Universidade de Brasília

Análise do Desempenho de Redes Neurais Artificiais em Problemas de Classificação Multiclasse

Raquel Souza Carvalho

Departamento de Matemática
Universidade de Brasília

Dissertação apresentada como requisito parcial para obtenção do grau de *Mestra em Matemática*

Brasília, 31 de julho de 2024

UNIVERSIDADE DE BRASÍLIA

DEPARTAMENTO DE MATEMÁTICA

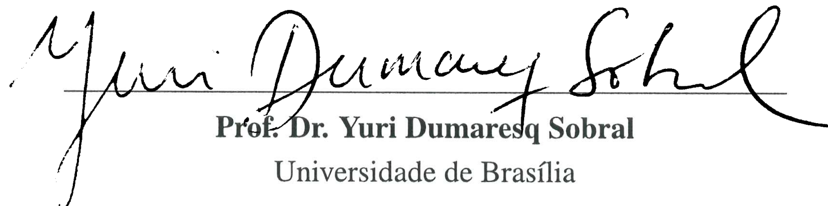
PROGRAMA DE PÓS-GRADUAÇÃO EM MATEMÁTICA

ANÁLISE DO DESEMPENHO DE REDES NEURAIS ARTIFICIAIS EM PROBLEMAS DE CLASSIFICAÇÃO MULTICLASSE

Raquel Souza Carvalho*

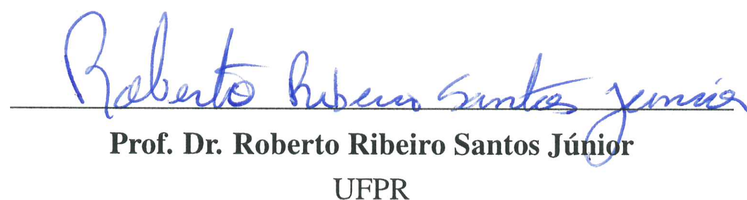
Dissertação apresentada como requisito parcial para obtenção do grau de Mestra em Matemática

Banca Examinadora:



Prof. Dr. Yuri Dumaresq Sobral
Universidade de Brasília

Prof. Dr. Vinícius de Carvalho Rispoli
Universidade de Brasília



Prof. Dr. Roberto Ribeiro Santos Júnior
UFPR

Brasília, 31 de julho de 2024

A autora foi bolsista CNPq durante a fase inicial da elaboração do trabalho.*

Com muito carinho, gratidão e orgulho, dedico este trabalho às pessoas que, com seu apoio incondicional, fizeram desta jornada uma travessia memorável. Dedico especialmente, a todas a mulheres da minha vida que tiveram suas asas cortadas, mas nunca deixaram de me ensinar a voar. Jamais esquecerei de ninguém que fez parte deste percurso.

Agradecimentos

Gostaria de expressar minha profunda gratidão a todas as pessoas que me apoiaram nesta jornada.

Primeiramente a Deus, pelo sustento e benevolência comigo.

À minha mãe, Sandra, por ser o meu porto seguro, minha fonte de vida e perseverança. Seu apoio e amor incondicional sempre será o farol da minha vida.

À Ely, por seu amor inabalável, paciência e incentivo constante. Você é minha maior fonte de força e inspiração, eu não chegaria até aqui sem você.

Ao meu orientador, Dr. Yuri, um grande pesquisador que, além de ser um excelente profissional, transforma vidas através do seu trabalho.

À minha família, por estar sempre ao meu lado, oferecendo suporte e amor nos momentos mais desafiadores. Em especial, à Maria José, Nivaldo Marcelo, Luciano Carvalho, Emanuela Carvalho e Darlan Azevedo, que sempre zelaram pelo meu bem-estar. E o meu amor, que está sempre comigo no coração, minha avó Ormezilda.

A um querido amigo, Rogério, pelo seu apoio constante e incentivo.

Ao IBPAD, minha sincera gratidão por proporcionar um ambiente fértil de crescimento e aprendizado, além de disponibilizar os recursos necessários para a realização deste trabalho. Um agradecimento especial ao Max, Jaque e Pira, pelo suporte constante e orientação valiosa. Agradeço também a todos os colegas pelas conversas enriquecedoras e pela ajuda nos momentos desafiadores, com destaque para Amilcar e Pedro.

Ao colega Luis F. Cury, pelo auxílio no direcionamento do desenvolvimento de algoritmos no ínicio da elaboração do trabalho.

Aos membros da banca examinadora.

Ao Departamento de Matemática da Universidade de Brasília.

À Universidade de Brasília por proporcionar a realização de um sonho.

Este trabalho é um reflexo de muita persistência e força de vontade. Muito obrigado por fazerem parte desta trajetória.

