característica é a que melhor separa as classes, que serão os nós folhas (as respostas), ao menos nesse momento.

Assim obtemos 2 ramos, e em cada um deles podemos repetir esse mesmo procedimento. Se fizermos isso para todas as características, então cada ramo de nossa árvore poderá eventualmente conter apenas uma das classes possíveis, o que seria perfeito para o conjunto de treino, mas seria péssimo para dados novos, pois nossa árvore estaria completamente *sobreajustada*.

É por isso que para um modelo de árvore de decisão, existem comumemente 2 **hiperparâmetros**[3] muito importantes segundo Korbut (Korbut, 2017), que são o número máximo de nós de decisão, ou seja, o número de perguntas e o número mínimo de nós folhas, ou seja, de classes por ramo.

Ajustando esses parâmetros, podemos garantir que uma árvore não-perfeita mas boa o suficiente para o conjunto de treino e seja igualmente boa para o conjunto de teste, e dessa forma, para dados desconhecidos, para que ela seja útil para a tarefa de classificação proposta.

Por fim, vale citar a ressalva de Korbut (Korbut, 2017) de que um modelo utilizando apenas uma árvore é raramente usado. Mas combinando várias árvores criamos os algoritmos chamados de *Florestas aleatórias*, que estão dentre os mais versáteis e utilizados para tarefas de classificação.

2.2.4 K-médias

Este algoritmo é o ponto central de todas as tarefas de aprendizado não-supervisionadas conhecidas como **agrupamento**. A ideia é agrupar os dados de acordo com um conceito de *distância* e com a suposição de que *proximidade* implica em *similaridade* no espaço em que calculamos essas distâncias.

O procedimento geral é agrupar os dados em k agrupamentos, em que, cada grupo terá uma *média* que irá caracterizar esse grupo, daí o nome do algoritmo. Segundo Korbut (Korbut, 2017), podemos partir de k pontos escolhidos aleatoriamente e nomeá-los como os centros (ou médias) de cada grupo.

[3] Parâmetros inerentes ao modelo e não aos dados, e que devem ser escolhidos durante a parte prática do treinamento do algoritmo.

A seguir, calculamos as distâncias de cada ponto aos centros, e incluímos em cada grupo aqueles pontos que estão mais próximos de algum dos centros. Isto é, dado um ponto, incluímos ele no grupo que possui o centro que está mais próximo dele.

Então, o centro real de cada grupo é calculado (a média, por exemplo), e o processo acima se repete. Esse ciclo é repetido, idealmente, até que haja convergência, ou seja, calcular a média dos grupos não altera a pertinência de mais nenhum ponto dentre os k grupos.

A maior questão aqui é, justamente, a escolha de k, que é a princípio desconhecido. Além disso, para k pontos iniciais escolhidos, pode haver uma convergência local dos agrupamentos, e não global, análogo ao problema de maximização/minimização global de funções.

2.3 As redes neurais e o *perceptron*

De acordo com Géron (Géron, 2019), as primeiras redes artificiais foram introduzidas em 1943 pelo neurofisiologista Warren McCulloch e o matemático Walter Pits através de um modelagem computacional do funcionamento conjunto de neurônios no cérebro de animais, enquanto realizam computações complexas de lógica. Esta foi a primeira **arquitetura** de uma rede neural artificial.

Esse começo promissor levou as pessoas a acreditarem que logo haveriam máquinas realmente inteligentes, o que ficou registrado na cultura da época, principalmente em séries televisivas de ficção científica como *Star Trek* e outras, mas conforme aponta Géron (Géron, 2019), essa promessa logo se mostrou inalcansável, ao menos era o que parecia ao final dos anos 60.

A partir dos anos 80, surgiram novas arquiteturas e melhores técnicas de aprendizagem, embora sua evolução fosse lenta devido ao poder computacional limitado da época. Atualmente, no entanto, isto mudou: há poder computacional em casa e na nuvem, há a internet e fórums de compartilhamento de códigos e conhecimentos em programação e ciência de dados, em resumo o mundo atual está consolidado numa era digital.

Por essa razão Géron (Géron, 2019) nos diz que estamos numa nova onda de entusiasmo sobre as redes neurais artificiais, sendo que esse entusiasmo leva o nome de **Deep Learning**, ou aprendizado profundo, e o uso desse adjetivo ajuda a descrever as redes neurais, constituídas de milhares de neurônios, que são utilizadas em várias aplicações de nosso dia-a-dia na internet.

Podemos citar a classificação de bilhões de imagens realizadas pelo *Google*, reconhecimento de fala realizado pela *Siri* da *Apple*, o sistema de recomendações de vídeos do *Youtube* e da *Netflix* e outras plataformas de *streaming*, e até mesmo os jogadores artificiais de xadrex ou do jogo *Go*. As redes neurais artificiais estão vivas em nosso mundo. Mas como funcionam na prática, por detrás dessa aparência de ficção científica?

Uma definição para uma rede neural artificial dada por Rosangela Ballini (Ballini, 2000) é a de um sistema de processamento paralelo e distribuído contruído em um formato e com funcionalidade que se parece com o arranjo de um sistema nervoso biológico, sendo

compostos por elementos computacionais chamados neurônios, que são organizados e interligados em padrões semelhantes aos neurônios biológicos.

Uma rede neural artificial é um dentre vários métodos de classificação, ou seja, de aprendizado supervisionado, embora ela também possa ser usada para aplicações de aprendizado não supervisionado. De acordo com Kopec (Kopec, 2019), ele é utilizado como um classificador não-linear, e por isso pode ser utilizado para classificar ou prever quaisquer tipos de funções, que podem ou não ter uma relação linear com o tempo ou com qualquer outro domínio no qual estejam definidas.

Na figura 2.9 está uma representação de um neurônio biológico. Ele recebe impulsos elétricos de entrada através dos dentritos, que são transmitidos ou não através do núcleo, caso sejam ativados por ele, para os terminais de saída dos axônios. Os neurônios se comunicam através de sinapses, que são ligações entre os dentritos de um e os axônios de outro que realizam a transmissão dos sinais.

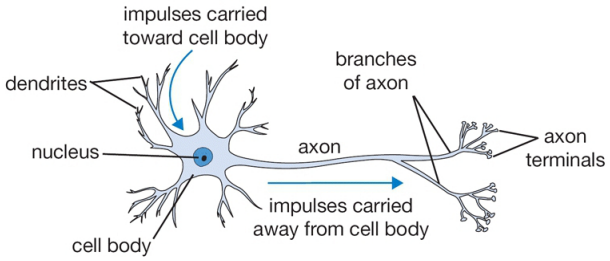

Figura 2.9: *Representação de um neurônio biológico.*[a]

[a]Extraído de https://cs231n.github.io/neural-networks-1/

Dá-se o nome de *perceptron* de camada única (*single-layer perceptron*) ou simplesmente *perceptron* a uma das simples arquiteturas de rede neural artificial, criada em 1957 por Frank Rosenblatt (Rosenblatt, 1958). Uma ilustração conceitual dela está na figura 2.10. Existem atualmente diversas outras arquiteturas de redes neurais, mas a extensão mais imediata que podemos citar de um perceptron constituído de uma camada de neurônios são as redes *perceptron* multi-camadas (*multi-layer perceptron*).

Os neurônios são representados por círculos, dentro deles há um valor numérico que intuitivamente podemos atribuir ao nível ou grau de ativação do neurônio, mesmo que no caso biológico se restrinja aos valores 0 e 1, ou seja, ativados ou não. Cada coluna de neurônios representa uma camada, nesse caso, da esquerda para a direita temos a camada de entrada, a camada oculta e a camada de saída. As linhas representam as ligações entre os neurônios, sendo que cada neurônio de uma camada está ligado a todos da camada anterior.

O perceptron de camada única consiste de uma camada de neurônios de entrada, uma camada oculta de neurônios usados na otimização, e uma camada de saída, que irá conter os dados previstos, ou ainda as probabilidades do dado pertencer a alguma das classes que a rede poderá classificá-lo. E é o fato de haver uma camada oculta nesta rede que a define como sendo de “camada única”. Caso houvessem mais do que uma camada oculta, ela seria do tipo “multi-camadas” mencionada acima.

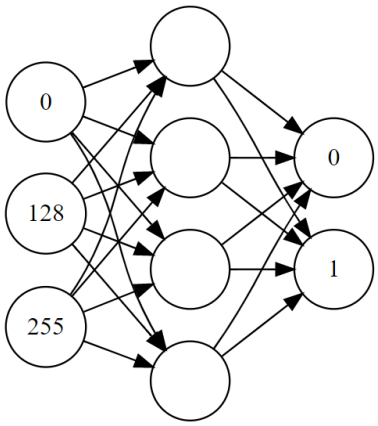

Figura 2.10: *Rede neural simples, o perceptron de camada única.*

De modo a entendermos as bases matemáticas do algoritmo, podemos começar de uma rede ainda mais básica, a partir um *perceptron* que seja constituído de apenas 1 neurônio na única camada oculta. Esta rede super simplificada, que está na figura 2.11, pode ser útil para para o entendimento uma vez que neste caso será possível acompanhar graficamente o resultado da execução do algoritmo.

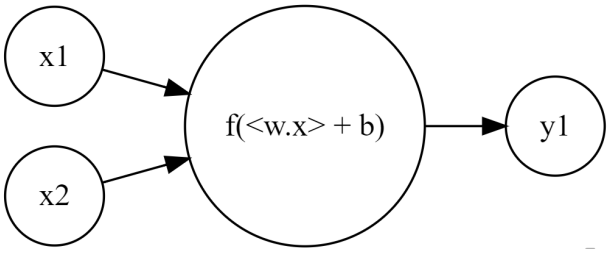

Figura 2.11: *Rede neural mais simples ainda, apenas um neurônio oculto.*

Esta rede possui 2 neurônios na camada de entrada, que são os números reais x_1 e x_2, 1 neurônio na camada oculta, no qual está a sua função de ativação $f(x_1 w_1 + x_2 w_2 + b)$, e 1 neurônio na camada de saída, que neste caso é um número real y_1. Pode-se notar a semelhança dessa rede neural artificial com a sua inspiração biológica com a ajuda da figura 2.12.

Temos os sinais de entrada (como o x_1) vindos como viriam os sinais dos axônios de outros neurônios. Eles entram pela camada de entrada da rede, ou dentritos do neurônio. A camada oculta processa as entradas com os pesos, definindo o formato final do sinal através de sua função de ativação, que aqui pode ser uma função real qualquer, mas com funcionalidade similar ao do núcleo do neurônio que ativa/transmite ou não o sinal recebido por ele. Por fim o sinal é enviado à camada de saída, ou aos axônios do neurônio, concluíndo o processamento.

A partir desta analogia podemos compreender o funcionamento básico da rede artificial *perceptron*. Ela recebe uma lista de valores como entrada, que podemos representar por um vetor real x. O neurônio oculto representa uma transformação linear neste vetor, que podemos escrever como o produto escalar por um outro vetor real, o vetor de **pesos** w, ou seja, $< w.x >$, que é o produto escalar usual dos números reais. A seguir, somamos um

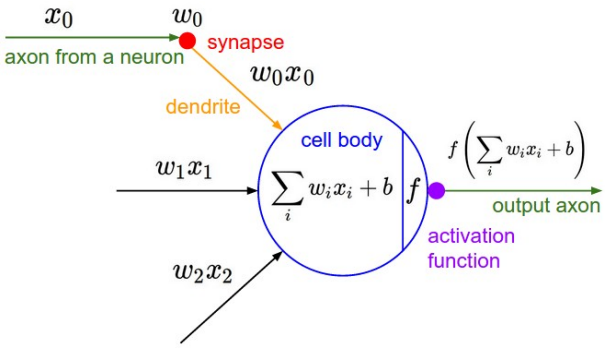

Figura 2.12: *Representação de um neurônio artificial.*[a]

[a]Extraído de https://cs231n.github.io/neural-networks-1/

outro número real b que é chamado de **viés**, que possui o mesmo papel que a constante de interceptação da reta com o eixo vertical de um ajuste linear.

Por fim, é aplicada uma função de ativação não-linear sobre esta transformação, o que configura a saída deste neurônio: $f(< w.x > + b)$, que é transmitida ao neurônio de saída, que pode aplicar uma transformação semelhante ou outra qualquer, dependendo da função de ativação utilizada em cada camada da rede. Por simplicidade mostramos uma rede bem simples, mas na prática podem haver muito mais camadas ocultas, e cada uma delas assim como a camada de saída, podem ter muitos neurônios cada.

Este processo de entrada, processamento e saída da rede é chamado de **feedforward**, e consiste no nível mais fundamental do *perceptron*. A partir daí, a forma como a rede será treinada, é o que define se ela será utilizada para um aprendizado supervisionado ou não-supervisionado.

Uma vez que estamos lidando com o aprendizado supervisionado, dever ser utilizado um algoritmo de treinamento que forneça à rede pares conhecidos de vetores de entradas e saídas esperadas, e um **critério de avaliação** de quão boa é a performance da rede para aproximar as suas saídas das saídas esperadas.

Este critério é uma função que fornece uma medida da distância entre as saídas obtidas pela rede e as saídas esperadas, que é genericamente chamada de função de custo (*cost function*). Se denotarmos por y uma saída conhecida, e por $a^{(L)}$ uma saída obtida pela última camada, exemplos comumente usados são as normas usuais como a distância euclidiana ($(a^{(L)^2} + y^2)^{1/2}$), a função de erro absoluto ($|a^{(L)} - y|$), e a função de erro quadrático médio ($(a^{(L)} - y)^2$) , (*mean square error*, MSE), que é a usada no algoritmo descrito por Kopec (Kopec, 2019) e que será usado neste trabalho.

Um dos algoritmos de treinamento que minimizam uma função de custo é o gradiente descendente (*gradient descent*), que segundo Géron (Géron, 2019) é um algoritmo muito geral e que serve para encontrar soluções ótimas para uma grande variedade de problemas de otimização. Detalhes de seu funcionamento podem ser vistos no Apêndice A.

2.4 Outras arquiteturas de redes neurais

Existem dezenas de arquiteturas de redes neurais artificiais, uma breve olhada na quantidade de arquiteturas listadas no website *The neural network zoo*[4] demonstra que essa brincadeira é levada muito a sério. A representação gráfica ali criada é útil para o entendimento das características das diversas arquiteturas, graças aos padrões de cores e de formas geométricas utilizadas.

A primeira estrutura é o *perceptron* simples, de apenas um neurônio, que processa n entradas através de uma transformação linear seguida de uma função de ativação. É exibido no canto superior esquerdo da Figura 2.13.

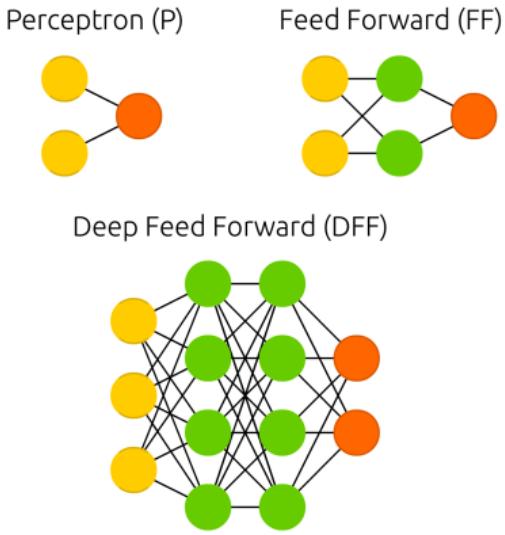

Figura 2.13: *As redes neurais recorrentes:* perceptron, feedforward *e* deep feedforward.[a]

[a]Extraído de https://www.asimovinstitute.org/neural-network-zoo/

No canto superior direito está a versão *feedforward*, possuindo uma camada oculta(representada pelos círculos verdes), e embaixo está a versão *deep feedforward*, que possui múltiplas camadas ocultas, e número variado de neurônios em todas as camadas, e é de fato a versão que foi implementada nesse trabalho.

As características em comum das arquiteturas derivadas da rede *perceptron*, chamadas de **redes neurais sequenciais**, são a conexão que existe entre cada neurônios de uma camada com todos os neurônios da camada anterior, e a forma que a informação é transmitida da rede, num sentido único, da camada de entrada(os círculos amarelos) para a camada de saída(os círculos vermelhos).

A próxima arquitetura em destaque define o que são as **redes neurais recorrentes**, ilustrada na Figura 2.14. Nesse tipo de rede a informação pode ir para frente e para trás, além disso pode passar pelo mesmo neurônio oculto mais de uma vez, o que determina o nome recorrente. Esses neurônios são representados pela cor azul na figura.

[4]https://www.asimovinstitute.org/neural-network-zoo/

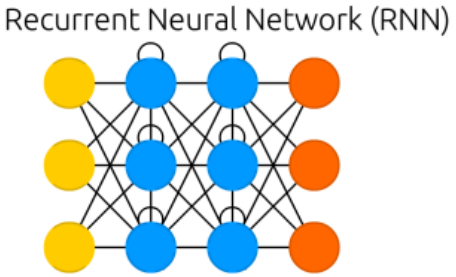

Figura 2.14: *As redes neurais recorrentes.*[a]

[a]Extraído de https://www.asimovinstitute.org/neural-network-zoo/

Esse tipo de rede, segundo Kopec (Kopec, 2019) é o mais usado em problemas em que os dados utilizados possuem uma dependência da ordem em que são obtidos e possuem entradas contínuas, por exemplo, os dados de uma série temporal de dados, ou seja, em que a previsão de um dado , o que representaria o futuro, depende da ordem em que estão os dados já conhecidos, o que representaria os dados do passado.

Outra arquitetura muito utilizada é o das **redes neurais covolucionais**, ilustradas na Figura 2.15. Segundo Kopec (Kopec, 2019) essas redes foram projetadas e usadas com sucesso para classificação de imagens "pesadas", como fotos de galáxias obtidas em telescópios.

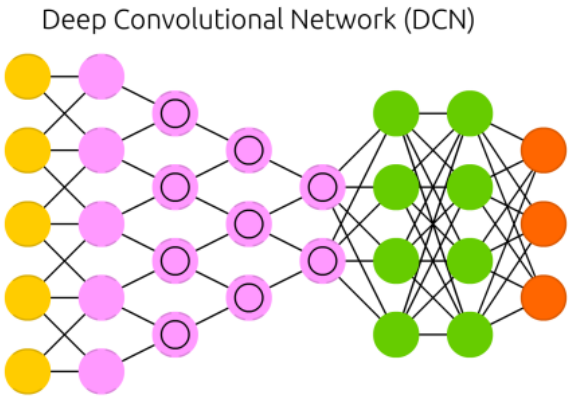

Figura 2.15: *As redes neurais convolucionais.*[a]

[a]Extraído de https://www.asimovinstitute.org/neural-network-zoo/

Em resumo, são redes em que os neurônios de entrada não são conectados totalmente com a primeira camada oculta, mas o que acontece é que vários conjuntos distintos de neurônios da camada de entrada são conectados a várias camadas ocultas separadamente.

A seguir a união dessas camadas ocultas, exibidas como círculos rosas, conectam-se em cascata, perdendo conexões progressivamente até que um número reduzido se conecta a outras camadas, dessa vez camadas ocultas simples, que são os círculos verdes, que

conectam-se em formato *feedforward* até o final da rede com a camada de saída.

Existem ainda outras dezenas de arquiteturas, algumas bem alternativas no formato e nas conexões entre as camadas, fica aqui um único exemplo dentre elas, que é a arquitetura de **redes neurais extremas**. Está ilustrada na Figura 2.16.

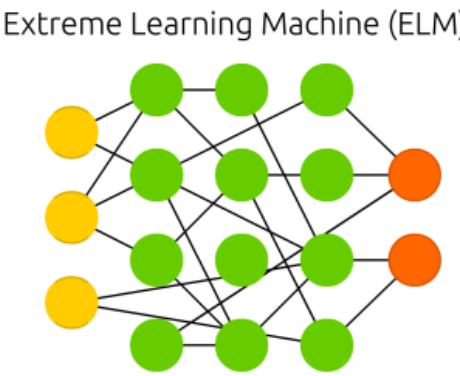

Figura 2.16: *Redes neurais extremas.*[a]

[a]Extraído de https://www.asimovinstitute.org/neural-network-zoo/

As conexões ocorrem de forma não-sequencial, e até mesmo aleatória entre as camadas ocultas, configurando um exemplo curioso de arquitetura, e discussões sobre essas outras arquiteturas não fazem parte do escopo desse trabalho.

Capítulo 3

Perceptron multi-camadas

Neste capítulo é descrita a implementação e funcionamento de uma versão do algoritmo *perceptron*, feito a partir de um núcleo básico disponibilizado no livro de Kopec (Kopec, 2019), e a partir do qual foram feitas modificações e criação de novos métodos de treinamento, de validação e de avaliação do treinamento.

O perceptron aqui implementado tem o objetivo de ser utilizado muito mais para fins didáticos do que práticos e pode ser usado para tarefas de aprendizagem contanto que sejam problemas que envolvam bases de dados de tamanho pequeno ou mediano, e neste capítulo é apresentadao um caso de uso para um problema de classificação de dados.

Na última parte desse capítulo é exibida uma biblioteca de *machine learning* utilizada nas aplicações reais de *deep learning* de redes neurais, muito mais avançada, com muitos outros recursos que vão além do escopo que esta versão simples do *perceptron* é capaz de lidar. Esta será a biblioteca de redes neurais que será utilizada nas partes práticas deste trabalho.

3.1 Matemática do algoritmo de retropropagação

Para a aprendizagem supervisionada foi utilizado o algoritmo de retropropagação (*retropropagation*), que consiste na minimização de uma função de custos, a partir do gradiente, ou seja, da derivada desta função de custos, neste caso o erro quadrático médio, conforme foi definido no capítulo anterior.

De acordo com Kopec (Kopec, 2019), o perceptron consiste de uma rede cujo sinal, ou seja, os dados, se propagam em uma só direção, da camada de entrada para a camada de saída, passando pelas camadas ocultas uma a uma, e por isso o nome de rede *feedforward* ao perceptron. Por sua vez, o erro que determinamos na camada final propaga-se no caminho inverso, sendo distribuídas correções da saída para a entrada, afetando aqueles neurônios que foram mais responsáveis pelo erro total. Por isso o nome de retropropagação.

Estendendo as definições já usadas no capítulo anterior, segue a derivação matemática do algoritmo de retropropagação. Como ficará claro mais à frente, podemos derivar as contas para apenas um neurônio por camada sem perda de generalidade. Dessa forma, se

temos uma rede com L camadas, o erro quadrático para um neurônio da camada de saída (a camada L) será:

$$C_0 = (a^{(L)} - y)^2$$

onde y é a saída esperada, e $a^{(L)}$ é a saída de um neurônio da camada de saída.

Temos que C_0 é uma função de $a^{(L)}$, uma vez que y é um valor fixo conhecido. Por sua vez, temos que de modo geral a saída de um neurônio é uma função do tipo:

$$a^{(L)} = \sigma(w^{(L)}\ a^{(L-1)} + b^{(L)})$$

onde escrevemos $a^{(L-1)}$ é a saída do neurônio da camada anterior, $w^{(L)}$ é o **peso** atribuido a essa saída, o que seria o parâmetro angular A na Figura 2.11, e $b^{(L)}$ é o chamado **viés** desse neurônio, análogo ao parâmetro linear de uma reta. Por fim temos a **função de ativação** que escrevemos como σ que é aplicada à essa equação linear.