Resumo

Este trabalho aborda a aplicação de redes neurais para problemas de classificação multiclasse, explorando e comparando o desempenho de múltiplas arquiteturas. Inicialmente, são apresentados os fundamentos matemáticos das redes neurais, incluindo a definição de neurônios artificiais, funções de ativação e o processo de treinamento utilizando retropropagação. Define-se arquitetura, neste trabalho, como a combinação da quantidade de camadas ocultas e de neurônios por camadas da rede. Utilizando um conjunto de dados padronizado para classificação multiclasse, e fixando os hiperparâmetros, as redes neurais são treinadas e testadas, e as taxas de acurácia são avaliadas de forma estatística. Os resultados demonstram que arquiteturas com maior disparidade de dimensionalidade entre as camadas tendem a ter um desempenho pior em comparação com as redes que possuem arquiteturas com distribuição de neurônios mais uniformes. Por outro lado, redes que tenham disparidade significativa em sua arquitetura podem alcançar bom índice de acurácia, desde que seu treinamento seja feito de maneira diferenciada. A análise detalhada dos resultados permite identificar algumas das configurações que tendem a exprimir precisão e eficiência. Conclui-se que a escolha da arquitetura ideal, que normalmente depende das especificidades do problema e dos recursos disponíveis, pode ser determinante no desempenho da rede e, portanto, destaca-se a importância de uma abordagem experimental na otimização de redes neurais para classificação multiclasse.

Palavras-Chave: Redes Neurais Artificiais, Aprendizado de Máquina, Inteligência Artificial, Arquitetura de Redes Neurais, Desempenho de Redes Neurais.

Abstract

This work addresses the application of neural networks to multiclass classification problems, exploring and comparing the performance of multiple architectures. Initially, the mathematical foundations of neural networks are presented, including the definition of artificial neurons, activation functions and the training process using backpropagation. Architecture is defined, in this work, as the combination of the number of hidden layers and neurons per layer of the network. Using a standardized dataset for multiclass classification, and fixing the hyperparameters, neural networks are trained and tested, and accuracy rates are statistically evaluated. The results demonstrate that architectures with greater disparity in dimensionality between layers tend to have worse performance compared to networks that have architectures with more uniform neuron distribution. On the other hand, networks that have significant disparities in their architecture can achieve a good accuracy rate, as long as their training is carried out differently. Detailed analysis of the results allows us to identify some of the configurations that tend to express precision and efficiency. It is concluded that the choice of the ideal architecture, which usually depends on the specifics of the problem and the available resources, can be decisive in the performance of the network and, therefore, the importance of an experimental approach in optimizing neural networks for multiclass classification is highlighted.

Keywords: Artificial Neural Networks, Machine Learning, Artificial Intelligence, Neural Network Architecture, Neural Network Performance.

Conteúdo

Lista de Tabelas ix

Lista de Figuras x

Lista de Códigos Fonte xii

Introdução 1

1 Otimização de Ajustes com o Aprendizado de Máquina 4

1.1 Ajuste Linear e o Método dos Mínimos Quadrados 4

1.2 Ajuste Não Linear e o Método do Gradiente Descendente 15

1.2.1 O Método do Gradiente Descendente 15

1.2.2 Função de Custo 17

1.2.3 Hiperparâmetros 18

1.2.4 Tipos de Gradiente Descendente 19

1.2.5 Implementação do Gradiente Descendente 21

2 Redes Neurais Artificiais 29

2.1 Estrutura Elementar 29

2.1.1 Alguns Tipos de Redes Neurais 33

2.2 Propagação e Retropropagação 36

2.3 Implementação em Python de Rede Neural Feedforward 41

3 Análise do Desempenho Redes Neurais Artificiais 51

3.1 Descrição do Problema 52

3.1.1 Base de Dados 52

3.1.2 Escolha da Estrutura das Redes Neurais Artificiais 53

3.1.3 Arquitetura das Redes Neurais Artificiais 53

3.1.4 Implementação Computacional 55

3.2 Análise da Acurácia 67
3.3 Análise da Acurácia em Relação a Disparidade de Dimensionalidade 73
3.4 Análise de Modelos com Baixo Desempenho 78

Conclusão **83**

Bibliografia **84**

Lista de Tabelas

3.1 Parâmetros Fixos de Treinamento. 68
3.2 Tabela de desempenho de modelos. 69
3.3 Acurácia das redes neurais sem contração nem expansão. 75
3.4 Dados de Treinamento dos Novos Modelos - Redes com Baixo Desempenho 79
3.5 Dados da Acurácia dos Novos Modelos 82

Lista de Figuras

1 Imagem adaptada das arquiteturas de redes neurais artificiais. Obtida originalmente em THE ASIMOV INSTITUTE (2019). 2

1.1 Exemplo de um conjunto de dados relacionando uma variável dependente a uma variável independente. 5
1.2 Ajuste linear obtido a partir do método dos mínimos quadrados 13
1.3 Esquema ilustrativo do método do gradiente descendente para diferentes taxas de aprendizado. 18
1.4 Dispersão de Dados Não Linear 25
1.5 Comparação do ajuste não linear com gradiente descendente e ajuste linear com método dos mínimos quadrados. 26
1.6 Comparação de ajustes não lineares obtidos através do gradiente descendente com e sem recursividade e o método dos mínimos quadrados. 27
1.7 Evolução do custo nos treinamentos do gradiente descendente com e sem recursividade nos ajustes não lineares. 28

2.1 Imagem adaptada da ilustração de um neurônio humano. Obtida originalmente em [1]. 30
2.2 Ilustração de uma rede neural artificial com camadas densas. 32
2.3 Ilustração de uma rede neural artificial feedforward. 34
2.4 Conjunto de dados gerados pela Listagem 2.3.1. 42
2.5 Ilustração da rede neural artificial implementada neste trabalho com uma camada oculta contendo quatro neurônios. 43
2.6 Ilustração da aplicação da função de ativação em uma rede neural artificial. 43
2.7 Ilustração de uma rede neural artificial com uma camada oculta contendo quatro neurônios com camada de saída encodificada. 44
2.8 Evolução do custo ao longo das épocas para as redes definidas nas Listagens [2.3.3, 2.3.4]. 49

3.1 Evolução da acurácia média de todos os treinamentos ao longo das épocas. . 68
3.2 Evolução da acurácia média do treinamento de modelos considerados com bom desempenho ao longo das épocas. 70
3.3 Evolução da acurácia ao longo do treinamento de modelos com melhor, pior e intermediária performance. 70
3.4 Média da acurácia com base na quantidade de neurônios das redes com barra de erro do desvião padrão. 71
3.5 Média da acurácia com base na quantidade de parâmetros. 72
3.6 Média da acurácia em relação ao tamanho de maior disparidade dimensional das redes com barra de erro do desvio padrão. 75
3.7 Média da acurácia em relação ao tamanho das lacunas na primeira camada oculta das redes neurais treinadas. 76
3.8 Média da acurácia em relação ao tamanho das lacunas na camada de saída das redes neurais treinadas. 76
3.9 Média da acurácia em relação ao tipo de disparidade predominante nas extremidade das redes neurais treinadas. 77
3.10 Comparação da Acurácia de Modelos com Hiperparâmetros ajustados. . . . 81

Lista de Códigos Fonte

1.1.1 Algoritmo do Método dos Mínimos Quadrados - Definição Conjuntos . . . 11
1.1.2 Algoritmo do Método dos Mínimos Quadrados 11
1.2.1 Parte I - Algoritmo do Gradiente Descendente 22
1.2.2 Parte II - Algoritmo do Gradiente Descendente 23
1.2.3 Parte III - Algoritmo do Gradiente Descendente com Momento 24
2.3.1 Rede Neural Feedforward Sem Camada Oculta - Gerando conjunto de dados 41
2.3.2 Rede Neural Fedforward Sem Camada Oculta - Definindo Funções e Conjunto de Treino e Teste 44
2.3.3 Rede Neural Feedforward Sem Camada Oculta - Algoritmo de Treinamento 46
2.3.4 Rede Neural Feedforward Com Camada Oculta - Algoritmo de Treinamento 47
3.1.1 Gerando Conjunto de Arquitetura das Camadas Ocultas 55
3.1.2 Parte I - Exemplificando Uso do Class 57
3.1.3 Parte II - Exemplificando Uso do Class 57
3.1.4 Parte III - Exemplificando Uso do Class 57
3.1.5 Parte IV - Exemplificando Uso do Class 58
3.1.6 Parte V - Exemplificando Uso do Class 58
3.1.7 Parte VI - Exemplificando Uso do Class 58
3.1.8 Definindo Class RNA 59
3.1.9 Definindo Função de Treinamento 61
3.1.10 Definindo Função Para Calcular Acurácia 64
3.1.11 Definindo Função Para Treinar Múltiplas Redes 65
3.1.12 Definindo Configurações de Treinamento das Redes 66

Introdução

Dentro do campo de aprendizado de máquinas, existe um conceito que é intríseco a vários modelos de inteligência artificial, chamado de redes neurais artificiais, como descrito em [2]. Essas redes são estruturas matemáticas e computacionais inspiradas na estrutura do cérebro humano, sendo compostas por camadas de neurônios artificiais, que processam dados em estágios sequenciais e, quando combinadas, permitem a compreensão de conjuntos de dados com grande volume de dados, como explica [3].