Nota-se que internamente à função de ativação, um neurônio se comporta como uma transformação linear dos neurônios da camada anterior. Caso tivéssemos n neurônios na camada anterior à de saída, teríamos então n pesos, denotados com índice i dessa forma: $\{w_i^{(L)}\}_{i=1}^{n}$. Cabe assim à função de ativação, dar o comportamento não-linear à rede perceptron.

Como o objetivo é minimizar C_0, temos que calcular a influência dos pesos e dos viéses nesse custo. Já sabemos que isso será obtido com o gradiente, isto é, a derivada dessa função em relação a esses parâmetros que, são os únicos que podemos otimizar.

De forma mais clara, temos que no início do treinamento da rede, atribuímos valores aleatórios aos pesos e aos viéses, e então executamos o *feedforward*, de forma que a rede irá calcular sequencialmente os valores de saída em todas as suas camadas, obtidos a partir dos dados de entrada, que serão fixos, e desses parâmetros inicialmente aleatórios. A partir daí, poderemos otimizar esses parâmetros, exatamente da forma que estamos construindo.

O cálculo dessas derivadas é feito segundo a regra da cadeia, e adicionalmente iremos denotar a transformação linear interna à função de ativação por $z^{(L)} = w^{(L)}\ a^{(L-1)} + b^{(L)}$, de forma que $a^{(L)} = \sigma(z^{(L)})$. Assim, ficamos com as derivadas para a camada de saída:

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}} \tag{3.1}$$

$$\frac{\partial C_0}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}} \tag{3.2}$$

Para a camada de saída, podemos calcular diretamente cada termo dessas derivadas:

$$\frac{\partial C_0}{\partial a^{(L)}} = 2(a^{(L)} - y) \ \propto \ (a^{(L)} - y) \tag{3.3}$$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)}) \tag{3.4}$$

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)} \tag{3.5}$$

$$\frac{\partial z^{(L)}}{\partial b^{(L)}} = 1 \tag{3.6}$$

O que resulta, fazendo todas as substituições, em:

$$\frac{\partial C_0}{\partial w^{(L)}} = a^{(L-1)}\, \sigma'(z^{(L)})\, (a^{(L)} - y) \tag{3.7}$$

$$\frac{\partial C_0}{\partial b^{(L)}} = \sigma'(z^{(L)})\, (a^{(L)} - y) \tag{3.8}$$

Na equação 3.3 ocultamos o termo constante 2 sob um símbolo de proporção, que a seguir iremos também ocultar, uma vez que usaremos o algoritmo do gradiente descendente, e assim, em seu lugar, e na verdade, todas as derivadas aqui mostradas serão multiplicadas pelo termo η, a **taxa de aprendizagem**, conforme explicado no Apêndice A.

Analogamente, podemos pensar numa forma de fazer esses cálculos para as camadas ocultas. A princípio, podemos calcular:

$$\frac{\partial C_0}{\partial a^{(L-1)}} = \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}} \tag{3.9}$$

Usando o fato de que:

$$\frac{\partial z^{(L)}}{\partial a^{(L-1)}} = w^{(L)} \tag{3.10}$$

Agora, seja a i-ésima camada oculta tal que $1 < i < L$, se observarmos a equação 3.9, e fizermos $i = L - 1$, usando a equação 3.10, ficamos com:

$$\frac{\partial C_0}{\partial a^{(i)}} = w^{(i+1)} \frac{\partial a^{(i+1)}}{\partial z^{(i+1)}} \frac{\partial C_0}{\partial a^{(i+1)}} \tag{3.11}$$

Podemos observar que há um mesmo termo duplo que aparece tanto nas equações 3.1 e 3.2 quanto na equação 3.11 acima, de forma que apenas o índice da camada é diferente. Para simplificar podemos nomear esse termo de *delta da camada i*:

$$\Delta^{(i)} = \frac{\partial a^{(i)}}{\partial z^{(i)}} \frac{\partial C_0}{\partial a^{(i)}} \tag{3.12}$$

Simplificando todas as demais expressões usando essa definição, ficamos com:

$$\frac{\partial C_0}{\partial w^{(i)}} = a^{(i-1)}\Delta^{(i)} \tag{3.13}$$

$$\frac{\partial C_0}{\partial b^{(i)}} = \Delta^{(i)} \tag{3.14}$$

Como vemos, as derivadas que precisamos todas dependem desse termo Δ, que por sua vez depende do cálculo do termo $\frac{\partial C_0}{\partial a^{(i)}}$ que será calculado de 2 formas distintas:

$$\begin{aligned} &\frac{\partial C_0}{\partial a^{(i)}} = w^{(i+1)}\,\Delta^{(i+1)} \implies \\ &\Delta^{(i)} = \sigma'(z^{(i)})\,w^{(i+1)}\,\Delta^{(i+1)} \end{aligned} \tag{3.15}$$

para as camadas ocultas.

$$\begin{aligned} &\frac{\partial C_0}{\partial a^{(L)}} = (y - a^{(L)}) \implies \\ &\Delta^{(L)} = \sigma'(z^{(L)})\,(y - a^{(L)}) \end{aligned} \tag{3.16}$$

para a camada de saída.

Percebe-se a natureza recursiva do algoritmo, onde o caso base é calculado na camada de saída, e que o cálculo vai propagando-se para as camadas ocultas, em direção à camada de entrada. Por essa mesma razão, pudemos derivar as contas para uma camada, e no fim elas estão prontas pra serem implementadas para qualquer número de camadas ocultas.

Outro fato útil é que a expressão interna do neurônio é uma transformação linear, assim as contas podem ser facilmente ajustadas para o caso geral em que há n_i neurônios em dada camada i da rede, conforme já explicado, e que será detalhado diretamente nos trechos de código que serão mostrados a seguir na implementação propriamente dita.

3.2 Implementação do algoritmo de retropropagação

A implementação seguiu uma estrutura orientada a objetos, voltemos então à representação visual da rede perceptron, a partir da imagem 3.1. Cada círculo representa um neurônio, cada coluna vertical de neurônios é uma camada da rede, uma das camadas, a camada oculta nesse caso, está destacada em roxo na imagem. As setas representam as conexões entre as camadas de neurônios, cada neurônio de uma camada está ligado a todos os neurônios da camada anterior, o sentido dessa conexão é da esquerda pra direita, o que indica o processo de *feedforward* da rede.

A implementação do perceptron deste trabalho teve como base a implementação feita por Kopec (Kopec, 2019), a partir da qual foram adicionados outros recursos, como o viés

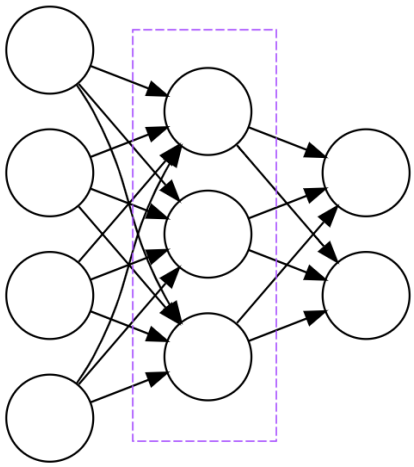

Figura 3.1: *Visão estrutural da rede perceptron. A linha tracejada destaca uma das camadas da rede.*

dos neurônios, não presentes na implementação de Kopec, como o uso da biblioteca *Numpy* para o uso de seus métodos mais eficientes para lidar com listas de números de ponto flutuante. Além dessa base, também estão implementadas várias outras classes, que serão explicadas de modo mais geral.

3.2.1 O neurônio

O primeiro passo é implementar a classe *Neuron* para representar cada neurônio. Esta é uma classe de entidade bem simples, contendo apenas um construtor, e o método *output* que recebe os valores de entrada para esse neurônio e faz o cálculo da transformação linear e a seguir aplica e retorna o valor da função de ativação utilizada, que é passada como parâmetro ao construtor da classe. A listagem 3.1 abaixo mostra este método, em conjunto com o construtor da classe.

```
1 class Neuron:
2     def __init__(self, weights, bias, learning_rate, ativacao, der_ativacao):
3         '''(list[float], float, float, Callable, Callable) -> None'''
4         ...
5
6     def output(self, inputs):
7         '''(list[float]) -> float'''
8         self.output_cache = np.dot(inputs, self.weights) + self.bias
9         return self.ativacao(self.output_cache)
```

Programa 3.1: *Trecho da classe Neuron*

A função *np.dot* da biblioteca *Numpy* é utilizada para calcular o produto escalar entre os valores de entrada e os pesos desse neurônio. O valor da transformação linear é armazenado num atributo de classe antes da aplicação da função de ativação, pois será utilizado mais à frente durante o treinamento da rede.

3.2.2 A função de ativação

A função de ativação possui o papel de ativar ou não a saída de um neurônio, conforme visto no capítulo anterior, e a forma com que essa ativação ocorre é definida pela função utilizada. Aqui o termo *ativar* significa que a função irá retornar um valor mais próximo de 1 enquanto que uma não-ativação retornará um valor mais próximo de 0. Essa é uma restrição para a função de ativação para a camada de saída, que será sempre da forma:

$$f : \mathbb{R} \longrightarrow [0, 1]$$

No caso do neurônio biológico, quando dizemos que ele ativa/transmite ou não o sinal elétrico que chegou até ele, é como se ele *retornasse* apenas 0 ou 1. De fato, poderíamos até usar uma função similar a essa em alguma camada de nossa rede artificial, e este tipo de função escada tem a seguinte definição:

$$f(x) = \begin{cases} 1 & \text{se} \quad x \geq 0 \\ 0 & \text{se} \quad x < 0 \end{cases}$$

A utilização dessa função de ativação, conforme nos diz Grus (Grus, 2016), faria com que um neurônio fizesse simplesmente a distinção entre espaços separados pelo hiperplano de pontos tal que $< w.x > + b = 0$, ou seja, o hiperplano definido pelos pontos de entrada cuja transformação linear resultasse em zero.

Esta função é claramente não contínua e portanto não diferenciável, e precisamos de uma função de ativação que o seja, uma vez que algumas das equações da otimização que calculamos anteriormente, dependem da expressão de sua derivada. É por essa razão, que Grus (Grus, 2016) nos explica que passou-se a considerar uma aproximação suave da função escada, essa aproximação é a função *sigmoid*:

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

Ela retorna valores somente no intervalo $[0, 1]$, igualmente à função escada, sua inspiração. Essa característica, no entanto, não é uma restrição para as camadas ocultas da mesma forma que é para a camada de saída, uma vez apenas a camada de saída será comparada com valores esperados no intervalo $[0, 1]$. A sua derivada pode ser facilmente calculada, e sua expressão simplifica-se como:

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

Podemos comparar o comportamento dessas funções de ativação no gráfico presente na imagem 3.2 abaixo. A seguir, na listagem 3.2, um trecho do script `util.py` com a implementação da função *sigmoid* e de sua derivada.

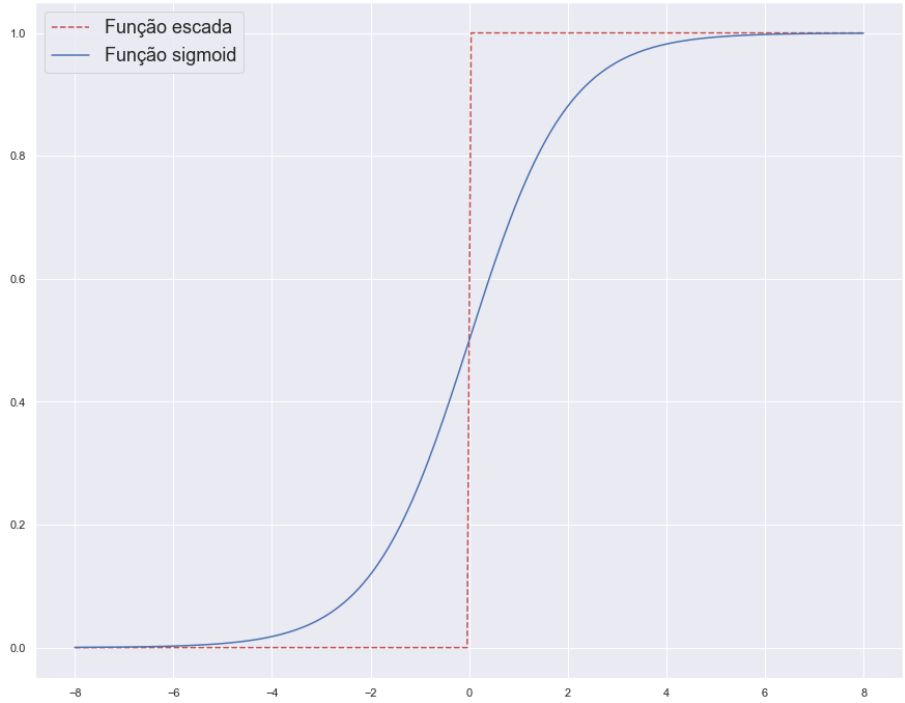

Figura 3.2: *Comparação entre as funções de ativação do tipo escada e a* sigmoid.

```
1 def sigmoid(x):
2     '''(float) -> float'''
3     return 1.0 / (1.0 + np.exp(-x))
4
5 def der_sigmoid(x):
6     '''(float) -> float'''
7     sig = sigmoid(x)
8     return sig * (1 - sig)
```

Programa 3.2: *Trecho do script util.py*

Com a popularização das redes neurais, várias outras funções de ativação foram criadas para ativarem as camadas ocultas do treinamento, devido aos problemas que podem acontecer ao se utilizar função *sigmoid*. Podemos identificar um desses problemas analisando seu gráfico. Vemos que ela se aproxima de 1, que é a ativação máxima, rapidamente a partir de $x > 4$, e aproxima-se simetricamente de zero com valores a partir de $x < -4$.

Como o método do gradiente tenta ajustar os valores dos pesos a partir dos valores de saída e esses ajustes dependem da derivada da função de ativação, temos que levar em conta o comportamento da derivada da função *sigmoid*, o qual podemos observar a partir de seu gráfico na figura 3.3.

Como podemos ver, a derivada retorna valores sempre menores do que 1, e além disso aproxima-se de 0 tão rapidamente quanto a *sigmoid* aproxima-se de 1. Isso faz com que atualizações para valores de saída que já estão muito altos não sejam efetivos para diminuí-los, pois justamente nessa região a derivada está muito próxima de 0. Essa é a desvantagem da função *sigmoid*.

Um problema relacionado a este é que se a regra da cadeia em 3.1 e 3.2, com as derivadas

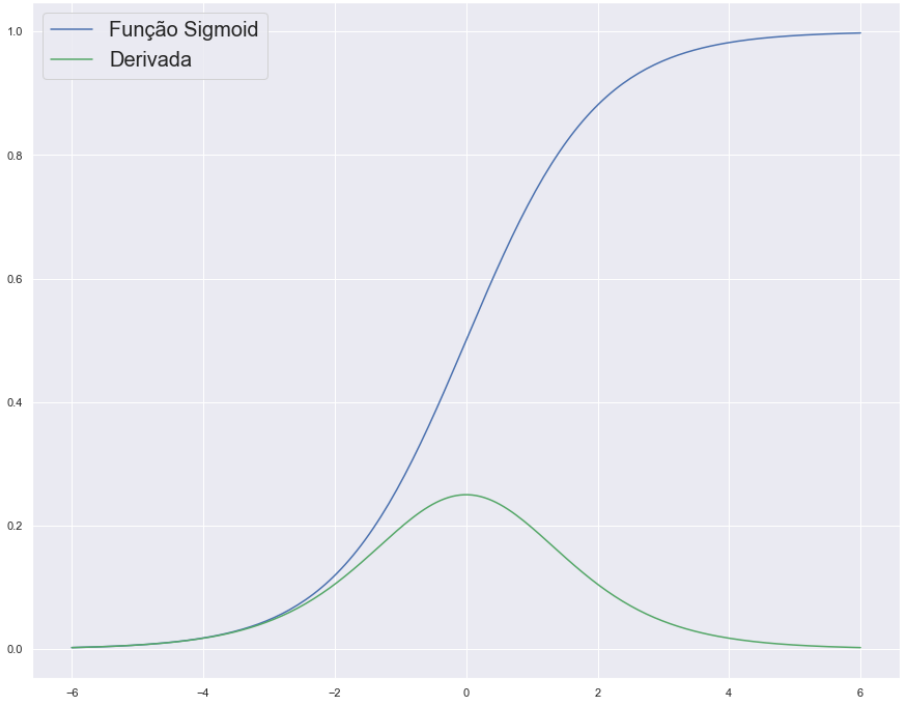

Figura 3.3: *Gráficos da função* sigmoid *e sua derivada.*

da função de ativação dadas em 3.4 multiplicadas através das várias camadas, pode resultar num número muito grande, se todas as derivadas resultarem em valores maiores do que 0, ou resultar num número muito próximo de 0 se todas as derivadas forem menores do que 0. Isto faz com que atualizações dadas pelo gradiente sejam instáveis. Este é o problema descrito por Matheus Facure (Facure, 2017a) e nomeado como problema do gradiente explodindo/desvanecendo.

Assim, conforme nos diz Facure (Facure, 2017b), a utilização da função *sigmoid* não é mais recomendada em problemas que envolvam de redes neurais maiores, sendo bem comum o problema do gradiente explodindo, já que a derivada é sempre maior do que 0. Porém, ele também diz que alguns modelos probabilísticos de variáveis binárias, modelagem de problemas biológicos onde ela é uma aproximação mais plausível da ativação elétrica-biológica, e também alguns modelos não supervisionados de redes tem restrições que fazem com que seja não só desejável como também necessário o uso da função *sigmoid*.

A próxima função de ativação é a o tangente hiperbólico $\tanh(x)$, ela é similar à *sigmoid* e pode ser escrita em função dela. Ela retorna valores no intervalo $[-1, 1]$ mas sua derivada retorna valores mais próximos de 1, chegando ao valor máximo de 1 quando $x = 0$. A expressão em função da função *sigmoid* e a derivada da função tangente hiperbólica são dadas por:

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

Na figura 3.4 podemos ver o gráfico da função e de sua derivada, a partir do que podemos notar como a derivada da *tanh* retorna valores maiores do que a derivada da

função *sigmoid*.

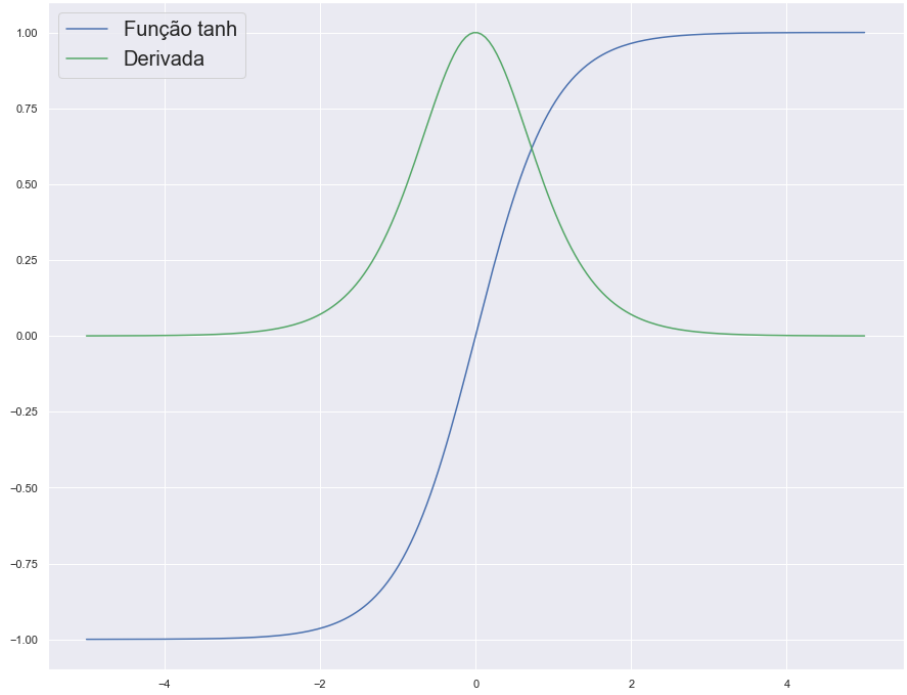

Figura 3.4: *Gráficos da função tangente hiperbólico e sua derivada.*

O próximo avanço é conseguido com a função de ativação linear retificada (**RELU**). Essa função é quase a função identidade, exceto que na região negativa do domínio ela vale identicamente 0. Ela não é derivável no ponto $x = 0$, mas podemos estender a definição fixando seu valor em 1 nesse ponto. Sua definição e de sua derivada estendida é dada por:

$$ReLU(x) = max\{0, x\} \qquad ReLU'(x) = \begin{cases} 1, & \text{se } x \geq 0 \\ 0, & \text{c.c.} \end{cases}$$

Podemos ver seus gráficos, na figura 3.5, a seguir. Usar essa função de ativação torna até mesmo a execução do código mais rápida, uma vez que não há cálculos matemáticos a serem feitos, apenas uma função de máximo que é trivial. Além disso, podemos notar que a derivada se mantém com o valor 1 constante enquanto o neurônio é ativado, sendo uma forma de tentar resolver o problema do gradiente explodindo/desvanecendo, além de agilizar o processo de treinamento.

Essa é a razão, conforme explica Facure (Facure, 2017b), dessa função ter contribuido para o recente aumento de popularidade das redes neurais. Adicionalmente, Bing Xu (Xu *et al.*, 2015) ressalta que outra vantagem das funções do tipo *RELU*, além de resolver o problema do gradiente explodindo/desvanescendo, é a de aumentar a velocidade da convergência do algoritmo de treinamento rumo a um mínimo da função de custos.

Uma desvantagem da função *RELU* é a chance de neurônios serem desativados permanentemente, já que uma vez que ele zera, a função de ativação e sua derivada são ambos 0,

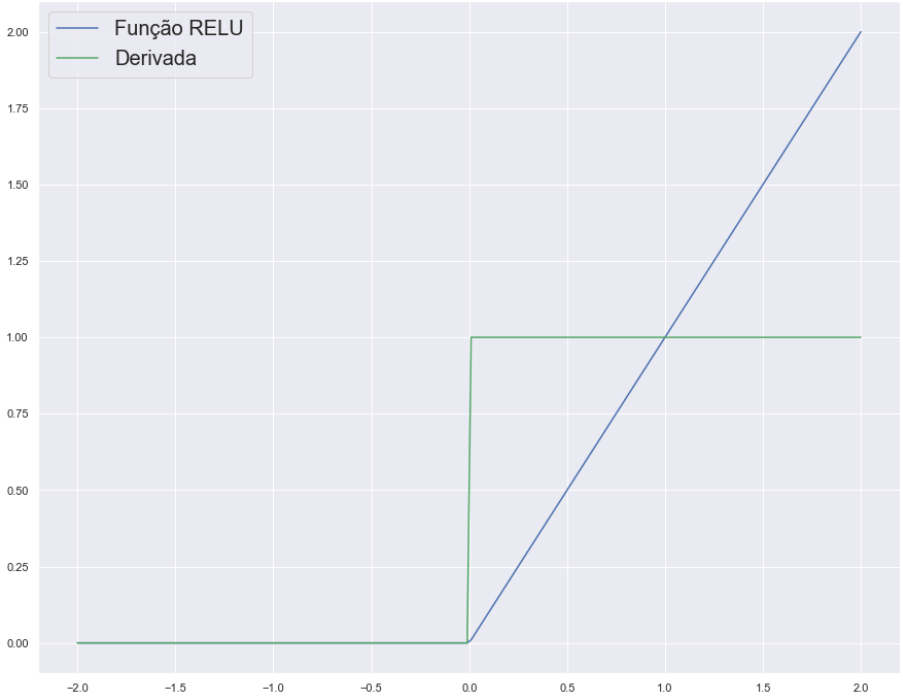

Figura 3.5: *Gráficos da função* RELU *e sua derivada.*

de forma que ele nunca mais irá aumentar durante o treinamento, tornando-se neurônios *mortos*.