As redes neurais artificiais começaram a ser teoricamente desenvolvidas com o trabalho de Warren McCulloch e Walter Pitts em 1943, que publicaram um artigo seminal sobre o funcionamento de neurônios e modelaram uma rede utilizando circuitos elétricos [4]. Mas foi no final da década de 50 que houve uma inovação prática idealizada por Frank Rosenblatt em 1958, com a definição de um perceptron [5]. O perceptron, que é uma rede neural artificial com uma única camada, foi uma das primeiras tentativas de simular a forma como os neurônios humanos processam informações, se tornando o grande marco inicial do campo. Este modelo inicial, embora simples, abriu caminho para avanços significativos nas décadas seguintes.

Nos anos seguintes, pesquisadores continuaram a desenvolver e aperfeiçoar as redes neurais, por meio da criação de algoritmos mais sofisticados e eficientes. Um marco importante foi a introdução do algoritmo de retropropagação nos anos 1980, descrita com detalhes em [6]. Isto permitiu o treinamento eficaz de redes neurais multicamadas. Este avanço foi crucial para a popularização das redes neurais e seu uso em diversas aplicações práticas, fazendo surgir o conceito da arquitetura de uma rede neural. Esse desenvolvimento fez surgir novas possibilidades para resolução de problemas não-lineares que exigiam, até então, teorias muito específicas para implementar uma possível solução.

Neste trabalho, as arquiteturas de redes neurais referem-se à forma como as camadas e os neurônios são organizados dentro da rede, como mostra a Figura 1, adequando-se para diferentes tipos de problemas. As bolas na cor amarela, representam os neurônios da camada de entrada, as de cor laranja representam os da camada de saída e as verdes representam os neurônios das camadas ocultas. Um perceptron é capaz de resolver problemas básicos de

classificação binária, já uma rede neural com múltiplas camadas, pode lidar com problemas mais complexos, como problemas de classificação multiclasse, reconhecimento de imagens, tradução automática de idiomas, entre outros. A escolha da arquitetura depende muito da natureza e complexidade do problema a ser resolvido, como pode ser visto na imagem adaptada obtida em [7].

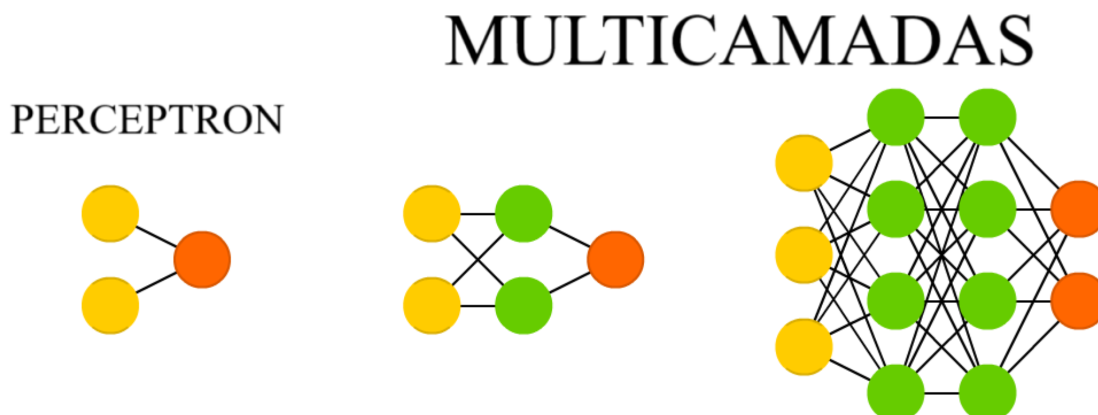


Figura 1 Imagem adaptada das arquiteturas de redes neurais artificiais. Obtida originalmente em THE ASIMOV INSTITUTE (2019).

Ao longo das décadas, a evolução das redes neurais foi impulsionada tanto pelo aumento do poder computacional quanto pelo desenvolvimento de novos algoritmos. Por trás do aumento da capacidade computacional e da popularização das redes neurais, o formalismo matemático acompanhou este desenvolvimento e, hoje, já há diversos teoremas os quais garantem que redes neurais são capazes de aproximar funções matemáticas que descrevem relações complexas entre variáveis [8, 9]. Estes teoremas são chamados de Teorema da Aproximação Universal e uma de suas versões mais conhecida é a proposta por Cybenko [8], a qual afirma que qualquer função contínua pode ser aproximada por uma soma de funções do tipo sigmoide em um determinado intervalo.

Nos últimos anos, este campo vem ganhando grande notoriedade no meio acadêmico com o surgimento do campo da inteligência artificial, que utiliza redes neurais de forma intríseca para resolver problemas complexos de reconhecimento de padrões, processamento de linguagem natural, entre outros, que matematicamente se assemelham com poblemas não-lineares de alta complexidade.

O treinamento de redes neurais é um processo crucial que visa minimizar o erro entre as previsões da rede e as observações esperadas [10]. Este erro, conhecido como função de custo, é uma medida de quão bem a rede está performando. O objetivo é ajustar os parâmetros

da rede, ou seja, os pesos das conexões entre os neurônios, para que a função de custo seja minimizada.

Para esse processo de minimização, utiliza-se um método de otimização chamado gradiente descendente durante a retropropagação do erro, como descrito detalhamente em [6]. Este método envolve calcular o gradiente da função de custo em relação aos pesos da rede, o que indica a direção na qual os pesos devem ser ajustados para reduzir o erro. O processo é repetido iterativamente, de forma que em cada iteração a rede faz a previsão, o erro é calculado, e os pesos são ajustados de acordo com o gradiente descendente. Ao longo das iterações, esses ajustes sucessivos permitem que a rede encontre um conjunto de pesos que minimiza o erro, obtendo sucesso com base num critério de parada.

O presente trabalho visa analisar matematicamente o desempenho de diferentes arquiteturas de redes neurais aplicadas a problemas de classificação multiclasse. Para tanto, serão explorados diversos aspectos fundamentais, incluindo a definição e funcionamento dos neurônios artificiais, que se comportam como as unidades de processamento, as funções de ativação, que introduzem a não-linearidade nas saídas dos neurônios, os métodos de treinamento baseados na retropropagação do erro e por fim a análise de um novo conceito introduzido neste trabalho, o de disparidade dimensional da rede, que visa analisar como a distribuição de neurônios por camada afeta o desempenho da rede.

A classificação multiclasse, que envolve a categorização de amostras dentre várias classes possíveis, apresenta desafios específicos que demandam arquiteturas de redes neurais bem adaptadas [11]. Os resultados obtidos visam fornecer informações e análises sobre as práticas recomendadas para a implementação de redes neurais em cenários de classificação multiclasse, contribuindo para o avanço do conhecimento na área e oferecendo diretrizes para futuros trabalhos e aplicações práticas. Além disso, a análise comparativa das arquiteturas em relação às suas disparidades dimensionais permite um entendimento mais profundo das dinâmicas de aprendizado e das capacidades de generalização das redes neurais, aspectos cruciais para a sua aplicação eficaz em problemas reais.

No Capítulo 1, trataremos de definir e formalizar matematicamente uma parte da teoria de ajuste de curvas lineares e não lineares, englobando os conceitos citados aqui. No Capítulo 2, o foco será definir os principais conceitos de redes neurais artificiais, juntamente com a confecção de algortimos, cuja implementação é feita em python. Por fim, no Capítulo 3 definiremos o problema específico a ser analisado neste trabalho, implementaremos o algortimo de treinamento e realizaremos análises voltadas para as caracterizações da acurácia em relação a arquitetura das redes usando para isso o conceito de disparidade dimensional.

Capítulo 1

Otimização de Ajustes com o Aprendizado de Máquina

O avanço significativo no campo do aprendizado de máquina trouxe consigo uma crescente necessidade de compreender e aplicar técnicas eficazes de modelagem para extrair padrões e informações de conjuntos complexos de dados. No centro desse desafio encontra-se o ajuste de curvas, uma abordagem fundamental para encontrar relações matemáticas que melhor descrevem fenômenos observados. Paralelamente, o algoritmo do gradiente descendente emergiu como um protagonista vital na otimização de modelos.

Este capítulo aborda a relação entre o ajuste de curvas e o gradiente descendente, explorando como esses conceitos se entrelaçam para aprimorar a capacidade dos modelos preditivos no contexto de aprendizadem de máquina. Investigaremos as nuances do ajuste de curvas, e em seguida, adentraremos no domínio do gradiente descendente, delineando sua função vital na otimização de modelos não lineares por meio da minimização da função de custo. O capítulo será finalizado com objetivo de demonstrar como funciona o processo de aprendizado da máquina utilizando, para tal feito, o princípio da recursividade, que é um elemento essencial e intrínseco de qualquer tipo de aprendizado.

Na próxima seção, vamos discutir o ajuste linear com fundamentações teóricas baseadas em [12, 13]. Para uma melhor compreensão, é desejável que o leitor tenha noções de conceitos da álgebra linear, que podem ser encontrados em [14].

1.1 Ajuste Linear e o Método dos Mínimos Quadrados

No universo da análise de dados e aprendizado de máquina, o ajuste de curvas emerge como uma técnica crucial. Esta técnica se destina a modelar a relação entre variáveis

independentes e dependentes. Em essência, o ajuste de curvas é o processo de encontrar a função que melhor se adapta a um conjunto de dados, minimizando a diferença entre os valores previstos pela função e os valores observados nos dados [15]. Tal diferença costuma ser definida como erro, ou custo, como veremos posteriormente na Subseção 1.2.2.

Suponha que exista um conjunto de dados quaisquer, distribuídos em pares ordenados (x,y), onde uma entrada corresponde à variável dependente e a outra corresponda à variável independente, como mostrado na Figura 1.1.