O próximo avanço foi dado pela função conhecida como *Leaky RELU*. Quase identica à *RELU*, exceto que na parte negativa do domínio ao invés de 0 a função retorna x/α, onde $\alpha \in (0, \infty)$. Isso já imediatamente corrige o problema dos neurônios desativados. A definição da função e de sua derivada, dada por Xu (Xu *et al.*, 2015), é:

$$LeakyReLU(x, \alpha) = \begin{cases} x, & \text{se } x \geq 0 \\ x/\alpha, & \text{c.c.} \end{cases} \qquad LeakyReLU'(x, \alpha) = \begin{cases} 1, & \text{se } x \geq 0 \\ \alpha, & \text{c.c.} \end{cases}$$

A partir dos resultados dos estudos feitos por Xu (Xu *et al.*, 2015), a função *Leaky RELU*, e suas variações, se saíram consistentemente melhores do que a *RELU* para as bases de dados de pequeno e médio portes. Além disso, ele testou a performance para diferentes valores de α, obtendo os melhores resultados com $\alpha = 5.5$. Este parâmetro é conhecido como **vazamento**, que dá o nome à função. Podemos ver o seu comportamento no gráfico da imagem 3.6.

Por fim, temos a função de unidade linear exponencial *ELU*, proposta por Djork-Arné Clevert (Clevert *et al.*, 2015), que é definida, com $\alpha > 0$, por:

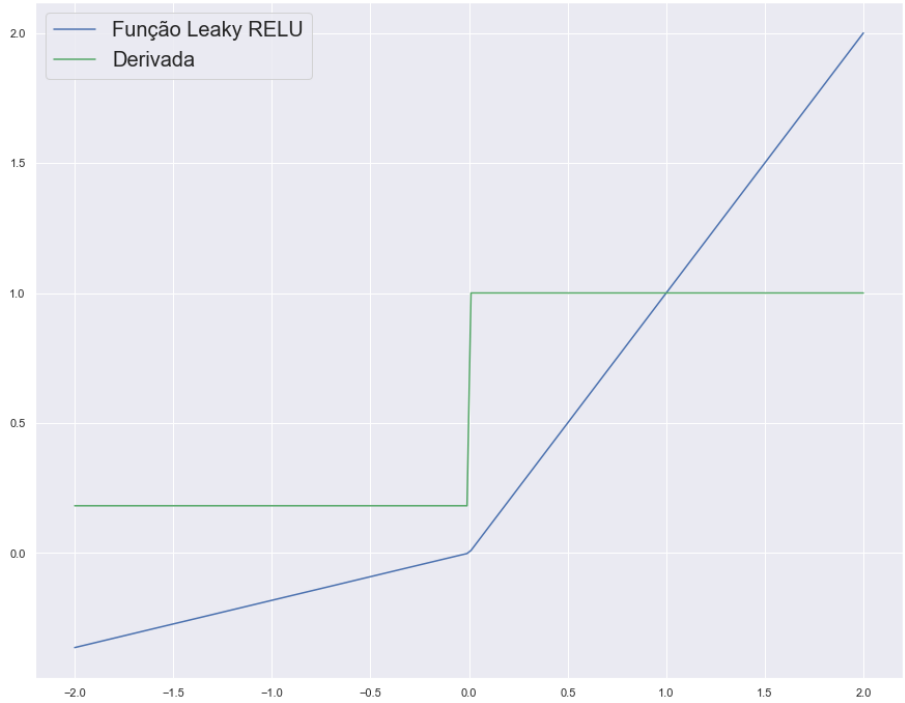

Figura 3.6: *Gráficos da função* Leaky RELU *e sua derivada.*

$$ELU(x,\alpha) = \begin{cases} x, & \text{se } x \geq 0 \\ \alpha(e^x - 1), & \text{c.c.} \end{cases} \qquad ELU'(x,\alpha) = \begin{cases} 1, & \text{se } x \geq 0 \\ ELU(x,\alpha) + \alpha, & \text{c.c.} \end{cases}$$

Em seu artigo, Clevert (Clevert *et al.*, 2015) utiliza o valor $\alpha = 1$, e com a função *ELU* conseguiu perfomances melhores, tanto de resultados mais corretos, quanto de velocidade de treinamento, em relação às funções *RELU* e *Leaky RELU* para as mesmas bases de dados avaliadas por XU (Xu *et al.*, 2015), mesmo com o uso da função exponencial em sua definição o que em teoria deveria diminuir a performance do treinamento. Podemos observar o comportamento dessa função e de sua derivada, com $\alpha = 1$ no gráfico mostrado na figura 3.7.

Testes feitos em condições similares por Facure (Facure, 2017b), mostram que essa diferença não é tão significativa em relação à *Leaky RELU*, mas que ambas, *Leaky RELU* e a *ELU* são sim melhores do que a original *RELU*, o que é consistente com o fato delas resolverem teoricamente as desvantagens dela. E são todas obviamente melhores escolhas do que a função *sigmoid*, em todos os estudos acima citados.

Na prática, podemos testar qual função de ativação irá performar melhor para o problema que queremos resolver. A abordagem mais comum, conforme descrita por Facure (Facure, 2017b) é utilizar a função *Leaky RELU* nas camadas ocultas, sendo o modo mais simples de obtermos bons resultados graças ao seu comportamento. Podemos avaliar a utilização das outras de acordo com o problema em questão, dado que algumas funções se saem melhor em alguns contextos específicos como é o caso da função *sigmoid*.

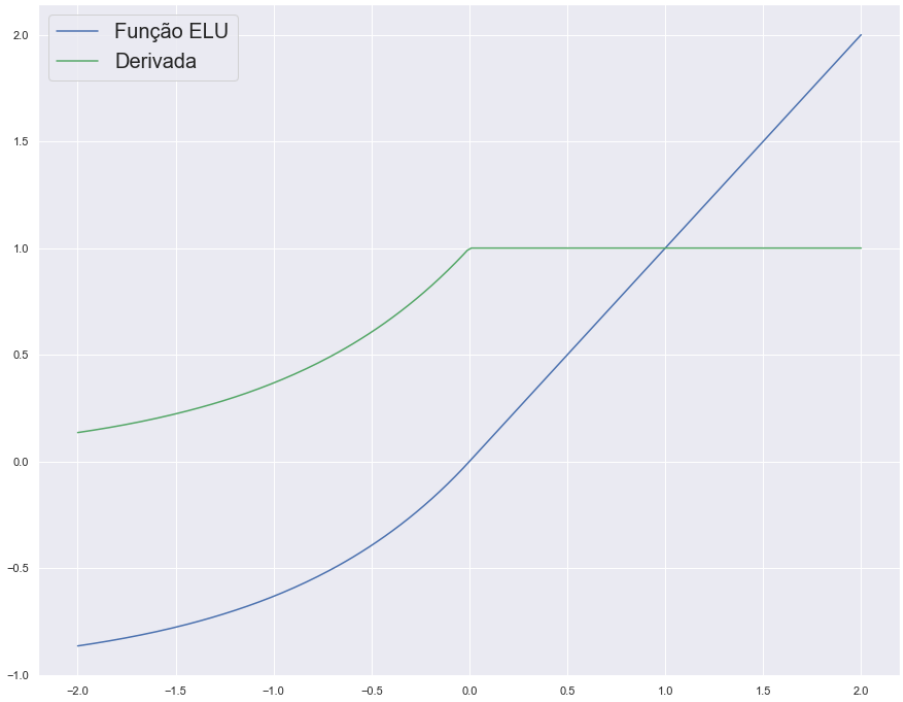

Figura 3.7: *Gráficos da função* ELU *e sua derivada.*

3.2.3 As camadas

A classe *Layer* representa uma camada de neurônios. Cada camada conecta-se com a sua camada anterior, com exceção da camada de entrada. Por essa razão, a rede perceptron possui um sentido único de conexão, que vai da entrada para a saída, passando por cada camada oculta. A classe é constituída de uma lista de objetos da classe *Neuron*, uma referência à camada anterior e uma lista para armazenar as saídas dos seus neurônios.

O construtor de *Layer* é responsável por inicializar seus neurônios. Nessa implementação todos os neurônios de uma camada irão usar a mesma função de ativação e a mesma taxa de aprendizagem. Além de receber esses parâmetros, o número de neurônios dessa camada, e a referência da camada anterior, o construtor inicializa os pesos de cada neurônio, lembrando que cada neurônio de uma camada possui a mesma quantidade de pesos do que a quantidade de neurônios da camada anterior, e também inicializa o viés de cada neurônio.

Na listagem 3.3 está o trecho do construtor que inicializa os pesos dos neurônios. Se a camada que estamos inicializando é a camada de entrada, então não criamos pesos e o viés, pois os valores de entrada serão utilizados diretamente como a saída dessa camada, este é o teste presente na linha 11 da listagem, pois a camada de entrada não possui referência a uma camada anterior já que ela é a primeira camada da rede.

A linha 12 de 3.3 inicializa os pesos dos neurônios da camada. Para fazer isso utiliza uma função que está definida no script `util.py`, que sortea números aleatórios para esses pesos seguindo uma distribuição normal de média 0 e desvio-padrão 0.3 e além disso *truncada* no intervalo $[-1, 1]$, para que nenhum peso esteja fora desse domínio e para que em média esse valor seja 0. O viés é inicializado com um valor constante, próximo de zero,

nesse caso com 0.01.

```
class Layer:
    def __init__(self, previous_layer, num_neurons, learning_rate,
                 ativacao=None, der_ativacao=None):
        '''(Layer, int, float, Callable, Callable) -> None
        Construtor da Camada de Neurônios
        '''
        ...
        for i in range(num_neurons):
            pesos = None
            bias = None
            if previous_layer is not None:
                pesos = normal_t.rvs(len(previous_layer.neurons))
                bias = 0.01

            neuron = Neuron(pesos, bias, learning_rate, ativacao, der_ativacao)
            self.neurons = np.append(self.neurons, neuron)
```

Programa 3.3: *Trecho da classe* Layer

Este procedimento é usado para tentar mitigar dois problemas que podem acontecer, conforme explicado por James Dellinger (Dellinger, 2019). Se inicializarmos os pesos com muitos números não tão próximos de 0, numa rede como muitos neurônios e muitas camadas, esses pesos podem somar-se rapidamente através das camadas, resultando em números com valores absolutos muito grandes na camada de saída o que pode prejudicar o treinamento e aprendizado da rede.

Se pelo contrário, inicializarmos todos os pesos com números muito próximos de 0, ocorre o problema oposto, os neurônios tem seus valores zerados, tornando-se *neurônios desativados*, que se tornam inúteis para o aprendizado já que serão ignorados durante o restante do treinamento.

Dellinger (Dellinger, 2019) distute esses problemas no contexto de redes bem grandes, com mais de 100 camadas, e exibe sua solução heurística que é utilizar uma distribuição normal (não-truncada) com média 0 e com desvio-padrão $\sqrt{2/n}$, sendo n o número de neurônios da camada anterior.

Para nossos fins didáticos, testei alguns desvios-padrão como 1, 0.3 e 0.1, em um dos exemplos que serão mostrados ainda nesse capítulo, e dentre eles, o valor 0.3 se saiu melhor sendo o suficiente para não explodir e nem desativar os neurônios da única camada oculta que foi usada na aplicação-exemplo em questão.

O desvio-padrão de 0.3 auxilia na tarefa de restringir os valores no intervalo $[-1, 1]$, sem que precisemos truncar muitos valores o que poderia aumentar a massa de probabilidade dos extremos -1 e 1, já que valores mais distantes da média do que 3 vezes o desvio-padrão são raramentes obtidos de uma distribuição normal.

Após inicializar cada neurônio, salvamos ele na lista de neurônios dessa camada, que é um dos atributos de classe discutidos no primeiro parágrafo. A próxima tarefa de uma camada é processar as entradas recebidas e retornar as saídas. Podemos observar esse comportamento na listagem 3.4.

```
def outputs(self, inputs):
        '''(list[float]) -> list[float]
        Armazena em cache as saidas dos neuronios e a retornam
        Se for uma camada de entrada, usa elas diretamente
        '''
        if self.previous_layer is None:
            self.output_cache = inputs
        else:
            self.output_cache = np.array([n.output(inputs) for n in self.neurons])
        return self.output_cache
```

Programa 3.4: *Trecho da classe* Layer

A camada de entrada não processa os dados, usando-os diretamente. As demais camadas devem processar cada neurônio, usando seu próprio método de processamento, aplicando a transformação linear e em seguida a função de ativação. O resultado é armazenado numa lista *numpy* que é o atributo de classe `output_cache`, que armazena as saídas dessa camada para uso posterior; por fim, a lista das saídas da camada é retornada.

A última tarefa da classe *Layer* é calcular os termos Δ definidos pelas equações 3.15 e 3.16, que definem respectivamente o cálculo que é feito se estamos calculando as derivadas para as camadas ocultas e o cálculo feito para a camada de saída. As duas versões são exibidas na listagem 3.5 abaixo.

```
def calcular_delta_camada_de_saida(self, expected):
        '''(list[float]) -> None'''
        for i, neuron in np.ndenumerate(self.neurons):
            der_cost = expected[i[0]] - self.output_cache[i]
            neuron.delta = neuron.der_ativacao(neuron.output_cache) * der_cost

    def calcular_delta_camada_oculta(self, next_layer):
        '''(Layer) -> None'''
        for i, neuron in np.ndenumerate(self.neurons):
            next_weights = np.array([n.weights[i[0]] for n in next_layer.neurons])
            next_deltas = np.array([n.delta for n in next_layer.neurons])
            der_cost = np.dot(next_weights, next_deltas)
            neuron.delta = neuron.der_ativacao(neuron.output_cache) * der_cost
```

Programa 3.5: *Trecho da classe* Layer

Os algoritmos são as traduções quase literais das equações 3.15 e 3.16. Podemos ver a natureza recursiva da regra da cadeia nas 2 linhas finais, onde usamos os deltas calculados da próxima camada para calcular os deltas da camada atual. O caso base é a função que calcula o delta da camada de saída. A lógica que orquestra essa recursão está implementada na próxima classe.

3.2.4 A rede

A classe *Network* representa a rede neural como um todo. Ela armazena uma lista de camadas, ou seja, objetos do tipo *Layer*, a partir dos parâmetros que recebe em seu construtor, que são a estrutura da rede que será criada, que é um vetor de inteiros que

representam as quantidades de neurônios para cada camada. Além disso, recebe a taxa de aprendizado que será utilizada em toda a rede, nessa versão, e quais as funções de ativação que serão utilizadas nas camadas ocultas e na camada de saída.

A listagem 3.6 exibe o trecho do construtor que cria cada camada e insere na lista de camadas do objeto da classe atual. A camada de entrada não possui camada anterior, nem função de ativação. Além disso, a camada de saída pode utilizar uma função de ativação diferente daquela utilizada pelas camadas ocultas, que usarão a mesma.

```
class Network:
    def __init__(self, layer_structure, taxa, ativacoes):
        '''(list[int], float, Tuple[Callable]) -> None'''
        ...
        self.layers = np.array([], dtype=np.float64)
        self.estrutura = layer_structure

        # camada de entrada
        input_layer = Layer(None, self.estrutura[0], taxa)
        self.layers = np.append(self.layers, input_layer)

        # camadas oculta(s)
        for previous, qtd_neurons in np.ndenumerate(self.estrutura[1::l]):
            next_layer = Layer(self.layers[previous[0]], qtd_neurons, taxa,
                               ativacoes[0], ativacoes[1])
            self.layers = np.append(self.layers, next_layer)

        # camada de saída
        output_layer = Layer(self.layers[-1], self.estrutura[-1], taxa,
                             ativacoes[2], ativacoes[3])
        self.layers = np.append(self.layers, output_layer)
```

Programa 3.6: *Trecho da classe* Network

A primeira tarefa da classe Network é o de processar entradas, fazendo elas atravessarem a rede, camada a camada, até a camada de saída, e retornar as saídas obtidas. É o processo de *feedforward* explicado no início do capítulo. Sua implementação mesmo para o caso geral de multi-camadas é bem simples, conforme exibido na listagem 3.7 abaixo.

```
def feedforward(self, entrada):
    '''(list[float]) -> list[float]'''
    ...
    saida = self.layers[0].outputs(entrada)
    for i in range(1, len(self.layers)):
        saida = self.layers[i].outputs(saida)
    return saida
```

Programa 3.7: *Trecho da classe* Network

A próxima tarefa é treinar a rede, passando uma lista de entradas e saídas esperadas, realizando o procedimento de *backpropagate* para atualizar os pesos e viéses dos neurônios de cada camada, tudo isso em sequência, para cada entrada fornecida. É o que está literalmente implementado na listagem 3.8 abaixo.

```
def train(self, entradas, saidas_reais):
```

```
    '''(list[list[floats]], list[list[floats]]) -> None'''
    ...
    for i, xs in enumerate(entradas):
        ys = saidas_reais[i]
        _ = self.feedforward(xs)
        self.backpropagate(ys)
        self.update_weights()
        self.update_bias()
return saida
```

Programa 3.8: *Trecho da classe* Network

Cada chamada à função *train* significa o procedimento de treinamento sendo executado uma única vez. Cada vez que a rede é treinada dizemos que ela avançou em uma **época** de treinamento. O treinamento consiste em primeiramente executar o *feedforward* para uma entrada, para que as camadas possam armazenar as saídas correspondentes aos valores atuais de seus parâmetros, os pesos e viéses, assim como as saídas em seus atributos *output_cache*, que serão usados pelo método *backpropagate* a seguir, de acordo com as equações que derivamos para o processo de treinamento.

O funcionamento do método *backpropagate* pode ser visto na listagem 3.9 a seguir. Tudo o que ele faz aqui é calcular os deltas das camadas, na ordem correta, começando pela camada de saída, e depois percorrendo as demais camadas do final para o início da rede, fazendo a chamada para cada objeto da classe *Layer* que constitui a rede.

```
def backpropagate(self, saidas_reais):
    '''(list[float]) -> None
    Calcula as mudanças em cada neurônio com base nos erros da saída
    em comparação com a saída esperada
    '''
    # calcula delta para os neurônios da camada de saída
    last_layer = len(self.layers) - 1
    self.layers[last_layer].calcular_delta_camada_de_saida(saidas_reais)

    # calcula delta para as camadas ocultas, da saída para o início da rede
    for l in range(last_layer - 1, 0, -1):
        self.layers[l].calcular_delta_camada_oculta(self.layers[l + 1])
```

Programa 3.9: *Trecho da classe* Network

A seguir, atualiza-se os pesos e os viéses com os métodos correspondentes, que podem ser visualizados na listagem 3.10. Como os valores são todos armazenados nos atributos de estado dos neurônios e das camadas, implementamos diretamente as contas das equações que obtemos para o método do gradiente e da regra da cadeia da retroprogação. Dessa forma, o que fazemos na classe *Network* é basicamente traduzir a matemática para a sintaxe da linguagem Python.

```
def update_weights(self):
    '''(None) -> None'''
    ...
    for layer in self.layers[1:]: # pula a camada de entrada
        for neuron in layer.neurons:
            for w in range(len(neuron.weights)):
                neuron.weights[w,] = neuron.weights[w,] + (neuron.learning_rate
                    * (layer.previous_layer.output_cache[w]) * neuron.delta)

def update_bias(self):
    '''(None) -> None'''
    ...
    for layer in self.layers[1:]: # pula a camada de entrada
        for neuron in layer.neurons:
            neuron.bias = neuron.bias + neuron.learning_rate * neuron.delta
```

Programa 3.10: *Trecho da classe* Network

A próxima responsabilidade da classe *Network*, uma vez que já foi treinada, é fazer a previsão de classes de novos dados de entrada. Isto é feito pelo método mostrado na listagem 3.11. Isto significa simplesmente processar as entradas fornecidas pelo método *feedforward*, que usará os parâmetros que foram ajustados anteriormente para fazer os cálculos.

```
def predict(self, entradas, interpretar):
    '''(list[list[floats]], Callable) -> list[list[floats]]
    '''
    self.previsoes = np.array([], dtype=np.float64)
    for entrada in entradas:
        self.previsoes = np.append(self.previsoes, interpretar(self.feedforward(entrada)))
    return self.previsoes.reshape(-1, 1)
```

Programa 3.11: *Trecho da classe* Network

Ao final, os dados da última camada são uma lista, isto é, um vetor de valores reais, que são interpretados por uma função que é passada por parâmetro que identifica a qual classe, previamente definida, pertence esse vetor de saída. A lógica dessa interpretação é externa à classe *Network* e será vista mais adiante. A lista de classes preditas em formato *numpy* é retornada.

A última responsabilidade dessa classe é calcular uma métrica de avaliação para esta rede, que irá servir de avaliação do quão boa a rede é para classificar os dados utilizados. A métrica mais simples a ser utilizada é a **acurácia** da previsão. Ela basicamente mede a proporção de classificações corretas dentre todas as classificações realizadas para um conjunto de dados. Essa lógica bem simples é implementada na listagem 3.12 abaixo.

```
def validate(self, esperados):
    '''(list[list[floats]], list[list[floats]], Callable) -> float
    '''
    ...
    corretos = 0
    for y_pred, esperado in zip(self.previsoes, esperados):
        if y_pred == esperado:
```

```
8                 corretos += 1
9         acuracia = corretos / len(self.previsoes)
10        return acuracia
```

Programa 3.12: *Trecho da classe* Network

Nota-se que ela utiliza as previsões salvas no atributo de classe que é atualizado toda vez que executamos o método *predict*, acima. As classes esperadas são passadas como parâmetro, uma vez que estamos lidando no *Perceptron* com uma aprendizagem supervisionada. Dessa forma, ao treinarmos a rede utilizamos um conjunto de dados para os quais já sabemos as classes, e ainda dividimos esse conjunto em duas partes, as quais chamamos de **conjunto de treino** e de **conjunto de teste ou validação**.

Treinamos a rede com o conjunto de treino, que deve sempre ser a maior parte de nossa partição, uma vez que é a partir dele que iremos usar o *backpropagate* para aproximar as saídas da rede às saídas esperadas contidas no conjunto. Géron (Géron, 2019) cita que tipicamente escolhemos aleatoriamente 20% dos dados como nosso conjunto de teste, ficando o restante como o conjunto de treino. A seguir, podemos calcular a acurácia da classificação do conjunto de treino, e a seguir prever e medir a acurácia do conjunto de teste para comparar os resultados.

Naturalmente a acurácia para o conjunto de teste tende a ser menor, mas não pode ser muito menor, senão dizemos que nossa rede sofre de um problema de classificação conhecido como **overfitting**, ou sobreajuste, o que significa que ela está boa para lidar com o conjunto com o qual foi treinado, o que era esperado dado que foi construída para isso, mas sofre para classificar dados novos, com os quais não foi treinada, e não é isso o que queremos.

O que queremos é justamente o contrário, que nossa rede, ou seja, nosso algoritmo de aprendizagem seja bom em **generalizar** os dados de entrada que fornecemos a ele. Anas Al-Masri (Al-Masri, 2019) define o termo generalização como a habilidade do modelo para fornecer saídas sensíveis para conjuntos de entradas que ele nunca viu antes.

Dentre as várias técnicas existentes para melhorar a generalização/prevenir o *overfitting* estão a inicialização dos pesos dos neurônios segundo uma distribuição normal de média zero, procedimento explicado anteriormente. Outra técnica é denominada de **dropout**, a qual Amar Budhiraja (Budhiraja, 2016) define como ignorar alguns neurônios, escolhidos aleatoriamente, durante o treinamento da rede, isto é, não calculamos deltas durante uma execução do *backpropagate* e nem usamos seu valor em consideração quando processamos uma entrada na rede como o *feedforward*.

É como se deliberadamente "desativássemos" alguns neurônios durante a fase de treinamento, de forma a diminuir o sobreajuste aos dados de treino, uma vez que, segundo Budhiraja (Budhiraja, 2016) esse procedimento previne que todos os neurônios se tornem dependentes um dos outros, isto é, que a derivada (da função de custo) de cada um se torne dependente das derivadas de todos os outros, criando uma mútua dependência geral para a diminuição do erro, diminuindo assim o poder individual que cada um teria se fossem ajustados de forma mais independente em função de sua contribuição para o custo total da rede.

Em nosso *Perceptron* didático optei por não implementar o procedimento de *dropout*, já que o objetivo dessa versão aqui demonstrada não é fornecer um modelo que seja utilizado em problemas reais, mas apenas didáticos. Em contrapartida essa técnica está presente e pode ser utilizada nas bibliotecas de redes neurais que usaremos na parte prática do trabalho.

3.2.5 A classe *Perceptron*

A última classe implementada, não foi baseada em um exemplo, mas criada a partir da necessidade de encapsular o comportamento da rede, para facilitar os testes do seu funcionamento que iremos realizar. Já discutimos sobre as diferentes taxas de aprendizagem para o gradiente descendente, discutimos sobre as diferentes funções de ativação que podem ser utilizadas nas redes neurais, assim como a sua topologia no que se refere apenas à quantidade de neurônios e a quantidade de camadas de neurônios.

Esses são basicamente os atributos que teremos que ajustar de acordo com a necessidade que os dados utilizados em nosso aprendizado irão criar. Dessa forma criamos a classe *Perceptron* que possui um construtor que irá lidar com a seleção dessas opções. Na listagem 3.13 está apenas a definição de seu construtor e a documentação explicativa.

```
1  class Perceptron():
2      def __init__(self, N=[1], M=50, ativacao="l_relu", taxa=0.001, debug=0):
3          '''(None, str, list[int], int, float, float, str, float) -> None
4          Construtor da minha classe Perceptron
5          Parâmetros da classe:
6              *N: quantidade de neurônios da camada oculta, podendo ser
7                  especificada um vetor de várias camadas ocultas ou apenas
8                  uma.
9              *M: quantidade de treinamentos desejada, denominado de número
10                 de "épocas" da rede, o valor padrão é50;
11             *ativacao: escolha de uma das funções de ativação disponíveis
12                 para a(s) camada(s) oculta(s).
13             *taxa: taxa de aprendizagem, padrão de 0.001
14             *debug: flag para exibição de parâmetros durante o treinamento
15         '''
16         ...
```

Programa 3.13: *Trecho da classe* Perceptron

No construtor é feita uma seleção dentre as funções de ativação existentes no script *util.py*, de forma que só precisamos passar um texto com o nome da função, que o construtor irá selecionar a função e sua derivada para posteriormente informá-las à classe *Network*. Deixamos por padrão a escolha da função de ativação *Leaky RELU*, de acordo com a orientação geral dada por Facure (Facure, 2017b).

O parâmetro *M* define a quantidade inicial padrão de **épocas** que serão treinadas. Uma época corresponde a uma passagem do conjunto de treino pelo *feedforward* e a seguir pelo *backpropagation*, ou seja, configura um único ajuste dos parâmetros através de nosso algoritmo de treinamento.

A arquitetura de camadas padrão definida pelo parâmetro *N* não tem de fato esse papel, serve apenas para uma validação existente no construtor para que não permita a passagem

de uma quantidade ≤ 0 de camadas ou de neurônios. Se quiséssemos uma arquitetura de 3 camadas com 4, 5, e 6 neurônios cada, por exemplo, então o parâmetro passado durante a criação de um objeto *Perceptron* deveria ser $N = [4, 5, 6]$.

Outra tarefa do construtor é criar um objeto da classe *OneHotEncoder*[1] da biblioteca *scikit-learn*[2]. Essa é uma das mais famosas e mais utilizadas bibliotecas da linguagem Python para tarefas de aprendizado de máquina. É a biblioteca utilizada pela maioria dos autores dos livros-textos da área de ciência de dados, como por exemplo Géron (Géron, 2019) e Grus (Grus, 2016).

Esse objeto codificador (*encoder*) é salvo como um atributo de classe: `self._enc`, e será utilizado no método de treinamento para criar automaticamente as classes numéricas a partir das classes fornecidas pelos conjuntos de treinamento e validação. Essas classes numéricas representam cada classe no formato de um vetor com todos os componentes zerados exceto um, o que identifica unicamente as classes.

Suponha, por exemplo, que estamos treinando um conjunto de fotos que possuem as classes *cachorro*, *gato* e *rato*. A função codificadora poderá transformar a palavra *cachorro* no vetor $[1, 0, 0]$, *gato* no vetor $[0, 1, 0]$ e *rato* no vetor $[0, 0, 1]$. De forma que estes serão os valores esperados para os 3 neurônios de saída que nossa rede obrigatoriamente deverá ter (número de classes = número de neurônios de saída). A ordem dessa codificação é irrelevante, sendo gerenciada internamente pela classe *OneHotEncoder*.

A seguir, no método utilizado para treinar a rede, que recebe os dados de entrada e as classes esperadas correspondentes a cada entrada, o primeiro passo é fazer essa codificação. O primeiro trecho do método `treinar` está na listagem 3.14.

```
1  def treinar(self, x_train, y_train, M=0):
2      '''(np.array, np.array, int) -> None
3      Processo de treinamento da rede neural
4      1- Tratar os dados, obtendo as classes das respostas
5      2- Treinar um número M de épocas
6      3- Armazenar no objeto o estado final da rede,
7      com os pesos e vieses ajustados pelo treinamento
8      '''
9      # onehotencoder extrai as classes únicas já ordenadas alfabeticamente
10     y_encoded = self._enc.fit_transform(y_train)
11     classes = self._enc.categories_[0]
```

Programa 3.14: *Trecho da classe* Perceptron

Nessa classe e nos exemplos subsequentes neste trabalho, sempre nomeio o conjunto dos dados de treino de `x_train` e `y_train`, e do conjunto de teste de `x_test` e `y_test`. A letra x indica que é o conjunto de dados de entrada e y indica as classes esperadas de classificação. No trecho acima usamos a classe *OneHotEncoder* para transformar quaisquer formatos que as classes forem informadas nos vetores numéricos que serão os valores esperados da saída da rede.

O próximo trecho de código irá se encarregar de criar a estrutura geral da rede, ao final

[1] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html
[2] https://scikit-learn.org/stable/index.html

criando um objeto da classe *Network* e salvando-o como um atributo de classe. É o que está presente na listagem 3.15. Os dados de entrada são esperados no formato de lista do tipo *numpy*, dessa forma a primeira linha do trecho obtém a quantidade de características (*features*) dos dados de entrada, ou seja, das variáveis explicativas do modelo, através do método `shape(1)` do tipo *numpy*, essa será a quantidade de neurônios da camada de entrada da rede, um para cada característica.

Usando o exemplo das fotos de animais, os pixels da foto seriam as variáveis explicativas para um modelo de classificação, assumindo que todas as fotos possuem a mesma quantidade de pixels e que cada pixel possui apenas um valor de intensidade de cinza, ou outra cor única qualquer. Se utilizássemos fotos coloridas, com por exemplo, as intensidades de 3 cores diferentes por pixel, então o número de variáveis explicativas de nosso modelo seria $3n$, sendo n o número de pixels da foto.

```
1  neurons_in = x_train.shape[1]
2
3  if self.network is None:
4      neurons_out = len(self.classes)
5      for i in range(len(self.N)):
6          if len(self.N) == 1 and self.N[0] < neurons_out:
7              self.N[0] = min(int(np.ceil(neurons_in*2/3 + neurons_out)), neurons_in)
8
9      rede = []
10     rede.append(neurons_in)
11     for hidden in self.N:
12         rede.append(hidden)
13     rede.append(neurons_out)
14
15     ativacoes = (self.ativacao, self.der_ativacao,
16                  self.ativacao_saida, self.der_ativacao_saida)
17
18     self.network = Network(np.array(rede), self.taxa, ativacoes)
```

Programa 3.15: *Trecho da classe* Perceptron

A seguir, o método irá criar a estrutura da rede, se essa é a primeira vez que o objeto estiver sendo utilizado para o treinamento, do contrário ele irá realizar outras M épocas de treinamento, a partir dos dados existentes na rede.

Obtém-se a quantidade de neurônios para a camada de saída a partir da quantidade de classes identificadas. A seguir ele utiliza as quantidades de neurônios para as camadas ocultas se estas foram previamente informadas manualmente durante a criação da classe, ou então, é feito um cálculo, para que seja utilizada uma quantidade apropriada de neurônios para a primeira camada oculta, que pode ser a única camada oculta por padrão.

Essa quantidade apropriada é definida por Jeff Heaton (Heaton, 2017) como sendo 2/3 da quantidade de neurônios de entrada mais a quantidade de neurônios de saída. Esta é uma das 'regras de ouro' que ele descobriu empiricamente pois se mostraram as regras mais gerais para garantir o bom funcionamento da rede em vários casos. Outra regrinha que ele encontrou, que é mais geral mas que incluí esta é que a quantidade de neurônios de uma camada oculta única deve ser tal que seja menor que a quantidade de neurônios de entrada mas maior que a quantidade de neurônios de saída.

Naturalmente o treinamento aceita quaisquer número de camadas ocultas e de neurônios em cada camada, apenas não há garantias de que o treinamento irá suceder sem ocorrer o problema do gradiente explodindo/desaparecendo. Mesmo a utilização das regrinhas de Heaton (Heaton, 2017) não asseguram esse sucesso do treinamento, apenas testes com outras quantidades de neurônios e camadas, e com outras funções de ativação e taxas de aprendizado que poderão resultar eventualmente num treinamento bem sucedido, caso essas opções-padrão não sejam suficientes.

Essa é uma dificuldade inerente das redes neurais, ainda mais quando tenta lidar com conjuntos de dados muito grandes e com um grande número de variáveis explicativas. Essa é uma das razões principais para ser preferível a utilização de uma biblioteca já consolidade e com muitos anos de desenvolvimento e ajustes por muitos desenvolvedores e cientistas de dados ao redor do mundo.

O próximo trecho, na listagem 3.16 é o trecho principal deste método, é o treinamento *backpropagate* feito um número M de épocas. São passados como parâmetros os dados de entrada, e as classes esperadas já codificadas. Ao final desse treinamento, a rede está salva no atributo de classe `self.network` com os parâmetros já ajustados e prontos para serem utilizados para validação e previsão de novos dados.

```
1 for _ in self.tqdm(range(self.M)):
2     self.network.train(x_train, y_encoded)
```

Programa 3.16: *Trecho da classe* Perceptron

O próximo método realiza a previsão das classes a partir dos dados informados, simplesmente utilizando a função da classe *Network* criada para isso. É o que está na listagem 3.17, abaixo.

```
1 def prever(self, X, interpretar=None):
2     '''(np.array, Callable) -> np.array'''
3     ...
4     if interpretar is None:
5         return self.network.predict(X, self.reinterpretar_saidas)
6     return self.network.predict(X, interpretar)
```

Programa 3.17: *Trecho da classe* Perceptron

Por padrão a função que irá interpretar os neurônios de saída, convertendo-os em uma classe, foi criada da forma como será mostrada na listagem 3.18, a seguir.

```
1 def reinterpretar_saidas(self, saidas):
2     '''(array) -> np.array
3     '''
4     maximo = max(saidas)
5     saida = np.array([int(x == maximo) for x in saidas])
6     return self._enc.inverse_transform(saida.reshape(1, -1))
```

Programa 3.18: *Trecho da classe* Perceptron

Essa função realiza a interpretação padrão dos neurônios de saída, primeiramente eles são convertidos em vetores com identificação única, ou seja, no formato $[0, \ldots, 0, 1, 0, \ldots, 0]$,

sendo que a posição que irá receber 1 é aquela que tiver originalmente o valor máximo, ou seja, mais distante de 0, e o restante será convertido em zeros. Dessa forma, basta utilizarmos a decodificação inversa do objeto *OneHotEncoder*, que está salvo no atributo de classe, e daí obtemos qual a classe mais provável à qual pertence o dado de entrada que foi processado pela rede.

Isto nos mostra que uma possível interpretação dos neurônios de saída é que cada um possui uma probabilidade de que o dado pertença àquela classe indexada na mesma posição a qual esse neurônio está na camada de saída. Se usarmos como exemplo nosso modelo fictício dos animais, ao processar uma foto, a rede iria devolver os valores dos neurônios de saída, por exemplo, como o seguinte vetor: $[0.002, 0.976, 0.013]$. Pode-se dizer que há uma probabilidade maior de que essa foto pertença à segunda classe, que digamos ser a classe *Gatos*, por exemplo.

O que a função `reinterpretar_saidas` faz é converter esse vetor de saídas da rede no vetor $[0, 1, 0]$, que agora está no formato das classes numéricas geradas por nosso objeto codificador. Dessa forma, executar uma decoficação com esse mesmo objeto, que havia originalmente codificado a palavra *Gato* no vetor $[0, 1, 0]$, irá fazer a operação inversa, retornando a palavra *Gato* como sendo a classe mais provável para a foto processada.

Alternativamente podemos utilizar outra função de interpretação dos dados de saída, devendo ser informada diretamente por referência para o método `prever`.

Por razões didáticas, ou mesmo em casos em que não queremos obter classes diretamente a partir dos neurônios de saída, mas queremos observar diretamente os valores calculados pela rede sem interpretação, criei um método que realiza apenas o *feedforward* para uma lista de entradas fornecidas como parâmetro. É o que vemos na listagem 3.19, a seguir.

```
1 def processar(self, X):
2     '''(np.array) -> np.array'''
3     ...
4     saidas = []
5     for x in X:
6         saidas.append(self.network.feedforward(x))
7     return np.array(saidas)
```

Programa 3.19: *Trecho da classe* Perceptron

Por fim, podemos querer observar qual o erro, ou custo, que nossa rede está produzindo para um dado par de conjuntos de entrada e saídas esperadas. Para isso criei o método `funcao_erro` exibido na listagem 3.20, a seguir.

```
1 def funcao_erro(self, X, Y):
2     '''(np.array, np.array) -> float'''
3     ...
4     y_encoded = self._enc.fit_transform(Y)
5     return self.network.mse(X, y_encoded)
```

Programa 3.20: *Trecho da classe* Perceptron

Ele utiliza a função de erro que está implementada na classe *Network*, que é a função de custo que utilizamos como base para a criação de nosso algoritmo de otimização, o erro quadrático médio (MSE), também conhecido como a norma euclidiana do vetor distância entre os vetores das saídas esperadas e o das saídas obtidas pela rede.

Em todos os trechos da classe *Perceptron* acima, várias linhas estão ocultas sob o símbolo de reticências, são linhas que fazem verificações dos dados utilizados e do estado atual do objeto, se ele pode ser usado para previsão por exemplo, ou seja, se já foi treinado previamente, entre outras verificações gerais para um bom funcionamento.

3.3 Exemplo de utilização do *perceptron*

Para demonstrar a utilização da versão aqui implementada da rede *perceptron*, resolvi utilizar aquela que é considerada a base de dados de entrada no mundo da ciência de dados, a base MNIST (*Modified National Institute of Standards and Technology*) de números escritos à mão compilada originalmente pela Universidade de Nova York[3].

A versão oficial da base de dados consistem em 60 mil imagens de dimensões 28 × 28 pixels. Cada imagem é uma foto de um dígito manuscrito entre 0 e 9, sendo que a proporção de cada dígito é aproximadamente de um décimo, além disso a base contém a informação dos valores nominais de cada número, o que torna essa base de dados muito útil para validar modelos de aprendizado supervisionado, antes que sejam utilizados em casos reais. Exemplos de fotos estão na figura 3.8.

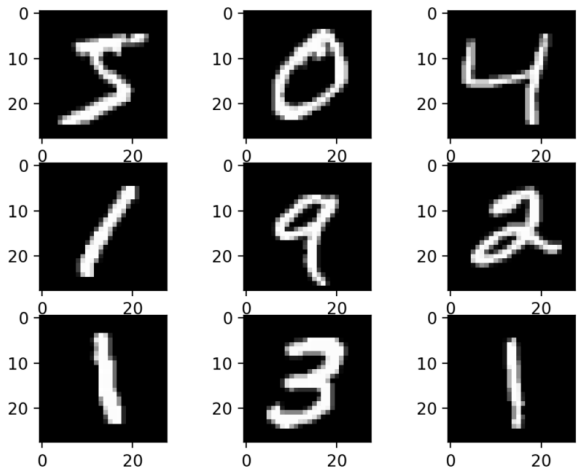

Figura 3.8: *Exemplos de fotos da base de dados MNIST de números manuscritos.*[a]

[a]Extraído de https://3qeqpr26caki16dnhd19sv6by6v-wpengine.netdna-ssl.com/wp-content/uploads/2019/02/Plot-of-a-Subset-of-Images-from-the-MNIST-Dataset-1024x768.png

É exatamente esse o objetivo aqui, testar nossa implementação didática do *perceptron*. Para isso, o primeiro passo foi obter uma versão mais *leve* da base de dados, com as fotos redimensionadas para 8 × 8 pixels, o que implica em 64 variáveis explicativas e mesmo número de neurônios de entrada. Além disso essa base possui apenas 1.800 fotos

[3]Disponível originalmente em http://yann.lecun.com/exdb/mnist/.

etiquetadas. A base original de $28^2 = 784$ pixels já se mostrou grande demais para essa implementação conseguir lidar em tempo hábil.[4]

A base foi importada através da biblioteca *sklearn*, que já possui opções para fazer download automático para o programa em execução de várias versões da base MNIST. O código da listagem 3.21 mostra a importação e também o próximo passo, que é separar a base nos conjuntos de treino, com 85% dos dados, e de validação com os 15% restantes.

```
# obtendo o conjunto de imagens de numeros escritos
from sklearn.datasets import load_digits
mnist = load_digits()

# dividindo a base nos conjuntos de treino e de teste
N = int(mnist.data.shape[0]*0.8)
x_train, y_train = mnist.data[:N], mnist.target[:N].astype(np.uint8)
x_test, y_test = mnist.data[N:], mnist.target[N:].astype(np.uint8)
```

Programa 3.21: *Trecho do script* mnist_test.py

A implementação está tão fácil de utilizar, que para a tarefa de treinar a classificação dos números digitados, não precisamos de mais de meia dúzia de linhas de código, mostradas na listagem 3.22.

```
perceptron = Perceptron(taxa=0.001, ativacao="elu", N=[48, 24])
perceptron.treinar(x_train, y_train, M=20)

y_train_pred = perceptron.prever(x_train)
score = Scores(y_train, y_train_pred)
score.exibir_grafico("Dados de treino")

y_test_pred = perceptron.prever(x_test)
score = Scores(y_test, y_test_pred)
score.exibir_grafico("Dados de teste")
```

Programa 3.22: *Trecho do script* mnist_test.py

Basicamente, uma rede é criada com a estrutura de uma camada de entrada com 64 neurônios, quantidade obtida automaticamente pela classe a partir das dimensões da lista de dados informada (*y_train*), duas camadas ocultas com 48 e 24 neurônios cada, com taxa de aprendizado 0.001 e a função de ativação *ELU*, ambos ajustados empiricamente sendo o que se saíram melhores em acurácia. A segunda linha realiza o treinamento com 20 épocas, informando a lista de imagens e a lista das classificações conhecidas.

As duas últimas linhas dessa listagem fazem a comparação dos valores previstos com os valores já conhecidos da classificação, para conhecermos a acurácia de nosso modelo de aprendizado. Abaixo nas figuras 3.9 e 3.10 está uma exibição gráfica com a função de erro MSE, a acurácia e a **matriz de confusão**, a partir da qual podemos calcular a acurácia de nossa rede neural.

A matriz de confusão permite relacionar as classes esperadas com as classes previstas de um conjunto de dados utilizados num algoritmo de aprendizagem supervisionada como

[4]Em tentativas que fiz com a base original, meu computador ficou calculando deltas por quase 2 horas sem completar uma só época de treinamento.

é o caso do *perceptron*. Ela mostra as contagens dessas relações, e dessa forma, a diagonal dessa matriz possui as classificações corretamente obtidas pelo algoritmo, e o restante da matriz a contagem das classificações incorretas.

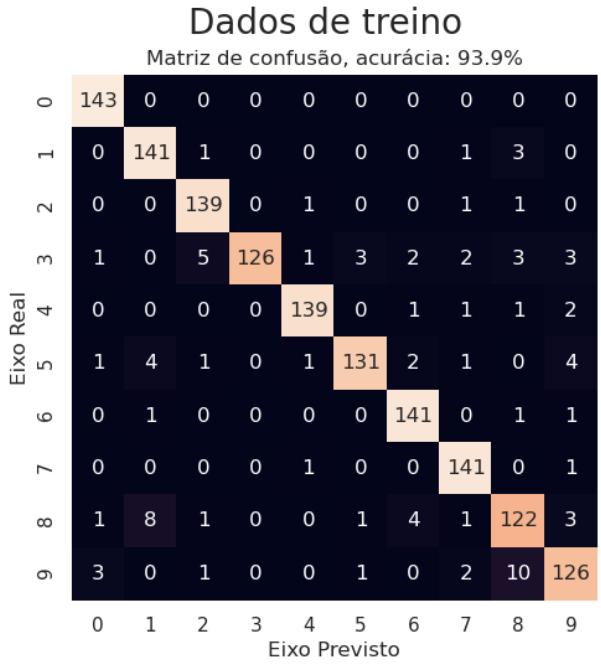

Figura 3.9: *Matriz de confusão e acurácia do conjunto de treino da base MNIST* 8 × 8 *pixels.*

Como os pesos da rede são inicializados aleatoriamente, cada treinamento pode obter resultados levemente diferentes, embora que, no geral os resultados irão convergir para um resultado médio, que dependem dos parâmetros utilizados, como a função de ativação, quantidade de camadas e de neurônios e de épocas de treinamento.