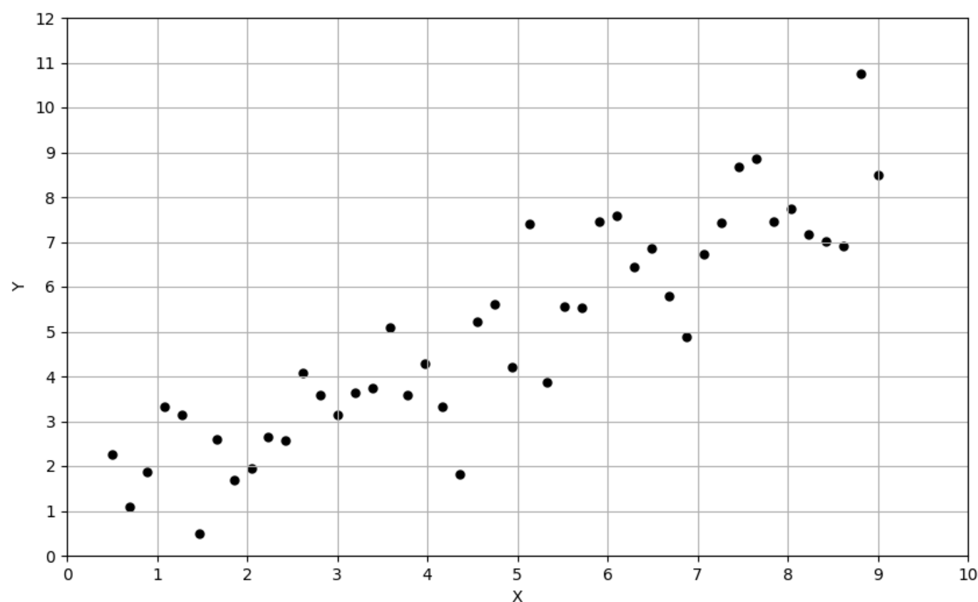


Figura 1.1 Exemplo de um conjunto de dados relacionando uma variável dependente a uma variável independente.

A primeira pergunta que podemos nos fazer é se há a possibilidade de determinar alguma função que consiga representar esses dados da melhor maneira possível. Olhando para a Figura 1.1, uma primeira tentativa seria um ajuste linear do tipo:

$$Y(x) = \theta + \beta \cdot x. \tag{1.1}$$

Obviamente, tal função não será capaz de descrever o conjunto de dados perfeitamente. Ainda assim, a função dada na Equação 1.1 fornece uma aproximação que muitas vezes pode ser útil, dependendo do problema. Diante disto, surge o questionamento de como determinar os melhores valores dos coeficientes que apresentam a melhor descrição do conjunto de

dados. A resposta para essa pergunta pode ser facilmente entendida usando álgebra linear matricial, como veremos a seguir.

Suponhamos que os coeficientes θ e β sejam conhecidos. Assim, podemos determinar o erro em cada ponto como sendo:

$$e_i = Y_i - Y_i^{previsto} = Y_i - (\theta + \beta \cdot x_i)\,. \tag{1.2}$$

Então, conhecendo-se um x_i, podemos determinar o valor de $Y_i^{previsto}$, que é a aproximação para o valor de Y_i calculada pela função dada na Equação 1.1.

Generalizando este conceito para os n pontos do conjunto de variáveis independentes, obtemos que:

$$e = \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ \vdots \\ e_n \end{bmatrix} = \begin{bmatrix} Y_1 - (\theta_1 + \beta \cdot x_1) \\ Y_2 - (\theta_2 + \beta \cdot x_2) \\ Y_3 - (\theta_3 + \beta \cdot x_3) \\ \vdots \\ Y_n - (\theta_n + \beta \cdot x_n) \end{bmatrix} = \underbrace{\begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ \vdots \\ Y_n \end{bmatrix}}_{b} - \underbrace{\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}}_{A} \cdot \underbrace{\begin{bmatrix} \theta \\ \beta \end{bmatrix}}_{x}, \tag{1.3}$$

em que $n > 2$ denota a quantidade de pares (x_i, Y_i).

E portanto, de forma mais compacta obtemos:

$$e = b - Ax, \tag{1.4}$$

com A e b identificados na Equação 1.3. Como temos apenas duas incógnitas e n equações, este tipo de sistema é chamado de sobredeterminado e nem sempre possui solução. Nesse caso, as equações impõem mais restrições do que as variáveis podem satisfazer simultaneamente. Isso significa que nem sempre é possível encontrar um conjunto de valores para as variáveis que satisfaça todas as equações ao mesmo tempo. Em outras palavras, o sistema pode ser inconsistente, resultando na inexistência de uma solução que atenda a todas as equações simultaneamente. Matematicamente, o sistema terá solução quando as equações são linearmente dependentes, ou seja, uma ou mais equações podem ser expressas como combinações lineares das outras.