Podemos notar que o conjunto de treino possui uma boa perfomance, obtendo mais de 93.9% de acurácia, ou seja, de classificações corretas. Por outro lado, o conjunto de treino obteve pouco mais de 83.6%, o que indica o problema de *overfitting* em nossa rede, quando a classificação do treino está boa, o que é esperado dado que é a base utilizada para o ajuste dos parâmetros internos, mas quando a rede treinada tenta lidar com dados inéditos, o que é o papel do conjunto de teste, se sai consideravelmente pior.

Está demonstrada a necessidade da utilização de alguma implementação mais robusta, e refinada com anos de contribuições da comunidade de programadores e cientistas de dados ao redor do planeta. E essa implementação, ou pelo menos uma delas, é a API Keras, que veremos na próxima seção, e que será utilizada no restante desse trabalho.

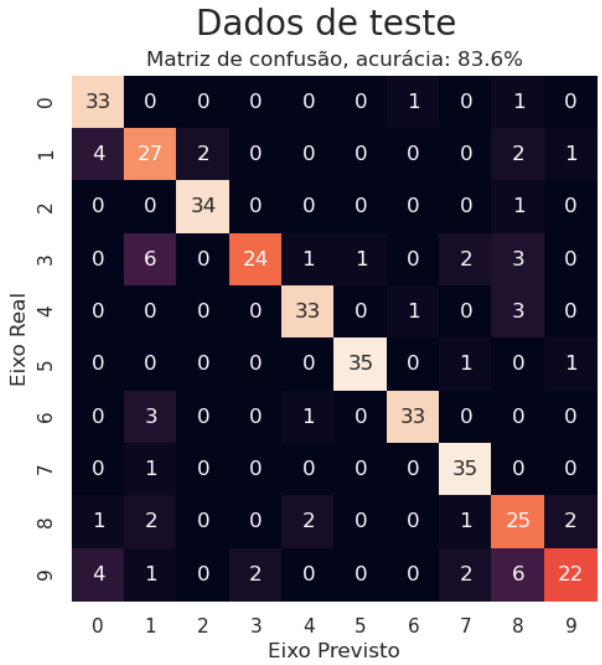

Figura 3.10: *Matriz de confusão e acurácia do conjunto de teste da base MNIST* 8 × 8 *pixels.*

3.4 Utilizando a API Keras

De acordo com seu site oficial[5], *Keras* é uma API (Interface de Programação de Aplicativos) de *deep learning* escrita em Python, e que roda sobre a plataforma de *deep learning* chamada de *TensorFlow*[6] que é de fato a biblioteca que devemos instalar em nosso ambiente Python, para podermos utilizar as redes neurais ali implementadas e quaisquer outros recursos da API *Keras* em nosso projeto de aprendizado.

Keras implementa quase todas as arquiteturas de redes neurais, segundo Géron (Géron, 2019), sua popularidade é devido sua facilidade de uso, flexibilidade aliadas a um lindo design de software. Existem algumas implementações da API, como a *TensorFlow*, que é a principal, a *Microsoft Cognitive Toolkit*, a *Apache MXNet*, a *Apple's Core ML*, etc.

Todas essas implementações podem ser utilizadas em conjunto, na biblioteca conhecida como *multibackend Keras*, sendo que a escolha entre uma delas ocorre de forma encapsulada até mesmo para o cientista de dados. Alternativamente, utilizar a versão própria presente na biblioteca *TensorFlow* traz benefícios como recursos exclusivos à ela. O funcionamento dessas 2 implementações está na Figura 3.11, a seguir.

Para entender o funcionamento e uso do Keras, criei uma rede com a mesma estrutura do Perceptron aqui implementado, mas com algumas melhorias já inerentes à API, com objetivo de classificar o mesmo conjunto de dados MNIST, para comparar a eficiência. O primeiro passo é importar a base de dados, sendo a mesma que usamos anteriormente, então reutilizamos o mesmo trecho de código mostrado na listagem 3.21.

[5] https://keras.io/about/
[6] https://www.tensorflow.org/

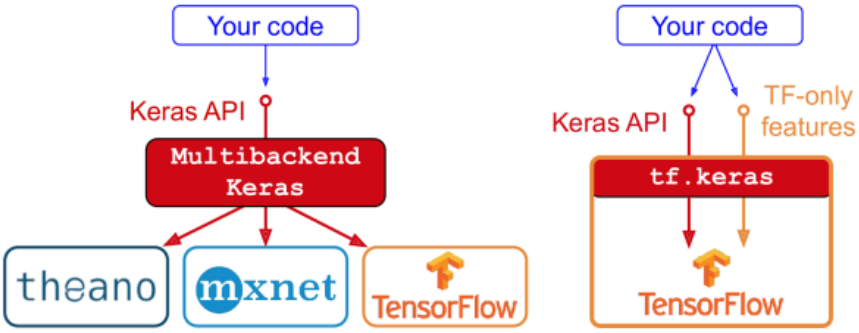

Figura 3.11: *As 2 implementações da API Keras.* Multibackend *à esquerda e* TensorFlow *à direita.*[a]

[a]Extraído de: Aurélien Géron, 'Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow', O'Reilly 2.a edição 2019, página 385.

O próximo passo é criar a rede utilizando a API Keras, o que vemos na listagem 3.23. A primeira linha cria uma rede sequencial, isto é, uma rede *feedforward*. A seguir, são adicionadas as camadas, a primeira camada é definida como *Flatten* pois é a camada de entrada, então ela não aplica nenhuma transformação nos dados.

```
1 model = tf.keras.Sequential()
2 layers = tf.keras.layers
3 model.add(layers.Flatten())
4 model.add(layers.Dense(48, activation='elu'))
5 model.add(layers.Dense(24, activation='elu'))
6 model.add(layers.Dense(10, activation='softmax'))
```

Programa 3.23: *Trecho do script* mnist_keras.py

As próximas camadas são adicionadas com o tipo *Dense*, o que significa uma conexão de todos os neurônios de uma camada com todos da próxima, dessa forma estamos criando um *Perceptron* exatamente como aquele implementado.

As camadas ocultas utilizam a função de ativação *ELU*, e a camada de saída utiliza a função *softmax*, o que na API Keras significa que estamos classificando os dados de acordo com o valor máximo dos neurônios de saída, o que em nossa implementação foi papel da função `reinterpretar_saidas`, que foi definida na listagem 3.18.

```
1 model.compile(optimizer=keras.optimizers.SGD(lr=0.001),
2                 loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Programa 3.24: *Trecho do script* mnist_keras.py

Na listagem 3.24, definimos os parâmetros finais da rede. Define-se a função de otimização *SGD* (*Stocastic Gradient Descent*), quase idêntico à implementação do gradiente descendente implementada, mas ao invés de utilizar todas as imagens de treino numa época de treinamento, algumas são escolhidas aleatoriamente, e isso é feito um certo número de vezes, e os deltas são escolhidos para o conjunto aleatório que tenha se saído melhor, de acordo com a função de perda e métrica utilizada.

A função de perda é escolhida com a opção *sparse_categorical_crossentropy*, pois é a opção padrão do Keras para tarefas de classificação, não sendo possível utilizar a função MSE como antes, por limitações de projeto da API, o que significa que ela irá tratar as categorias de dados da forma que ela foi obtida, com as 10 classes de 0 a 9, onde cada imagem pertence a apenas uma dessas classes.

Esse parâmetro seria diferente se fosse criada uma rede para classificação binária, por exemplo, ou então se nossos vetores `y_train` e `y_test` tivessem sido codificados em vetores do tipo $[0, \ldots, 1, 0, \ldots]$ da forma que foi feita na nossa implementação.

Por fim a acurácia é escolhida como a métrica para avaliação da rede, e que será utilizada pelo gradiente estocástico para definir o melhor sub-conjunto de treino durante uma dada época de treinamento. Basta treinar a rede como o método *fit*, especificando o número de épocas de treinamento desejado, neste caso 20 foram suficientes, o que está feito na listagem 3.25.

```
1 model.fit(x_train, y_train, epochs=20)
2 # avaliando dados de treinamento
3 model.evaluate(x_train, y_train, verbose=2)
4 # avaliando os dados de teste
5 model.evaluate(x_test, y_test, verbose=2)
```

Programa 3.25: *Trecho do script* mnist_keras.py

Nessa listagem também está a avaliação da rede, para o conjunto de treino e para o conjunto de teste. Os resultados obtidos após 20 épocas de treinamento foram 99.0% de acurácia para o conjunto de treino e 89.7% para o conjunto de teste.

Por fim podemos fazer previsões com a rede treinada, nesse caso a API possui o método *predict* que irá retornar os valores da camada de saída, ou seja, as probabilidades de cada imagem pertencer à uma das 10 classes informadas durante o treinamento.

Nas últimas linhas da listagem 3.25, está a lógica de obter a classificação predita para o conjunto de teste, utiliza a função *argmax* para obter a classe que obteve a probabilidade máxima. E por fim exibe a matriz de confusão dessa predição, o que está na Figura 3.12.

Vemos que a API produz melhores resultados, levando em conta a mesma base de dados, a mesma arquitetura de rede e o mesmo algoritmo de treinamento. A diferença será possivelmente devida a alguma estratégia interna de implementação, a princípio oculta aos usuários, que faz com que os resultados sejam melhores.

Mesmo assim, como não utilizamos diretamente outros recursos de melhoria disponíveis, ainda é possível notar o *overfitting*, dado que a acurácia no conjunto de teste permanece menor que a acurácia no conjunto de treino.

A seguir, faço um teste com a versão oficial da base de dados, de dimensão maior de pixels (28×28), o que é possível já que a biblioteca está implementada com muita eficiência e funciona bem para bases maiores mesmo num ambiente pessoal de computação. O código está na íntegra na listagem 3.26.

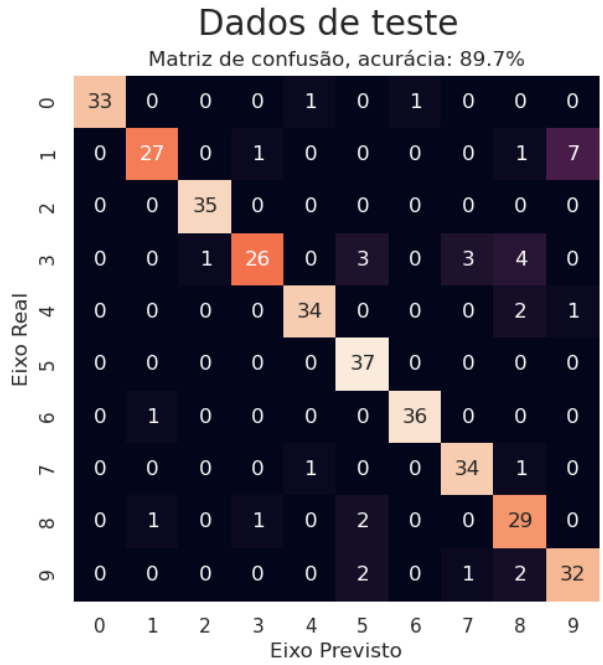

Figura 3.12: *Matriz de confusão, função de perda e acurácia do conjunto de teste da base MNIST* 8×8 *pixels, utilizando a API Keras.*

```
mnist = fetch_openml('mnist_784', version=1) # versao 28x28

model = tf.keras.Sequential()
layers = tf.keras.layers

model.add(layers.Flatten(input_shape=(28, 28)))
model.add(layers.Dense(512, activation='elu'))
model.add(layers.Dense(256, activation='elu'))
model.add(layers.Dense(128, activation='elu'))
model.add(layers.Dense(10, activation='softmax'))

model.compile(optimizer=keras.optimizers.SGD(lr=0.001),
              loss='sparse_categorical_crossentropy', metrics=['accuracy'])

model.fit(x_train, y_train, epochs=15)

model.evaluate(x_train, y_train, verbose=2)
model.evaluate(x_test, y_test, verbose=2)

y_pred = np.argmax(model.predict(x), axis=-1)