Como não há maneira de resolver exatamente o sistema na Equação 1.4, podemos tentar determinar sua melhor solução que é aquela que minimiza a quantidade global do erro, isto é,

minimizando $|e|$, que pode ser escrita como:

$$|e| \overset{(1.4)}{=} |b - Ax| = \sqrt{\sum_{i=1}^{n} e_i^2} = \sqrt{e^T e}. \tag{1.5}$$

Sendo assim, o objetivo se torna determinar x tal qual $|e|$ seja minímo. Se obtivermos $Ax = b$, teremos a resolução do problema com erro global $|e| = 0$. Mas, como dito anteriormente, isto nem sempre é possível e é justamente para esses casos, que são os casos de interesse, que vamos buscar resolver o sistema com a melhor aproximação possível. Para realizar tal feito, devemos analisar o erro global mais profundamente. Desta forma:

$$\begin{aligned} |e|^2 &= |b - Ax|^2 \\ &\overset{(1.5)}{=} (b - Ax)^T (b - Ax) \\ &= (b^T - x^T A^T)(b - Ax) \\ &= b^T b - b^T Ax - x^T A^T b + x^T A^T Ax \\ &\overset{\star}{=} b^T b - x^T A^T b - x^T A^T b + x^T A^T Ax \\ &= b^T b - 2x^T A^T b + x^T A^T Ax \\ &= x^T A^T Ax - 2x^T A^T b + b^T b, \end{aligned} \tag{1.6}$$

em que o passo $\star$ é válido pelo fato de que $x^T A^T b = (b^T AX)^T$, e como $b^T Ax$ é uma matriz $1x1$, se comporta como um escalar, fazendo com que sua transposta seja ela mesmo. Com isso, obtemos que:

$$|e|^2 = x^T A^T Ax - 2x^T A^T b + b^T b. \tag{1.7}$$

Considere $K = A^T A$, $l = A^T b$ e $m = b^T b$. Veja que esta equação representa uma forma quadrática, em que K, l e m representam o conjunto de coeficientes do termo quadrático, linear e constante, respectivamente. Ainda falando sobre a Equação 1.7, note que ela pode ser generalizada do seguinte modo:

$$f(x_1, x_2, \ldots, x_n) = \sum_{i,j=1}^{n} K_{i,j} x_i x_j - 2 \sum_{i=1}^{n} l_i x_i + m, \tag{1.8}$$

em que K é simétrica por construção, visto que $K = A^T A$ e

$$\left(A^T A\right)^T = A^T \left(A^T\right)^T = A^T A. \tag{1.9}$$

Portanto, em notação matricial, temos que:

$$f = x^T Kx - 2x^T l + m. \tag{1.10}$$

Note que uma forma quadrática é uma extensão natural da função escalar quadrática. Se observamos atentamente, uma função escalar quadrática tem a seguinte expressão:

$$f(x) = ax^2 + bx + c, x \in \mathbb{R}, \tag{1.11}$$

que denota uma parabóla, que por sua vez, possui um mínimo se o termo de maior grau for estritamente positivo. Por analogia, isto significa que devemos ter $x^T Kx > 0$ para todo $x \in \mathbb{R}^n$. A seguir, utilizaremos essa informação para demonstrar um resultado que garante a possibilidade de encontrar um ajuste ótimo. Mas, antes disso, precisamos entender alguns conceitos.

Definição 1.1.0.1. *Dizemos que uma matriz K é positiva definida se ela for simétrica e se*

$$x^T Kx > 0, \forall x \neq 0 \in \mathbb{R}^n. \tag{1.12}$$

Dizemos que K é semi-positiva definida se

$$x^T Kx \geq 0, \forall x \in \mathbb{R}^n. \tag{1.13}$$

Definição 1.1.0.2. *Um número $\lambda \in \mathbb{R}$ é chamado de autovalor de uma matriz quadrada A se existe um vetor não nulo $v \in \mathbb{R}^n$ tal que $Av = \lambda v$. Este vetor v é chamado de autovetor de A correspondente ao autovalor λ.*

Proposição 1.1.0.3. *Seja A uma matriz de ordem n e $\{\lambda_1, \lambda_2, ..., \lambda_n\}$ os autovalores de A. Então,*

$$det(A) = \lambda_1 \cdot \lambda_2 \cdot \lambda_3 \ldots \lambda_n. \tag{1.14}$$