# usando a minha propria classe de validação
score = Scores(y_test, y_pred)
score.exibir_grafico("Dados de teste")
```

Programa 3.26: *Trecho do script* mnist_keras.py

A importação da base é feita agora usando outra função da biblioteca *sklearn*. A seguir, os dados são divididos entre os conjuntos de treino e de teste, com as listas `x_train`,

`x_test` e `y_train, y_test` contendo os pixels e as classficações respectivamente, como antes.

Dessa vez, a rede é criada de acordo com o tamanho da base utilizada, como a camada de entrada é bem maior, já que são 28 × 28 = 784 pixels. Crio 3 camadas ocultas, com os valores progressivamente menores, mas usando ainda a mesma estrutura densa e com a camada final novamente com 10 neurônios representando os diferentes algarismos.

Por fim, obtemos da mesma forma que no caso anterior, a classificação predita para o conjunto de teste. Dessa vez os resultados são 99.9% de acurácia no conjunto de treino e 96.4% de acurácia no conjunto de teste, conforme pode ser visto na figura 3.13, junto com a matriz de confusão.

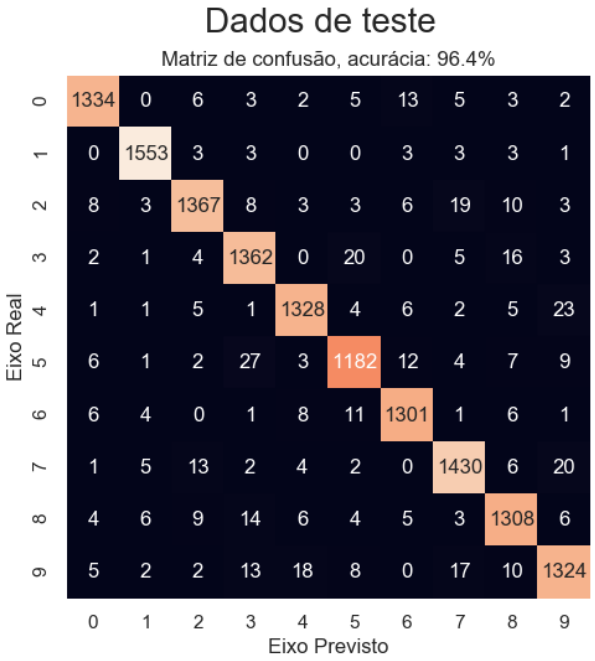

Figura 3.13: *Matriz de confusão, função de perda e acurácia do conjunto de teste da base MNIST 28 × 28 pixels, utilizando a API Keras.*

É notável a eficiência da API em comparação à nossa implementação simples. A rede é treinada em menos de 2 minutos, mesmo sendo utilizada a base MNIST original que possui muito mais pixels e também muito mais imagens, 60 mil em comparação às 2 mil da base menor utilizada anteriormente.

Além disso a acurácia final do conjunto de treino é praticamente perfeita, e o conjunto de treino, apesar de muito melhor em relação à implementação, ainda está relativamente menor à acurácia do treino, indicando que nessa versão imitada do *Perceptron* existe o problema de *overfitting*, mesmo que em menor intensidade.

É para resolver esse problema e outros, que existem diversas otimizações e configurações alternativas ao simples Perceptron, implementadas na API Keras. Existem várias outros métodos de otimização, outras funções de perda e sobretudo outras arquiteturas de rede que podem ser utilizadas em alternativa às camadas densas do perceptron.

A estratégia utilizada no contexto de *deep-learning* é sobretudo de tentativa e erro, e conforme um cientista de dados vai fazendo isso muitas vezes vai construindo conhecimentos sobre qual arquitetura usar para um problema, qual otimização funciona melhor, uma vez que isso irá sempre variar de acordo com a base de dados utilizada, seja para classificação seja para regressão.

Uma tentativa recente de melhora nessa estratégia foi a criação de uma biblioteca cujo objetivo é o de justamente testar entre diversas arquiteturas e demais parâmetros de criação de uma rede neural, qual a que se sairá melhor para um dado conjunto de dados e objetivo de aprendizagem. Essa biblioteca é chamada de *AutoKeras* e pertence à essa nova vertente de *deep-learning* conhecida como AutoML (*Automated Machine Learning*).

Os objetivos da AutoML é, segundo Andre Ye (Ye, 2020), é o de tornar *deep-learning* mais acessível para o aprendizado de todos os entusiastas, e também o de acelerar o desenvolvimento de pesquisas nessa área, para tornar a criação de conhecimentos mais sólidos e gerar mais *insigths* para a interpretabilidade dos modelos de redes neurais.

Em seu artigo, Andre Ye (Ye, 2020) demostra como instalar e utilizar o *AutoKeras*. Resumidamente, o que fazemos é informá-la com a nossa base de dados, e ela irá fazer todo o trabalho de escolha da arquitetura e dos parâmetros, e ao final retornará um objeto da API Keras, que poderá ser usado tanto para tarefas de classificação quanto de regressão.

Uma desvantagem a ser considerada é o tempo de processamento da biblioteca *AutoKeras*. Dessa forma, para problemas de menor escala, tanto de quantidade de dados utilizada quanto de poder computacional disponível, o que inclui o que iremos fazer a seguir na previsão de séries temporais, é mais viável testar dentre alguns poucos parâmetros conhecidamente melhores a partir da experiência de outros cientistas de dados, o que implica num tempo consideravelmente menor de processamento.

Capítulo 4

Séries temporais

Neste capítulo são apresentados os conceitos básicos de séries temporais. Também são discutidos brevemente os modelos tradicionais de análise e descrição das séries, ou de previsões de valores futuros, de acordo com o objetivo do estudo. Tais modelos são, por exemplo, baseados em médias móveis, tendências e sazonalidades presentes nos dados.

De acordo com Pedro A. Morettin e Clélia M C. Toloi (Morettin e Castro Toloi, 2019), uma série temporal é qualquer conjunto de observações ordenadas no tempo. São exemplos: valores diários de poluição de uma cidade, valores mensais de temperatura, índices diários da bolsa de valores, número médio anual de manchas solares e registro de marés em portos e estuários.

A análise e predição de cotações de moedas estrangeiras, índices de ações, entre outros dados econômicos, constituem uma área essencial da economia e que exige a avaliação de um número enorme de fatores, muitos dos quais de características humanas e portanto imprevisíveis em sua exatidão, sendo descritos como exemplos de fenômenos *estocásticos*, isto é, probabilísticos.

As séries temporais podem ser **contínuas** em função do tempo, como é o exemplo de registros de marés, ou **discretas** como são todos os outros exemplos, ou seja, os valores são tomados a N intervalos regulares de um período T considerado, tal que $N = T/\Delta t$. Na prática, para o uso em modelos, segundo explica Morettin e Toloi (Morettin e Castro Toloi, 2019), as séries contínuas devem ser discretizadas em intervalos, uma vez que é esse tipo de dado que poderá ser processado num computador.

Pode-se classificar a análise das séries temporais de acordo com seu objetivo de estudo. Morettin e Toloi (Morettin e Castro Toloi, 2019) listam os alguns objetivos principais:

- Investigar o mecanismo gerador da série temporal, procurando descrevê-la a partir de uma função teórica.
- Fazer previsões de valores futuros da série, seja tanto a curto quanto a longo prazo.
- Descrição da série, em termos de tendências, variações sazonais, ou então análises descritivas por meio de histogramas, médias móveis, etc.

- Procurar por periodicidades relevantes nos dados, quando não fazemos suposições de periodicidades comuns como semanais, mensais ou anuais, por exemplo.

Neste contexto, Morettin e Toloi (Morettin e Castro Toloi, 2019) definem que um modelo é uma descrição probabilística de uma série temporal, cabendo ao cientista de dados decidir a melhor utilização desse modelo segundo seus objetivos.

Além disso eles afirmam que qualquer tarefa de previsão, sendo este o objetivo, será baseada em algum procedimento computacional que calcula uma estimativa do futuro baseada na otimização de uma função de perda calculada sobre combinações lineares de valores do passado. É a união de um modelo probabilístico com a otimização de uma função de perda que define um **método de previsão**.

Há dois tipos básicos de modelos que lidam com séries temporais, de acordo com Morettin e Toloi (Morettin e Castro Toloi, 2019), que são os modelos **paramétricos**, onde a análise é feita no *domínio temporal* com suposições e parâmetros a serem estimados, e os modelos **não-paramétricos**, em que a análise é feita no *domínio das frequências* com um enfoque mais descritivo e sem muitas suposições.

Dentre os modelos paramétricos, destacam-se os modelos *ARIMA* (*autorregressivos integrados de médias móveis*), disponíveis como bibliotecas das linguagens *Python* e *R*, enquanto que entre os modelos não-paramétricos destaca-se a *análise espectral*, também denominada de *análise de Fourier*, já que as ferramentas utilizadas são as transformadas de Fourier e suas variações.

Na última seção desse capítulo, uma estrutura de uma rede neural recorrente, implementada com a API Keras, é proposta como um modelo **semiparamétrico**, ou seja, estimando parâmetros inerentes às redes neurais, mas não utilizando parâmetros ou suposições específicas sobre os dados, para a realização de previsões de valores futuros de séries temporais financeiras, com enfoque nas séries históricas de cotações de moedas estrangeiras.[1].

4.1 Processos estocásticos

De acordo com Morettin e Toloi (Morettin e Castro Toloi, 2019), um **processo estocástico** é uma familia $Z = \{Z(t), t \in \mathcal{T}\}$, onde o conjunto $\mathcal{T}$ é normalmente tomado como $\mathcal{T} = \mathbb{Z}$ ou $\mathcal{T} = \mathbb{R}$, tal que, para cada $t \in \mathcal{T}$, $Z(t)$ é uma variável aleatória (v.a.).

Portanto, um processo estocástico é uma família de variáveis aleatórias reais $Z(t),\ t \in \mathcal{T}$, definidas num mesmo espaço de probabilidades $(\Omega, \mathcal{A}, \mathcal{P})$, e portanto $Z(t)$ é uma função de dois argumentos, isto é, $Z(t, \omega),\ t \in \mathcal{T},\ \omega \in \Omega$.

Para cada $t \in \mathcal{T}$ fixado, $Z(t, \omega)$ será uma v.a. com uma distribuição de probabilidades. Pode haver uma função de densidade de probabilidade diferente para cada $t \in \mathcal{T}$, mas normalmente assume-se que é a mesma, conforme anotado por Morettin e Toloi (Morettin e Castro Toloi, 2019).

[1] Tais séries estão disponíveis para *download* no site do Banco Central do Brasil: https://www.bcb.gov.br/

Por outro lado, para cada $\omega_i \in \Omega$ fixado, denotamos $Z(t, \omega_i)$ por $Z^{(i)}(t)$ como uma função de t, denominada de **trajetória** do processo, ou simplesmente de *série temporal*. Isto define uma série temporal como uma trajetória ou realização de um processo estocástico, o que pode ser melhor ilustrado pela Figura 4.1, abaixo.

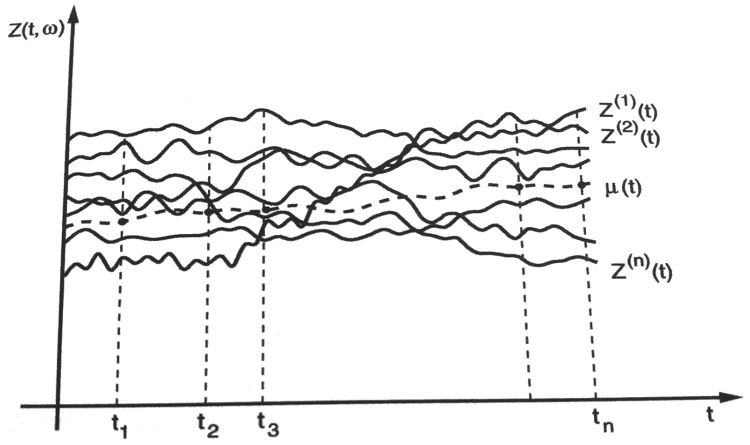

Figura 4.1: *Processo estocástico como uma família de trajetórias, isto é, de séries temporais.*[a]

[a]Extraído de Morettin e Toloi, 2019, pág 27.

Estaremos interessados em processos univariados tanto em $\mathcal{T}$ quanto em Ω, isto é, séries temporais de apenas um argumento temporal, e com um evento $\omega \in \Omega$ fixado e portanto omitido, o que simplifica a notação de $\{Z^{(i)}(t), t \in \mathcal{T}\}$ para $\{Z(t), t \in \mathcal{T}\}$, o que definem as *séries temporais univariadas*, que descrevem como uma v.a. real evolui no domínio temporal $\mathcal{T}$.

Além disso, para as séries e modelos aqui tratados iremos restringir $\mathcal{T} = \mathbb{Z}$, assim omitiremos a definição de um domínio geral e fixaremos a notação como sendo $\{Z(t), t \in \mathbb{Z}\}$ ou ainda $\{Z_t, t \in \mathbb{Z}\}$, para denotar as *séries temporais discretas univariadas*, que além de discretas são equiespaçadas no tempo. A partir de agora, serão chamadas simplesmente de **séries temporais**, pois são os únicos processos estocásticos de interesse neste trabalho.

4.1.1 Definições

Seguem algumas definições que nos levarão a classes específicas de processos estocásticos, com as quais lidaremos daqui em frente. Sejam $t_1, \dots, t_n$ elementos quaisquer de $\mathcal{T}$, daí se conhecermos as **distribuições finito-dimensionais** de Z dadas por:

$$F(z_1, \dots, z_n; t_1, \dots, t_n) = P\{Z_{t_1} \leq z_1, \dots, Z_{t_n} \leq z_n\} \tag{4.1}$$

Teremos então que o processo estocástico $Z = \{Z_t, t \in \mathcal{T}\}$ estará especificado, para todo $n \geq 1$. Tais funções de distribuição devem, de acordo com Morettin e Toloi (Morettin e Castro Toloi, 2019) satisfazer as condições:

(i) (Simetria) Para qualquer permutação $j_1, \dots, j_n$ dos índices $1, \dots, n$:

$$F(z_{j_1}, \dots, z_{j_n}; t_{j_1}, \dots, t_{j_n}) = F(z_1, \dots, z_n; t_1, \dots, t_n)$$

(ii) (Compatibilidade) Para $m < n$:

$$\lim_{z_{m+1} \to \infty, \ldots, z_n \to \infty} F(z_1, \ldots, z_m, z_{m+1}, \ldots, z_n; t_1, \ldots, t_n) = F(z_1, \ldots, z_m; t_1, \ldots, t_m)$$

Segundo Morettin e Toloi (Morettin e Castro Toloi, 2019), pode-se demonstrar que qualquer conjunto de funções de distribuição da forma (4.1) satisfazendo as duas condições acima define um processo estocástico Z sobre $\mathcal{T}$.

Em termos práticos, não se conhecem as funções de distribuição finito-dimensionais de um processo Z sobre $\mathcal{T}$. Assim a abordagem mais utilizada, conforme Morettin e Toloi (Morettin e Castro Toloi, 2019) é tentar determinar os momentos, principalmente os de primeira e segunda ordem, das v.a. $Z_{t_1}, \ldots, Z_{t_n}$.

O momento de primeira ordem, isto é, a **média** de Z é definida por:

$$\mu(1; t) = \mu(t) = \mathrm{E}\{Z(t)\} = \int_{-\infty}^{\infty} z f(z; t) dz, t \in \mathcal{T} \tag{4.2}$$

Define-se, a partir dos momentos de primeira ordem, a **função de autocovariância** (facv) de Z:

$$\gamma(t_1, t_2) = \mu(1, 1; t_1, t_2) - \mu(1; t_1)\mu(1; t_2) = \mathrm{Cov}\{Z(t_1), Z(t_2)\} \tag{4.3}$$

Particularmente, quando $t = t_1 = t_2$, define-se a função **variância** de Z, configurando um momento de segunda ordem, por:

$$\gamma(t, t) = \mathrm{Var}\{Z(t)\} = \mathrm{E}\{Z^2(t)\} - \mathrm{E}^2\{Z(t)\} \tag{4.4}$$

4.1.2 Processos estacionários

Um processo Z é dito **estacionário** se suas características para qualquer tempo não dependem da escolha da origem do domínio temporal, isto é, as características de $Z(t + \tau)$ para todo τ, são as mesmas de $Z(t)$. Morettin e Toloi (Morettin e Castro Toloi, 2019) nomeia o parâmetro τ de "**lag**"[2], e dá como exemplo de processo estacionário as medidas de vibrações de um avião em regime estável de vôo.

Formalmente, um processo estocástico $Z = \{Z(t), t \in \mathcal{T}\}$ é **fracamente estacionário** ou estacionário de segunda ordem, se e somente se:

(i) $\mathrm{E}\{Z(t)\} = \mu(t) = \mu, \quad \forall\, t \in \mathcal{T}$;

(ii) $\mathrm{E}\{Z^2(t)\} < \infty, \quad \forall\, t \in \mathcal{T}$;

(iii) $\gamma(t_1, t_2) = \mathrm{Cov}\{Z(t_1), Z(t_2)\}$ é uma função de $|t_1 - t_2|, \quad \forall\, t_1, t_2 \in \mathcal{T}$.

[2] jargão em inglês que em português pode significar latência ou atraso.

Dessa forma, podemos dizer que processos estacionários de segunda ordem desenvolvem-se em torno de uma média constante, ou seja, ao redor de uma mesma tendência ou reta. É possível citar dois tipos de não-estacionaridade.

Existem os processos *homogêneos*, que apresentam uma estacionariedade inicial mas que depois sofrem uma mudança de tendência e então tornam-se estacionárias novamente, mas não necessariamente ao redor da mesma média inicial, e que segundo Morettin e Toloi (Morettin e Castro Toloi, 2019) podem se tornar estacionários se tomarmos diferenças sucessivas da série original.

Tomar diferenças de uma série $Z(t)$ corresponde a criar uma nova série a partir da original. A primeira diferença de $Z(t)$ é definida por:

$$\Delta Z(t) = Z(t) - Z(t-1) \tag{4.5}$$

Recursivamente, a partir dessa primeira definição, podemos escrever a n-ésima diferença de $Z(t)$ como sendo:

$$\Delta^n Z(t) = \Delta[\Delta^{n-1} Z(t)] \tag{4.6}$$

Adicionalmente, aplicar a transformação não-linear $\log Z(t)$ também pode ser útil para transformar uma série não-estacionária em uma série estacionária. De acordo com Morettin e Toloi (Morettin e Castro Toloi, 2019), a transformação logarítmica é muito usada em séries econômicas e será apropriada se a variância da série for proporcional à média.

Para exemplificar esses conceitos, considere a Figura 4.2. Temos à esquerda uma série temporal que é uma realização de um processo não-estacionário homogêneo, a saber, índices mensais da bolsa de valores Ibovespa. À direita, foi aplicado o logaritmo da primeira diferença da série, o que gerou uma série estacionária.

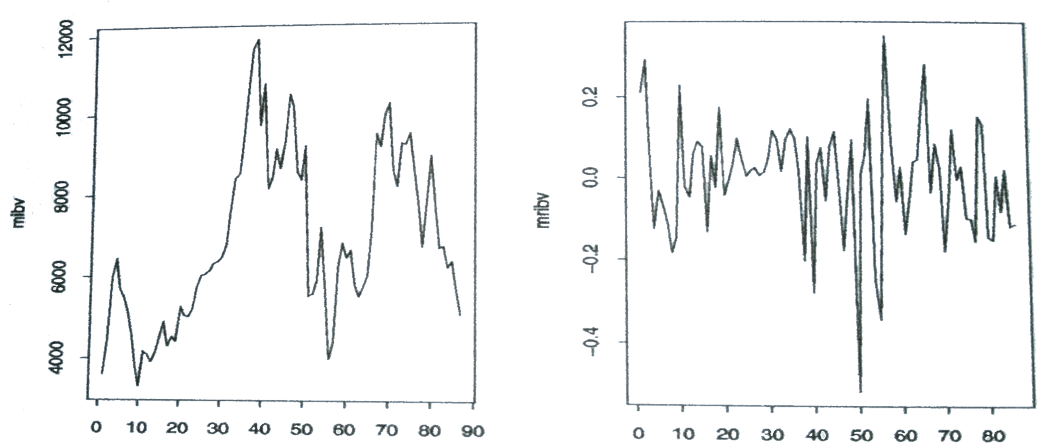

Figura 4.2: *Esquerda: Índices mensais do Ibovespa. Direita: Log-diferença do Ibovespa.*[a]

[a]Extraído de Morettin e Toloi, 2019, pág 7.

O segundo tipo de não-estacionaridade é chamada de *explosiva*. Um exemplo de um processo não-estacionário explosivo é uma série temporal que descreve o crescimento de uma população de bactérias. Não tratamos de processos deste tipo neste trabalho.

4.1.3 Função de autocorrelação

Seja $\{X_t, t \in \mathbb{Z}\}$ um processo estacionário real com tempo discreto, de média $\mu = 0$ e facv $\gamma_\tau = \mathrm{E}\{X_t, X_{t+\tau}\}$. Sob essas condições, Morettin e Toloi (Morettin e Castro Toloi, 2019) demonstram que a facv γ_τ satisfaz as propriedades:

(i) $\gamma_0 > 0$,

(ii) $\gamma_{-\tau} = \gamma_\tau$,

(iii) $|\gamma_\tau| \leq \gamma_0$,

(iv) $\sum_{j=1}^{n} \sum_{k=1}^{n} a_j a_k \gamma_{\tau_j - \tau_k} \geq 0 \quad \forall a_1, \ldots, a_n \in \mathbb{R}$ e $\tau_1, \ldots, \tau_n \in \mathbb{Z}$,

(v) $\lim_{|\tau| \to \infty} \gamma_\tau = 0$.

Define-se a **função de autocorrelação** (fac) de um processo estocástico por:

$$\rho_\tau = \frac{\gamma_\tau}{\gamma_0} \tag{4.7}$$

A fac de um processo estacionário possui todas as propriedades da facv acima listadas e, em particular, $\rho_0 = 1$. O mais interessante é a ressalva de Morettin e Toloi (Morettin e Castro Toloi, 2019) de que a recíproca é verdadeira, isto é, dado um processo cuja fac ou facv possuem essas propriedades, então ele é estacionário. Dessa forma, temos um arcabouço teórico que nos permite investigar a existência da propriedade estacionária de uma série temporal dada.

4.1.4 Exemplos de processos estocásticos

O exemplo mais simples é o de uma *sequência aleatória*. Uma sequência de v.a. definidas num mesmo espaço amostral Ω dada por $\{X_n, n = 1, 2, \ldots\}$ é um processo estocástico com parâmetro discreto, ou seja, $\mathcal{T} = \{1, 2, \ldots\}$. Para todo $n \geq 1$ temos, em geral, que:

$$P\{X_1 = a_1, \ldots, X_n = a_n\} = P\{X_1 = a_1\} \times P\{X_2 = a_2 | X_1 = a_1\}$$
$$\times \ldots \times P\{X_n = a_n | X_1 = a_1, \ldots, X_{n-1} = a_{n-1}\}$$

Se simplificarmos esse caso geral para o caso de uma sequência $\{X_n, n \geq 1\}$ de v.a. *mutuamente independentes* então:

$$P\{X_1 = a_1, \ldots, X_n = a_n\} = P\{X_1 = a_1\} \times \ldots \times P\{X_n = a_n\}$$

E se, além disso, todas as v.a. dessa sequência tiverem a mesma distribuição de probabilidades, elas serão portanto independentes e identicamente distribuidas (i.i.d.), o que

configura $X_n = \{X_n, n \geq 1\}$, uma sequência de v.a. i.i.d., como um processo estocástico estacionário.

Definindo $\mathrm{E}\{X_n\} = \mu$, $\mathrm{Var}\{X_n\} = \sigma^2$, para todo $n \geq 1$, teremos que a facv de X_n será dada por:

$$\gamma_\tau = \mathrm{Cov}\{X_n, X_{n+\tau}\} = \begin{cases} \sigma^2, & \text{se } \tau = 0 \\ 0, & \text{se } \tau \neq 0 \end{cases} \tag{4.8}$$

E, a fac de X_n, será tal que:

$$\rho_\tau = \begin{cases} 1, & \text{se } \tau = 0 \\ 0, & \text{se } \tau \neq 0 \end{cases} \tag{4.9}$$

Um segundo exemplo de processo estocástico, e muito mais útil para os estudos de séries temporais, é o de **ruído branco**. Morettin e Toloi (Morettin e Castro Toloi, 2019) definem que a sequência $\{\epsilon_t,\ t \in \mathbb{Z}\}$ é um *ruído branco discreto* se as v.a. ϵ_t não são correlacionadas, ou seja, $\mathrm{Cov}\{\epsilon_t, \epsilon_s\} = 0, t \neq s$.

Esse processo será estacionário se $\mathrm{E}\{\epsilon_t\} = \mu_\epsilon$ e $\mathrm{Var}\{\epsilon - t\} = \sigma^2_\epsilon$, para todo $t \in \mathbb{Z}$. Dessa forma, as facv e fac de um ruído branco serão dadas, respectivamente e analogamente, por (4.8) e (4.9). Tipicamente representa-se o gráfico de uma função de autocorrelação conforme o exemplo do fac de um ruído branco, dado na Figura 4.3.

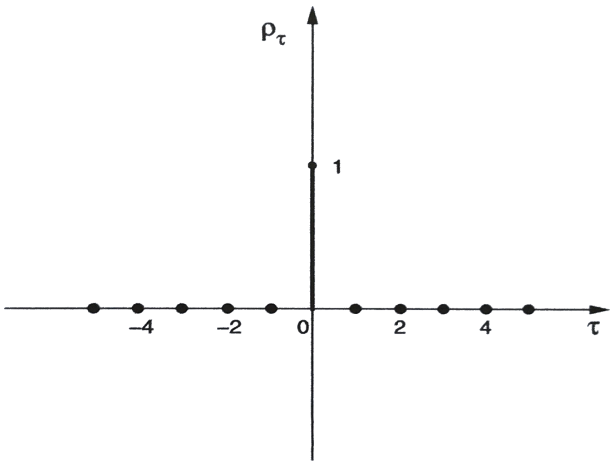

Figura 4.3: *Função de autocorrelação (fac) de um ruído branco.*[a]

[a]Extraído de Morettin e Toloi, 2019, pág 35.

Normalmente, é dito por Morettin e Toloi (Morettin e Castro Toloi, 2019) que, sem perda de generalidade, podemos assumir que a média de um ruído branco é zero. E, assim, escrevemos:

$$\epsilon_t \sim RB(0, \sigma^2_\epsilon).$$

Se, as v.a. de ϵ_t forem independentes, então é um resultado de probabilidade conhecido

e citado por Morettin e Toloi (Morettin e Castro Toloi, 2019) que serão também não correlacionados. Um ruído branco, como definido acima, e com a propriedade adicional da independência, é um terceiro exemplo de um processo estocástico, chamado de *processo puramente aleatório*. Nesse caso, escrevemos:

$$\epsilon_t \sim i.i.d.(0, \sigma_\epsilon^2).$$

Outros exemplos de processos estocásticos que podemos citar, menos formais e mais físicos, são os *passeios aleatórios*, que fazem por exemplo, a modelagem de um sistema de partículas livres, como as moléculas de um gás ideal. E, outro exemplo, análogo, é o do *movimento browniano*, que modela, por exemplo, como partículas de poeira movimentam-se num fluído visto como um conjunto de muitas moléculas ligadas entre si, como a água no estado líquido.

4.2 Os modelos ARIMA

Já foi citado acima que existem os modelos parâmétricos, que lidam com as séries no domínio temporal, e os modelos não-paramétricos que lidam com o **espectro** da série, isto é, o conjunto das frequências da série, onde as componentes desse espaço das frequências são ortogonais entre si, o que garante a não correlação entre as frequências, diferindo dos valores sob o domínio temporal que quase sempre são correlacionados entre si, conforme explicado por Morettin e Toloi (Morettin e Castro Toloi, 2019).

Se temos uma série temporal discreta $Z = \{Z_t, t \in \mathbb{Z}\}$, que assumimos ser um processo estacionário, e assumindo também que as autocovariâncias γ_τ são tais que $\sum_{\tau=-\infty}^{\infty} |\gamma_\tau| < \infty$, então o espectro de Z, isto é, a **transformada de Fourier** de Z será:

$$f(\omega) = \frac{1}{2\pi} \sum_{\tau=-\infty}^{\infty} \gamma_\tau e^{-i\omega\tau}, \quad -\pi \leq \omega \leq \pi. \tag{4.10}$$

Ora, isto permite uma definição alternativa para a função de autocovariância (facv), como sendo a *anti-transformada* de Z, pois:

$$\gamma_\tau = \int_{-\pi}^{\pi} e^{i\omega\tau} f(\omega) d\omega, \quad \tau \in \mathbb{Z}. \tag{4.11}$$

Embora úteis para a construção de modelos de engenharia e física, Morettin e Toloi (Morettin e Castro Toloi, 2019) ressaltam que usar o espectro ou a facv para modelar uma série temporal é mais factível no contexto de processos industriais, e que o uso mais prático em outros contextos, como o de séries financeiras é o papel que desempenham nos modelos paramétricos como o ARIMA.

Isto vem do fato de que os modelos paramétricos tem comportamentos esperados, com valores de facv, em alguns casos, muito bem definidos e que podem ser testados e usados como uma validação ao modelo que pretende-se estimar a série temporal de estudo.

Os modelos ARIMA correspondem a um conjunto de modelos, alguns mais básicos e outros que são formados pelo agrupamento de 2 ou 3 dos modelos básicos. A metodologia mais usada, inclusive por Morettin e Toloi (Morettin e Castro Toloi, 2019), para a análise desses modelos paramétricos é aquela conhecida como abordagem de Box e Jenkins (Box, Jenkins *et al.*, 2016).

Essa metodologia é análoga àquela utilizada no contexto de aprendizado de máquina, seguindo um ciclo iterativo composto de quatro estágios. O primeiro estágio é a *especificação* de uma classe de modelos, neste caso, os modelos ARIMA.

O segundo estágio é a *identificação* de um modelo específico, o que será feito após análise da função de autocorrelação e de outros critérios. O terceiro estágio consiste na *estimação*, na qual os parâmetros do modelo escolhido são estimados.

Ao final do ciclo temos a *verificação* do modelo ajustado, por meio de uma análise de resíduos, de forma a saber se este é de fato um modelo adequado para os objetivos do estudo, por exemplo, a previsão de valores futuros da série.

Se o modelo ajustado se mostrar adequado, então o ciclo é repetido novamente. Ao final de alguns ciclos deve-se escolher, dentre os que se mostraram adequados, o melhor possível, segundo uma métrica escolhida, o erro quadrático médio, por exemplo, que irá fornecer as previsões.

Seguem algumas definições úteis para os modelos que serão descritos a seguir. Temos o operador *translação para o passado B*, que será definido como:

$$BZ_t = Z_{t-1}, \quad B^m Z_t = Z_{t-m}$$

A seguir, o operador *translação para o futuro F*, definido por:

$$FZ_t = Z_{t+1}, \quad F^m Z_t = Z_{t+m}$$

O operador *diferença*, denotado por Δ e definido por:

$$\Delta Z_t = Z_t - Z_{t-1} = (1 - B)Z_t \tag{4.12}$$

E o operador *soma*, denotado por S e definido por:

$$SZ_t = Z_t + Z_{t-1} + Z_{t-2} + \ldots = (1 + B + B^2 + \ldots)Z_t$$

Combinando as expressões acima, resulta que:

$$S = \Delta^{-1}$$

4.2.1 Processo linear geral

O modelo mais básico, assume que a série temporal é gerada por um sistema linear, que possui como entradas ruídos brancos, já caracterizados acima como processos estocásticos. Uma série Z_t é dita um **processo linear** se for definida como:

$$Z_t = \mu + a_t + \psi_1 a_{t-1} + \psi_2 a_{t-2} + \ldots = \mu + \psi(B)a_t \tag{4.13}$$

onde os termos $a_t \sim RB(0, \sigma_a^2)$, ou seja, são ruídos brancos, μ é um parâmetro que define o nível da série e ψ é a *função de transferência* definida por:

$$\psi(B) = 1 + \psi_1 B + \psi_2 B^2 + \ldots$$

Escrevendo $\tilde{Z}_t = Z_t - \mu$, simplifica-se a notação para:

$$\tilde{Z}_t = \psi(B)a_t \tag{4.14}$$

A função de transferência é composta de uma sequência de *pesos*, $\{\psi_j, j \geq 1\}$, que, se convergir a um número real, dizemos que a função é estável, o que nos leva à seguinte **proposição** feita por Morettin e Toloi (Morettin e Castro Toloi, 2019) e demonstrada por Box e Jenkins (Box, Jenkins *et al.*, 2016):

Um processo linear será estacionário se a série $\psi(B)$ convergir quando $|B| \leq 1$. (4.15)

Com isto define-se o conceito de raízes de equações envolvendo operadores (como $\psi(B)$) dentro ou fora do *círculo unitário*, abstraindo a condição acima para B^p, com $p \geq 1$. Pode-se observar a validade dessa proposição tomando a esperança da expressão (4.13):

$$\mathrm{E}(Z_t) = \mu + \mathrm{E}\left(a_t + \sum_{j=1}^{\infty} \psi_j a_{t-j}\right)$$

Se a série dos pesos acima convergir, segue o resultado descrito acima, já que $\mathrm{E}(a_t) = 0$ para todo t já que os termos a_t são ruídos brancos, e assim $\mathrm{E}[\tilde{Z}_t] = 0$. Neste caso, o parâmetro μ será a média do processo. Caso contrário, μ não terá significado específico e apenas representará o nível da série em algum intervalo de tempo observado.

Uma forma alternativa de escrever (4.13), definida por Morettin e Toloi (Morettin e Castro Toloi, 2019), é como uma soma ponderada de valores passados, já utilizando a notação simplificada de (4.14):

$$\tilde{Z}_t = \sum_{j=1}^{\infty} \phi_j \tilde{Z}_{t-j} + a_t \tag{4.16}$$

Assim, nessa notação temos um outro conjunto de pesos, denotados por ϕ_t, e com os quais pode-se definir o operador:

$$\phi(B) = 1 - \sum_{j=1}^{\infty} \phi_j B^j \tag{4.17}$$

A partir do qual, pode-se simplificar a notação de (4.16) para:

$$\phi(B)\tilde{Z}_t = a_t \tag{4.18}$$

Assim, combinando (4.18) e (4.14), obtemos uma relação entre os operadores definidos para as duas notações de um processo linear geral:

$$\phi(B) = \psi^{-1}(B) \tag{4.19}$$

Esse tipo de modelo é um ponto de partida, a partir do qual complicações são adicionadas para criar os outros modelos, mas que já serve para modelar, por exemplo, passeios aleatórios. Morettin e Toloi (Morettin e Castro Toloi, 2019) definem alguns conceitos adicionais como a estacionaridade e a invertibilidade desses processos, mas que configuram detalhes que fogem do escopo deste trabalho.

4.2.2 Modelos autorregressivos (AR)

Tomando $\phi_j = 0$ para $j > p$ em (4.16), obtém-se um *processo autorregressivo de ordem p*, denotado por $AR(p)$ e escrito por:

$$\tilde{Z}_t = \phi_1\tilde{Z}_{t-1} + \phi_2\tilde{Z}_{t-2} + \ldots + \phi_p\tilde{Z}_{t-p} + a_t \tag{4.20}$$

Definindo o operador autorregressivo de ordem p por:

$$\phi(B) = 1 - \phi_1 B - \phi_2 B^2 - \ldots - \phi_p B^p$$

Pode-se escrever (4.20) como:[3]

$$\phi(B)\tilde{Z}_t = a_t \tag{4.21}$$

Tomando o exemplo mais simples possível de um modelo AR poderemos investigar a estacionaridade desse modelo. Quando $\tilde{Z}_t \sim AR(1)$:

$$\tilde{Z}_t = \phi\tilde{Z}_{t-1} + a_t \tag{4.22}$$

ou, usando a notação de (4.21), nesse caso:

$$(1 - \phi B)\tilde{Z}_t = a_t$$

[3]Neste caso, o operador $\phi(B)$ é finito

o que nos diz que nesse modelo, um valor da série num tempo t depende apenas do valor da série e de um ruído no tempo anterior.

Substituindo recursivamente $\tilde{Z}_{t-1}, \tilde{Z}_{t-2}, \ldots$ em (4.22), e lembrando da definição da função de transferência em (4.13), obtemos:

$$\tilde{Z}_t = \sum_{j=0}^{\infty} \phi^j a_{t-j} = \psi(B) a_t$$

Por (4.19) e (4.22) chegamos em:

$$\psi(B) = [\phi(B)]^{-1} = (1 - \phi B)^{-1}$$

De acordo com a Proposição (4.15), Z_t será estacionária se $\psi(B)$ convergir a um número real quando $|B| \leq 1$. Pela expressão anterior, deveremos ter também $|\phi| < 1$. Isto significa dizer que a raiz da equação $\phi(\hat{B}) = 0$ estará *fora do círculo unitário*, pois: $\hat{B} = \phi^{-1} > 1$.

4.2.3 Modelos de médias móveis (MA)

Considerando o processo linear em (4.13), e supondo que $\psi_j = 0$ quando $j > q$, obtemos um *processo de médias móveis de ordem q*, denotado por $MA(q)$, e é escrito, usando a notação $\theta_j := -\psi_j$, $\forall j > 0$, da forma:

$$\tilde{Z}_t = a_t - \theta_1 a_{t-1} - \ldots - \theta_q a_{t-q} = (1 - \theta_1 B - \ldots - \theta_q B^q) a_t \tag{4.23}$$

Ou seja, definindo o operador de médias móveis de ordem q, $MA(q)$ por:

$$\theta(B) = 1 - \theta_1 B - \theta_2 B^2 - \ldots - \theta_q B^q$$

Escrevemos (4.23) simplificadamente como:

$$\tilde{Z}_t = \theta(B) a_t \tag{4.24}$$

Considerando o modelo mais simples, ou seja, $MA(1)$, uma série assim modelada seria escrita:

$$\tilde{Z}_t = a_t - \theta a_{t-1} = (1 - \theta B) a_t$$

Novamente, temos que esta é uma série cujo valor em qualquer tempo só depende do valor no tempo anterior e de um ruído. Aplicando o mesmo raciocínio que foi utilizado no modelo $AR(1)$, utilizando a proposição (4.15), a série Z_t será estacionária quando as raízes da equação $\theta(\hat{B}) = 0$ se encontrarem fora do círculo unitário, isto é, $\hat{B} > 1$.

4.2.4 Modelos autorregressivos e de médias móveis (ARMA)

Como o nome já indica, podemos combinar os dois modelos num só, assumindo que uma série seja escrita como a soma das componentes autorregressivas com as de médias móveis, resultando num modelo $ARMA(p, q)$:

$$\tilde{Z}_t = \phi_1 \tilde{Z}_{t-1} + \ldots + \phi_p \tilde{Z}_{t-p} + a_t - \theta_1 a_{t-1} - \ldots - \theta_q a_{t-q} \tag{4.25}$$

Utilizando os operadores já respectivamente definidos, simplifica-se para:

$$\phi(B)\tilde{Z}_t = \theta(B)a_t \tag{4.26}$$

Analisando o modelo simples definido por $ARMA(1, 1)$, a série em (4.25) é escrita:

$$\tilde{Z}_t = \phi \tilde{Z}_{t-1} + a_t - \theta a_{t-1} \tag{4.27}$$

Substituindo-se recursivamente $\tilde{Z}_{t-1}, \tilde{Z}_{t-2}, \ldots$ nessa expressão, a série será escrita na forma de um processo linear, ou seja, um modelo de médias móveis de ordem infinita:

$$\tilde{Z}_t = \psi(B)a_t$$

Assim, Morettin e Toloi (Morettin e Castro Toloi, 2019) mostram que, da mesma forma que no modelo $AR(1)$, a série $ARMA(1, 1)$ será estacionária se a raíz da equação $\phi(\hat{B}) = 0$ estiver fora do círculo unitário. Inclusive, vale a generalização para os modelos $AR(p)$ e $ARMA(p, q)$, que serão processos estacionários se as p raízes do polinômio autorregressivo estiverem todas fora do círculo unitário.

Essa generalização é bem útil quando temos que decidir se uma série dada é ou não estacionária. Há testes estatísticos específicos que testam, para uma série real, a hipótese nula que é a existência de raízes no círculo unitário, o que a configura como uma série não estacionária, contra a hipótese alternativa da não-existência de raízes unitárias, caracterizando-a então como uma série estacionária.

Dentre os testes de estacionaridade existentes, Morettin e Toloi (Morettin e Castro Toloi, 2019) apresentam-nos o teste de *Dickey e Fuller*, que utiliza uma estatística de teste com distribuição tabelada, e que será utilizado na parte prática do trabalho, a partir de bibliotecas disponíveis na linguagem *python*, durante a estimação dos modelos paramétricos para as séries temporais aqui estudadas.

4.2.5 Os modelos integrados não-estacionários (ARIMA)

Todos os modelos até agora vistos são úteis para modelar séries estacionárias. Morettin e Toloi (Morettin e Castro Toloi, 2019) nos lembram que várias séries econômicas e financeiras não são estacionárias, mas tornam-se estacionárias após operações de diferenças, já definidas em (4.12).

Já vimos que as séries não-estacionárias podem ser do tipo *homogêneas* ou do tipo *explosivas*. Seja Z_t uma série representada por um modelo $AR(1)$:

$$(1 - \phi B)\tilde{Z}_t = a_t$$

A condição de estacionariedade é $|\phi| < 1$. Se $\phi = 1$ então a série é não-estacionária homogênea. Note que nesse caso:

$$(1 - B)\tilde{Z}_t = \Delta\tilde{Z}_t = a_t$$

pela própria definição do operador diferença em (4.12), e assim torna-se estacionária.

Se, por outro lado, $|\phi| > 1$, é fácil ver pela expressão acima que a série será do tipo *explosivo*, já que cada termo será aumentado numa taxa maior do que 1, caracterizando um crescimento exponencial do valor da série.

Dessa forma, séries Z_t que, sendo diferençadas d vezes tornam-se estacionárias, isto é, séries tais que $\Delta^d\tilde{Z}_t = a_t$, são chamadas de não estacionárias homogêneas, ou também de *integradas de ordem d*.

Se temos uma série $W_t = \Delta^d Z_t$ estacionária[4], então W_t pode ser representada por uma modelo $ARMA(p, q)$, isto é:

$$\phi(B)W_t = \theta(B)a_t$$

Sendo W_t uma diferença de Z_t, isto torna Z_t uma *integral* de W_t. Dizemos, neste caso, que Z_t segue um modelo *autorregressivo, integrado e de médias móveis* ou $ARIMA(p, d, q)$, escrito como:

$$\phi(B)\Delta^d Z_t = \theta(B)a_t \tag{4.28}$$

Assim, o operador $\phi(B)\Delta^d$ é chamado de autorregressivo não-estacionário, de ordem $p+d$, onde d raízes são iguais a 1, isto é, sobre o círculo unitário, e as demais p raízes estarão fora do círculo unitário. Temos que a d–ésima diferença de Z_t pode ser representada por um modelo $ARMA(p, q)$, estacionário.

4.2.6 Identificação dos modelos utilizando a função de autocorrelação

Com uma série real em mãos, queremos poder representá-la com algum dos modelos $ARIMA$ vistos. Mas como determinar os parâmetros p, d e q? Morettin e Toloi (Morettin e Castro Toloi, 2019) consideram um procedimento de três passos.

O **primeiro passo** é verificar se há a necessidade de aplicar uma transformação não-linear à série original, com o objetivo de estabilizar a sua variância, uma vez que, de

[4] Perceba que $\Delta^d\tilde{Z}_t = \Delta^d Z_t$

acordo com Morettin e Toloi (Morettin e Castro Toloi, 2019) é comum que em séries econômicas e financeiras existam tendências que causem um aumento da variância em função do tempo.

Uma das transformações mais usadas, nesse caso, é a logarítmica, ou uma versão geral dela, conhecida como *transformação de Box-Cox*. Os detalhes da transformação e o procedimento de como utilizá-la para estabilizar a variância de uma série temporal estão presentes no Apêndice B.

O **segundo passo**, é tomar diferenças da série, já transformada (se houve necessidade), um número de vezes que for necessária a torná-la estacionária, caso ela não seja. Este número de vezes será o parâmetro d do modelo $ARIMA$, de forma que a nova série, dada por $\Delta^d Z_t$ poderá ser dada por um modelo $ARMA(p, q)$.

A escolha apropriada de d será feita com a ajuda do teste de estacionaridade, conforme explicado no item anterior, de forma iterativa. Assim, fazendo o teste para a série original, se este identificar a presença de raízes unitárias, uma vez que o teste de Dickey-Fuller testa se há *alguma* raiz unitária, então a série não é estacionária.

Assim tomamos uma diferença na série e realizamos o teste novamente. Se ele ainda apontar a existência de alguma raiz unitária, tomamos nova diferença. Repetimos até que o teste dê evidência de que não há mais raízes unitárias.

Este é um procedimento delicado e de natureza estocástica, assim pode ser útil o uso de outra verificação conforme as diferenças vão sendo tomadas da série. Morettin e Toloi (Morettin e Castro Toloi, 2019) sugerem avaliarmos a variância da série diferençada.

Quando a série é corretamente diferençada, a variância irá diminuir, porém, um excesso de diferenças fará com que a variância volte a aumentar. Assim, um monitoramento do comportamento da variância pode ser útil para verificarmos quando devemos parar e assim definir o parâmetro d mais adequado.

Por fim, o **terceiro passo**, é estimar os parâmetros autorregressivos e de médias móveis, respectivamente p e q. Isto pode ser feito via análise de estimativas da função de autocorrelação, e de uma versão alternativa desta, chamada de *autocorrelação parcial*, que será definida a seguir.

Essas estimativas deverão imitar o comportamento das quantidades teóricas dessas funções, que são definidas justamente pelos parâmetros p e q. É a análise *visual* das estimativas dessas funções, e com a ajuda da receita do comportamento teórico que listaremos abaixo, que serão os guias para a escolha dos parâmetros do modelo.

O comportamento das funções de autocorrelação dos modelos $AR(p)$, $MA(q)$ e $ARMA(p, q)$ são listados por Morettin e Toloi (Morettin e Castro Toloi, 2019) como se segue:

(i) A fac de um modelo $AR(p)$ decai exponencialmente, ou de acordo com senóides amortecidas, possuindo uma extensão infinita de valores não-nulos.

(ii) A fac de um modelo $MA(q)$ é finita, no sentido que seu valor é zerado após o *lag*[5]

[5]Lembramos que o *lag* de ordem τ, é um deslocamento temporal da série, isto é, $Z_{t+\tau}$, a partir do qual

temporal de ordem q.

(iii) A fac de um modelo $ARMA(p, q)$ é infinita em extensão, e decai exponencialmente após o *lag* $q-p$.

Como pode ser complicada a análise de apenas a função de autocorrelação, Box e Jenkins (Box, Jenkins *et al.*, 2016) derivaram a **função de autocorrelação parcial** (facp) a partir da fac de um modelo $AR(k)$, da seguinte maneira:

$$\varphi(k) = \frac{|\mathbf{P}_k^*|}{|\mathbf{P}_k|} \tag{4.29}$$

Onde, $\mathbf{P}_k$ é a matriz de autocorrelações do modelo $AR(k)$, e $\mathbf{P}_k^*$ é a mesma matriz mas com a última coluna substituída pelo vetor de autocorrelações do modelo. As definições detalhadas de ambos, e a derivação da equação (4.29) podem ser vistas no Apêndice C.

Dessa forma, a facp, dada pela quantidade $\varphi(k)$ é uma função do parâmetro k, que será o *lag*, e para a qual é demonstrada em Box e Jenkins (Box, Jenkins *et al.*, 2016) o seguinte comportamento:

(i) A facp de um modelo $AR(p)$ é tal que $\varphi(k) \neq 0$, para $k \leq p$ e $\varphi(k) = 0$, para *lags* $k > p$.

(ii) A facp de um modelo $MA(q)$ tem comportamento análogo ao fac de um modelo $AR(p)$, ou seja, com decaimento exponencial ou por senóides amortecidas e infinita em extensão.

(iii) A facp de um modelo $ARMA(p, q)$ tem um comportamento similar ao comportamento da facp de um modelo $MA(q)$.

Pode ser bem complicada a análise de modelos com parâmetros p, q ou d muito grandes. Para uma tentativa de ilustração, temos na Figura 4.4 abaixo, exemplos de fac e facp dos modelos $AR(1)$, $MA(1)$ e $ARMA(1, 1)$.

Esta é uma clara desvantagem dos modelos paramétricos de aprendizagem, já que dependem de parâmetros que devem ser definidos *a priori*, e os procedimentos para estimá-los, como neste caso, podem ser inexatos e até mesmo inconclusivos, dada a natureza complicada das funções de autocorrelação.

A recomendação dada por Morettin e Toloi (Morettin e Castro Toloi, 2019) é a mais comum, no âmbito de aprendizado de máquina, e consiste em testar vários parâmetros que se mostrem razoáveis após as análises feitas, e verificar qual série modelada é a mais adequada à série real de interesse, de acordo com alguma métrica de erro.

as funções de autocovariância/autocorrelação são calculadas.

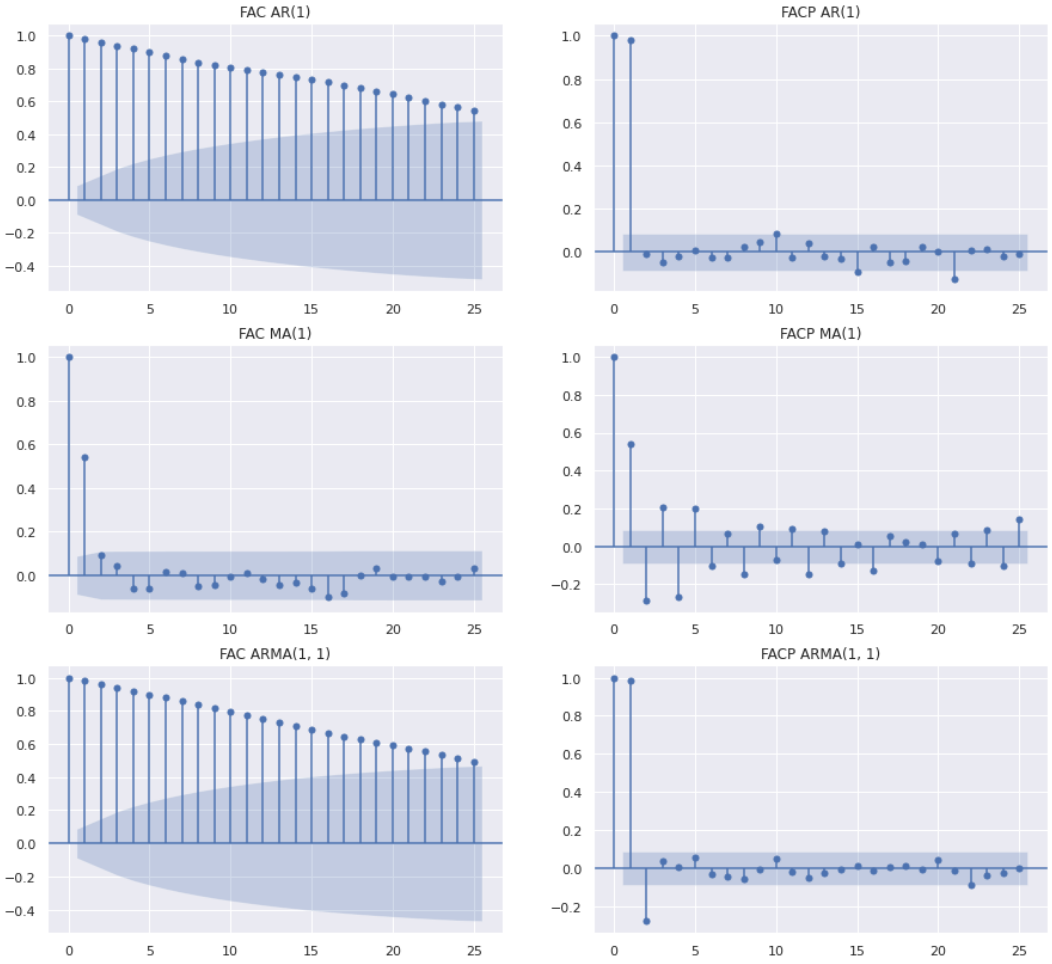

Figura 4.4: *Autocorrelações e autocorrelações parciais amostrais de alguns modelos ARIMA.*

Capítulo 5

Procedimentos de comparação e resultados

Neste capítulo serão definidos procedimentos de modelagem alternativos às séries temporais financeiras de taxas de câmbio. Um dos modelos é o modelo paramétrico *ARIMA*, estudado no Capítulo 4.

O outro será um modelo semiparamétrico criado com redes neurais, estudadas no Capítulo 3, sendo utilizadas redes neurais recorrentes, devido à sua arquitetura mais condizente com o problema em questão, de acordo com Kopec (Kopec, 2019) e Géron (Géron, 2019).

A seguir, serão feitas comparações entre os modelos preditivos de séries temporais, através de medidas de desempenho comuns como percentual de acerto das previsões e observações das funções de erros dos algoritmos utilizados.

5.1 Conhecendo a série temporal de interesse

5.2 Estimando um modelo ARIMA

5.3 Construção do modelo com uma rede neural

Apêndice A

O gradiente descendente

O gradiente descendente é um dos métodos de otimização de funções, no contexto de aprendizado máquina é geralmente utilizado para minimizar funções de custo. A ideia geral é ajustar parâmetros iterativamente para otimizar a função de custo gradativamente.

O seu funcionamento utiliza a ideia fundamental do Cálculo em que utilizamos a derivada de uma função de forma a encontrar seus pontos extremos. Dada uma função $f : \mathbb{R}^n \longrightarrow \mathbb{R}$, temos que se um ponto $x = \hat{x} \in \mathbb{R}$ é um ponto extremo de f então é condição necessária[1] que cada derivada parcial de primeira ordem de f exista e seja igual a zero. Denotando $x = (x_0, x_1, \dots, x_n)$ e $\hat{x} = (\hat{x_0}, \hat{x_1}, \dots, \hat{x_n})$, temos:

$$\frac{\partial f(\hat{x_0})}{\partial x_0} = 0, \quad \frac{\partial f(\hat{x_1})}{\partial x_1} = 0, \quad \dots, \quad \frac{\partial f(\hat{x_n})}{\partial x_n} = 0 \tag{A.1}$$

Usando a notação de vetores, podemos simplificar a equação acima, uma vez que o conjunto das derivadas parciais de uma função de várias variáveis é o vetor gradiente desta função, assim, denotando $\mathbf{0} \in \mathbb{R}^n = (0, \dots, 0)$, temos equivalentemente à equação A.1:

$$\nabla f(\hat{x}) = \mathbf{0} \tag{A.2}$$

onde $\nabla : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ é a função que calcula o gradiente para um dado ponto de uma função.

Geometricamente é nos dada a intuição, por Luis Hamilton Guidorizzi (Guidorizzi, 1986), de que o vetor gradiente de um dado ponto de uma função nos dá a direção de maior aumento da função naquele ponto. Como nosso objetivo é minimizar a função de custo, fica explicado o nome do algoritmo como "gradiente descendente", de forma que devemos utilizar o sentido negativo do vetor gradiente.

Assim, podemos dizer que a direção de minimização da função está na direção do vetor gradiente, o que significa dar um passo ($f(x + dx)$) nessa direção no domínio da função, tal passo com tamanho que seja *proporcional* a cada componente do vetor gradiente. Dessa

[1] Mais detalhes em Guidorizzi (Guidorizzi, 1986). Um curso de cálculo Vol. 2, pág. 894.

forma podemos escrever dx como sendo um passo na direção do mínimo da função de custo dessa forma:

$$dx = -\eta \nabla f(x) \tag{A.3}$$

onde η é a constante de proporcionalidade, que é conhecida como **taxa de aprendizado**, que tem por objetivo tornar a velocidade do treinamento ajustável durante a execução do algoritmo, sendo tarefa do cientista de dados testar e obter os valores que dêem os melhores resultados caso-a-caso.

Podemos observar as diferenças de utilizar uma taxa de aprendizado fixa ou variável (que mude a cada iteração do treinamento, por exemplo), utilizando gráficos. Suponha que estamos querendo minimizar uma função quadrática do tipo $f(x) = x^2$, essa restrição é particularmente útil uma vez que a função de custo que será utilizada, a MSE, é uma função quadrática deste tipo.

Outra característica igualmente útil é que a segunda derivada deste tipo de função quadrática é sempre positiva[2], o que implica que o ponto extremo encontrado será necessariamente um ponto de mínimo.

Inicialmente, sorteamos um ponto inicial e calculamos o valor da função e o valor do gradiente neste ponto. Se as componentes do gradiente não forem todas arbitrariamente próximas de zero, então quer dizer que não atingimos o mínimo, e dessa forma obtemos um novo ponto a partir deste somado com dx definido acima na equação A.3.

Repetimos este processo até que o gradiente do ponto atual seja arbitrariamente próximo do vetor nulo. Uma visualização deste processo, utilizando taxa de aprendizado fixada, está na Figura A.1.

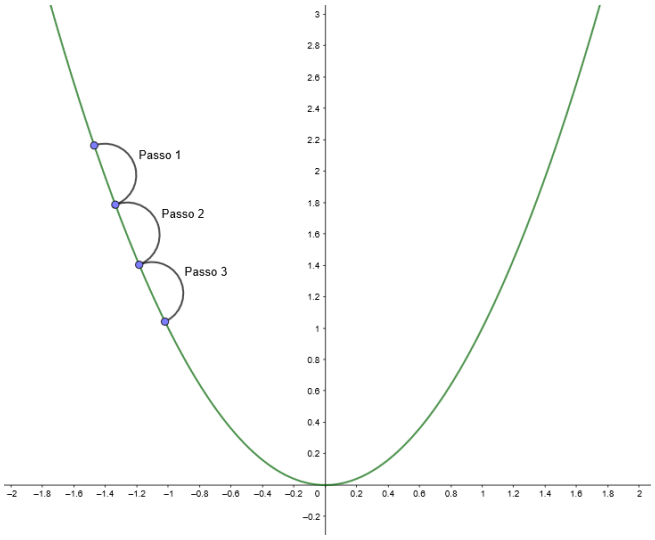

Figura A.1: *Visualização do método do gradiente descendente com taxa de aprendizado única. Os pontos azuis representam candidatos a ponto mínimo em cada iteração do algoritmo.*

[2] A segunda derivada de uma função de muitas variáveis é a matriz Hessiana, neste caso ela seria uma matriz definida positiva.

Podemos notar que as estimativas aproximam-se a uma velocidade constante do ponto de mínimo, que nesse caso ilustrativo é bem conhecido. Esse é o comportamento gerado por uma taxa de aprendizado fixa, e além disso com uma magnitude mediana.

O que poderia acontecer se utilizarmos uma taxa de aprendizado muito grande é que com um passo do algoritmo, o ponto estimado poderia ir para o outro lado do arco da função, e depois retornar, e assim por diante, nunca convergindo para o mínimo, uma ilustração disso está na Figura A.2.

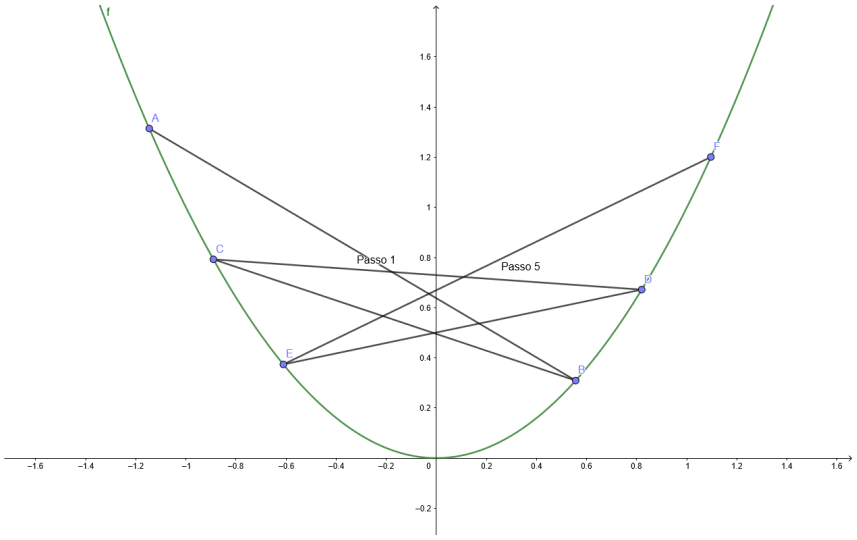

Figura A.2: *Visualização do método do gradiente descendente com taxa de aprendizado única. Ilustração do uso de um valor de taxa de aprendizado muito grande.*

O caso oposto a este, ou seja, usar uma taxa muito pequena, claramente irá fazer com os passos dados sejam muito pequenos, e dessa forma o algoritmo demore muito a convergir, por isso é importante usar valores medianos que podem ser obtidos de forma heurística, embora na prática, conforme dito por Géron (Géron, 2019), utiliza-se $\eta = 0.1$, sendo este um valor consensualmente utilizado pelo menos como ponto de partida.

A outra abordagem é utilizar valores variáveis, sendo o caso mais comum utilizar uma taxa que começa até mesmo maior do que o valor comum de 0.1 mas que vai diminuindo a cada passo, numa tentativa de obter uma convergência mais rápida. Uma ilustração desse caso está na Figura A.3.

Qualquer que seja o tipo de taxa de apredizado que venha a ser utilizado, permanece como melhor estratégia testar qual deles irá gerar o melhor resultado, analisando diretamente os valores candidatos a mínimo obtidos pelo algoritmo como função do número do passo, criando assim outro tipo de gráfico, no qual não precisamos saber o formato da função objetivo, o que é razoável uma vez que não precisariamos de um método numérico para obter seu ponto de mínimo.

Podemos observar este comportamento genérico para os 4 casos acima mencionados, na Figura A.4 temos os gráficos de valores hipotéticos de candidatos a mínimo gerados por (a) taxa de aprendizado fixa e grande, (b) taxa de aprendizado fixa e pequena, (c) taxa de aprendizado fixa e mediana, (d) taxa de aprendizado decrescente.

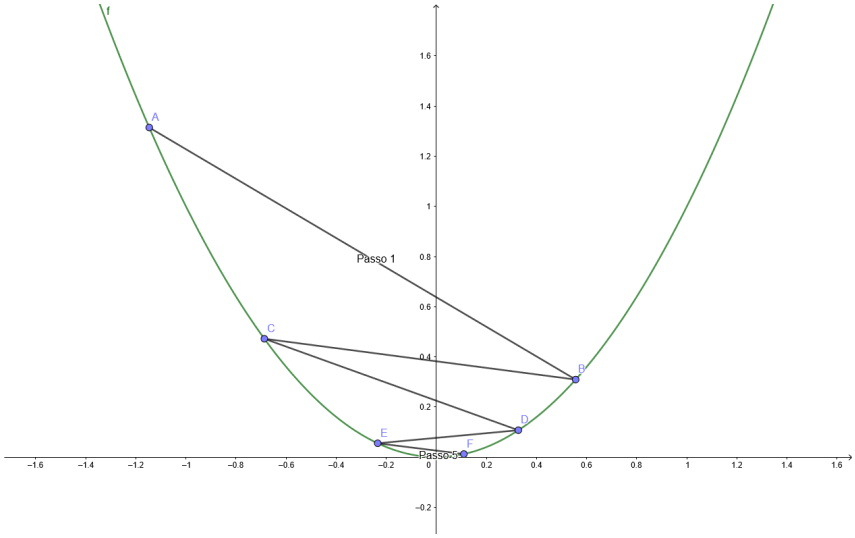

Figura A.3: *Visualização do método do gradiente descendente com taxa de aprendizado variável que vai diminuindo passo-a-passo do algoritmo.*

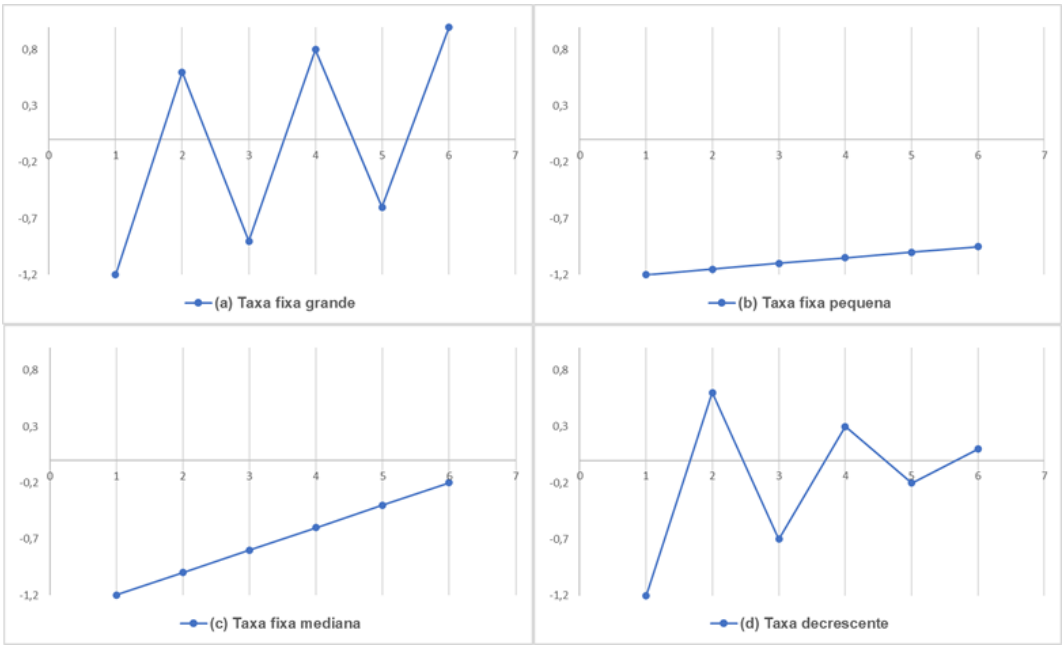

Figura A.4: *Comportamento de diferentes taxas de aprendizado nos valores candidatos a mínimo.*

A partir deste comportamento geral, podemos testar nosso problema-alvo, verificar a qual comportamento ele mais se parece e assim decidir se devemos aumentar ou diminuir nossa taxa até obtermos um bom comportamento como aqueles vistos em (c) ou (d).

Apêndice B

Transformação de Box-Cox

De forma a estabilizar a variância de séries temporais financeiras, faz-se necessária a aplicação de transformações não-lineares à série original, de forma a modelar a série transformada com os modelos *ARIMA*, que assumem uma variância constante, mesmo quando a média possui algumas tendências que podem ser compensadas aplicando-se diferenças, conforme visto no Capítulo 4.

A transformação a seguir, criada por George Edward Pelham Box e David Roxbee Cox (Box e Cox, 1964), e conhecida por transformação de Box-Cox, é uma generalização da transformação logarítmica. Fada uma série temporal Z_t, ela é definida por:

$$Z_t^{(\lambda)} = \begin{cases} \dfrac{Z_t^{\lambda} - 1}{\lambda}, & \text{se } \lambda \neq 0 \\ \log Z_t, & \text{se } \lambda = 0 \end{cases} \tag{B.1}$$

Onde, λ é um parâmetro real que deve ser estimado. Para isso, Morettin e Toloi (Morettin e Castro Toloi, 2019) sugerem a análise de um gráfico que será construído a partir de dados da série temporal. No eixo das abcissas calculam-se médias de subconjuntos de observações, e no eixo das ordenadas, calculam-se as amplitudes de cada um desses subconjuntos.

Seja $Z_1, \ldots, Z_k$ um subconjunto de k elementos da série temporal, definimos o par $(\bar{Z}, w)$ que será um ponto no gráfico, pelas componentes:

$$\begin{aligned} \bar{Z} &= \tfrac{1}{k} \textstyle\sum_{i=1}^{k} Z_{t_i} \\ w &= \max(Z_{t_i}) - \min(Z_{t_i}) \end{aligned}$$

Se, a partir desse gráfico, verificarmos que w independer de $\bar{Z}$, ou seja, se os pontos estiverem espalhados ao redor de uma reta paralela ao eixo das abcissas, não haverá necessidade de transformação, pois assim a variância da série é estável, de acordo com Box e Jenkins (Box, Jenkins *et al.*, 2016).

Se, por outro lado, w for diretamente proporcional a $\bar{Z}$, ou seja, correlacionadas com uma reta com inclinação próxima de 45 graus, então poderemos assumir que $\lambda = 0$. Por se

tratar de um procedimento empírico, Box e Jenkins (Box, Jenkins *et al.*, 2016) fornecem uma guia que pode ser utilizado para decidirmos por um valor adequado de λ.

Exibido na Figura B.1, são dados possíveis gráficos que poderemos obter com esse procedimento e qual o valor correspondente de λ que usaremos para aplicar a transformação B.1.

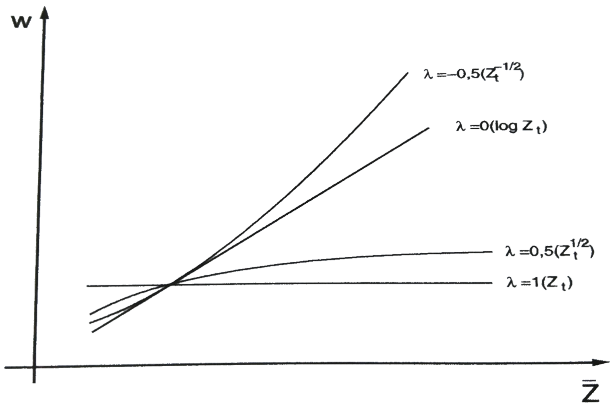

Figura B.1: *Gráficos da amplitude por média de subconjuntos de Z_t, com os correspondentes valores de λ.*[a]

[a]Extraido de Morettin e Toloi, 2019, pág 10.

Apêndice C

Função de autocorrelação parcial

A derivação da função de autocorrelação parcial mostrada abaixo, é extraída de Morettin e Toloi (Morettin e Castro Toloi, 2019), nas páginas 138-140.

Suponha um modelo $AR(k)$ e seja ϕ_{kj} o seu j-ésimo coeficiente. Sabe-se que:

$$\rho_j = \phi_{k1}\rho_{j-1} + \phi_{k2}\rho_{j-2} + \ldots + \phi_{kk}\rho_{j-k}\ , \quad j = 1, \ldots, k$$

A partir dessas k expressões, obtemos as chamadas equações de Yule-Walker:

$$\begin{bmatrix} 1 & \rho_1 & \rho_2 & \cdots & \rho_{k-1} \\ \rho_1 & 1 & \rho_1 & \cdots & \rho_{k-2} \\ & \vdots & & \ddots & \vdots \\ \rho_{k-1} & \rho_{k-2} & \rho_{k-3} & \cdots & 1 \end{bmatrix} \begin{bmatrix} \phi_{k1} \\ \phi_{k2} \\ \vdots \\ \phi_{kk} \end{bmatrix} = \begin{bmatrix} \rho_1 \\ \rho_2 \\ \vdots \\ \rho_k \end{bmatrix} \tag{C.1}$$

Resolvendo essas equações sucessivamente para $k = 1, 2, \ldots$ obtemos:

$$\begin{aligned} \phi_{11} &= \rho_1 \\ \phi_{22} &= \frac{\begin{vmatrix} 1 & \rho_1 \\ \rho_1 & \rho_2 \end{vmatrix}}{\begin{vmatrix} 1 & \rho_1 \\ \rho_1 & 1 \end{vmatrix}} \\ \phi_{33} &= \frac{\begin{vmatrix} 1 & \rho_1 & \rho_1 \\ \rho_1 & 1 & \rho_2 \\ \rho_2 & \rho_1 & \rho_3 \end{vmatrix}}{\begin{vmatrix} 1 & \rho_1 & \rho_2 \\ \rho_1 & 1 & \rho_1 \\ \rho_2 & \rho_1 & 1 \end{vmatrix}} \\ &\vdots \end{aligned}$$

De modo geral, pode-se escrever:

$$\varphi_k := \phi_{kk} = \frac{|\mathbf{P}_k^*|}{|\mathbf{P}_k|} \tag{C.2}$$

Onde nota-se que $\mathbf{P}_k$ é a matriz de autocorrelações do modelo $AR(k)$, e $\mathbf{P}_k^*$ é a matriz $\mathbf{P}_k$ com a última coluna substituída pelo vetor de autocorrelações, que é o termo à direita da igualdade das equações de Yule-Walker (C.1).

Dessa forma, a definição acima de φ_k é chamada de *função de autocorrelação parcial* do *lag* k, ou seja, de ordem autorregressiva k. Ela pode ser entendida como a correlação parcial entre as variáveis Z_t e Z_{t-k}, ajustadas às variáveis intermediárias $Z_{t-1}, \dots, Z_{t-k+1}$. Isto é, ela mede a correlação remanescente entre Z_t e Z_{t-k} depois de removida a influência de $Z_{t-1}, \dots, Z_{t-k+1}$.

Referências

[Allen 2020] Robbie Allen. *A Gentle Introduction to Machine Learning Concepts*. https://medium.com/machine-learning-in-practice/a-gentle-introduction-to-machine-learning-concepts-cfe710910eb. Fev. de 2020 (citado nas pgs. 5–11).

[Alsmadi *et al.* 2009] M. k. Alsmadi, K. B. Omar, S. A. Noah e I. Almarashdah. "Performance comparison of multi-layer perceptron (back propagation, delta rule and perceptron) algorithms in neural networks". Em: *2009 IEEE International Advance Computing Conference* (mar. de 2009), pgs. 296–299. doi: 10.1109/IADCC.2009.4809024 (citado na pg. 3).

[Ballini 2000] Rosangela Ballini. "Análise e Previsão de Vazões Utilizando Modelos de Séries Temporais, Redes Neurais e Redes Neurais Nebulosas". Doutorado em Engenharia Elétrica. Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas, 2000 (citado na pg. 17).

[Barpaga 2019] Prince Barpaga. *A Gentle Introduction to Machine Learning*. https://towardsdatascience.com/a-gentle-introduction-to-machine-learning-599210ec34ad. Jun. de 2019 (citado na pg. 5).

[Blei e Smyth 2017] David M. Blei e Padhraic Smyth. "Science and data science". Em: *PNAS* 114.33 (ago. de 2017), pgs. 8689–8692 (citado na pg. 1).

[Box e Cox 1964] George Edward Pelham Box e David Roxbee Cox. "An analysis of transformations". Em: *Journal of the Royal Statistical Society, Series B (Methodological)* 26.2 (1964), pgs. 211–252. url: %5Curl%7Bhttps://www.jstor.org/stable/2984418%7D (citado na pg. 81).

[Box, Jenkins *et al.* 2016] George Edward Pelham Box, Gwilym Meirion Jenkins, Gregory C. Reinsel e Greta M. Ljung. *Time Series Analysis - Forecasting and Control*. 5º. John Wiley & Sons, 2016 (citado nas pgs. 65, 66, 72, 81, 82).

[Budhiraja 2016] Amar Budhiraja. *Dropout in (Deep) Machine learning*. https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5. Dez. de 2016 (citado na pg. 42).

[Clevert *et al.* 2015] Djork-Arné Clevert, Thomas Unterthiner e Sepp Hochreiter. "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)". Em: *arXiv e-prints*, arXiv:1511.07289 (nov. de 2015), arXiv:1511.07289. arXiv: 1511.07289 [cs.LG] (citado nas pgs. 34, 35).

[Dellinger 2019] James Dellinger. *Weight Initialization in Neural Networks: A Journey From the Basics to Kaiming*. https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79. Abr. de 2019 (citado na pg. 37).

[Facure 2017a] Matheus Facure. *Dificuldades no Treinamento de Redes Neurais - Examinando o problema de gradientes explodindo ou desvanecendo*. https://matheusfacure.github.io/2017/07/10/problemas-treinamento/. Jul. de 2017 (citado na pg. 32).

[Facure 2017b] Matheus Facure. *Funções de Ativação - Entendendo a importância da ativação correta nas redes neurais*. https://matheusfacure.github.io/2017/07/12/activ-func/. Jul. de 2017 (citado nas pgs. 32, 33, 35, 43).

[Géron 2019] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. 2º. O'Reilly, 2019 (citado nas pgs. 2, 5, 6, 12, 17, 20, 42, 44, 51, 75, 79).

[Grus 2016] Joel Grus. *Data Science do Zero*. 1º. O'Reilly, 2016 (citado nas pgs. 1, 2, 13–16, 30, 44).

[Guidorizzi 1986] Hamilton Luiz Guidorizzi. *Um curso de cálculo*. v. 2. LTC, 1986. isbn: 8521604254 (citado na pg. 77).

[Heaton 2017] Jeff Heaton. *The Number of Hidden Layers*. https://www.heatonresearch.com/2017/06/01/hidden-layers.html. Jun. de 2017 (citado nas pgs. 45, 46).

[Kopec 2019] David Kopec. *Problemas Clássicos de Ciência da Computação com Python*. 1º. Novatec, 2019 (citado nas pgs. 3, 18, 20, 22, 25, 28, 75).

[Korbut 2017] Daniil Korbut. *Machine Learning Algorithms: Which One to Choose for Your Problem*. https://blog.statsbot.co/machine-learning-algorithms-183cc73197c. Out. de 2017 (citado nas pgs. 12, 16).

[Magalhães e Lima 2002] Marcos Nascimento Magalhães e Antonio Carlos Pedroso de Lima. *Noções de Probabilidade e Estatística*. 5º. Edusp, 2002 (citado na pg. 11).

[Al-Masri 2019] Anas Al-Masri. *What Are Overfitting and Underfitting in Machine Learning?* https://towardsdatascience.com/what-are-overfitting-and-underfitting-in-machine-learning-a96b30864690. Jun. de 2019 (citado na pg. 42).

[Morettin e Castro Toloi 2019] Pedro Alberto Morettin e Clélia Maria de Castro Toloi. *Análise de séries temporais, vol. 1: Modelos lineares univariados*. 3º. Edgard Blücher Ltda., 2019 (citado nas pgs. 57–67, 69–72, 81, 83).

[Morettin e Motta Singer 2020] Pedro Alberto Morettin e Julio da Motta Singer. *Introdução à Ciência de Dados - Fundamentos e Aplicações*. Departamento de Estatística. Universidade de São Paulo, 2020 (citado na pg. 1).

[Rosenblatt 1958] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain". Em: *Psychological Review* 65.6 (1958), pgs. 386–408. url: %5Curl%7Bhttps://www.ling.upenn.edu/courses/cogs501/Rosenblatt1958.pdf%7D (citado na pg. 18).

[Silver 2018] Andrew Silver. *The Essential Data Science Venn Diagram*. https://towardsdatascience.com/the-essential-data-science-venn-diagram-35800c3bef40. Set. de 2018 (citado nas pgs. 1, 2).

[Xu *et al.* 2015] Bing Xu, Naiyan Wang, Tianqi Chen e Mu Li. "Empirical Evaluation of Rectified Activations in Convolutional Network". Em: *arXiv e-prints*, arXiv:1505.00853 (mai. de 2015), arXiv:1505.00853. arXiv: 1505.00853 [cs.LG] (citado nas pgs. 33–35).

[Ye 2020] Andre Ye. *AutoML: Creating Top-Performing Neural Networks Without Defining Architectures*. https://towardsdatascience.com/automl-creating-top-performing-neural-networks-without-defining-architectures-c7d3b08cddc. Set. de 2020 (citado na pg. 56).