Demonstração. Dada uma matriz A cujos autovalores são $\{\lambda_1, \lambda_2, ..., \lambda_n\}$, seu polinômio característico é dado por

$$p(\gamma) = det(A - \gamma I) = (-1)^n \gamma^n + c_{n-1}\gamma^{n-1} \cdot \ldots \cdot c_1\gamma + c_0, \tag{1.15}$$

em que I é a matriz identidade de ordem n, γ é um valor real e $\{c_0, ..., c_n\}$ são coeficientes que dependem da matriz A. Pelo Teorema Fundamental da Álgebra, sabemos que esse polinômio possui n raízes complexas, contando com suas multiplicidades, que são exatamente os

autovalores da matriz A. Portanto, o polinômio característico pode ser fatorado como:

$$p(\gamma) = (-1)^n(\gamma - \lambda_1)(\gamma - \lambda_2)\ldots(\gamma - \lambda_n), \tag{1.16}$$

com autovalores não necessariamente distintos. Ademais, note que

$$det(A) = det(A - 0I) = p(0). \tag{1.17}$$

E portanto,

$$\begin{aligned} \det(A) &= p(0) \\ &= (-1)^n(-\lambda_1)(-\lambda_2)\cdots(-\lambda_n) \\ &= \lambda_1\lambda_2\cdots\lambda_n. \end{aligned} \tag{1.18}$$

■

Definição 1.1.0.4. *Dada uma matriz quadrada A de ordem n, ela é dita inversível quando existe uma matriz B, tal que*

$$AB = I_n = BA, \tag{1.19}$$

em que B é denotada por A^{-1} e I_n é a matriz identidade de ordem n.

Observação 1.1.0.5. *Note que se K é definida positiva, segue da definição que todos os seus autovalores são estritamente positivos. Daí, por consequência direta da Proposição 1.1.0.3, K é inversível.*

Agora, vamos demonstrar o resultado que vai nos ajudar a determinar θ e β que minimizam o erro global $|e|$.

Teorema 1.1.0.6. *Seja $f = x^T Kx - 2x^T l + m$ uma forma quadrática . Então f tem um único mínimo dado por $x^\star = K^{-1}l$, desde que K seja positiva definida.*

Demonstração. Vamos verificar quais são os valores x que são mínimo de f quando $l = Kx^\star$. Veja que,

$$\begin{aligned} f &= x^T Kx - 2x^T l + m \\ &= x^T Kx - 2x^T Kx^\star + m \\ &= x^T Kx - x^T Kx^\star - (x^T Kx^\star)^T + m \\ &= x^T K(x - x^\star) - (x^\star)^T K^T x + m \\ &= x^T K(x - x^\star) + \underbrace{((x^\star)^T Kx^\star - (x^\star)^T Kx^\star)}_{0} - (x^\star)^T K^T x + m \end{aligned} \tag{1.20}$$

$$\overset{(i)}{=} x^T K(x-x^\star) - (x^\star)^T K(x-x^\star) - (x^\star)^T K x^\star + m$$
$$= \underbrace{(x-x^\star)^T K(x-x^\star)}_{M} - (x^\star)^T K x^\star + m,$$

em que (i) é válido pois estamos assumindo que K é positiva definida, e em particular $K^T = K$. Além disso,

$$\underbrace{(x-x^\star)^T K(x-x^\star)}_{M} > 0 \tag{1.21}$$

é válido pelo mesmo motivo. Com isso, temos que M só será zero, isto é, f alcançará o seu mínimo, quando $(x-x^\star) = 0$, ou seja,

$$x = x^\star = K^{-1}L. \tag{1.22}$$

E como os demais termos não dependem de x, eles não afetam o mínimo de f. ■

Observação 1.1.0.7. *Note que se K fosse apenas semi-positiva definida, então o mínimo não seria único.*

Agora, podemos determinar a solução que minimiza o erro global $|e|$ por meio da Equação 1.7, bastando para isso, verificarmos que $K = A^T A$ é definida positiva. Mas, pela Equação 1.9, sabemos que ela é simétrica e além disso, veja que

$$x^T A^T A x = (Ax)^T (Ax) \overset{(ii)}{=} |Ax|^2 \geq 0, \forall x \in \mathbb{R}, \tag{1.23}$$

em que (ii) é válida pois $(Ax)^T(Ax)$ representa o produto escalar de Ax com si mesmo, o qual é sempre não negativo. Portanto, $K = A^T A$ é sempre ao menos semi-positiva definida e, no caso de $\ker(A) = 0$, será positiva definida, visto que para todo x não nulo, Ax não será o vetor nulo e consequentemente $(Ax)^T(Ax)$ será estritamente positivo.

Observação 1.1.0.8. *A matriz $K = A^T A$ é chamada de Matriz de Gram.*

Observação 1.1.0.9. *A Matriz de Gram será inversível desde que as suas colunas sejam linearmente independentes, o que é o mesmo que dizer que* $\ker(A) = 0$. *Diante do que vimos acima, podemos afirmar que neste caso ela será positiva definida e pela Observação 1.1.0.5, consequentemente inversível.*

Finalmente, concluímos que a solução que minimiza o erro global é

$$x^\star = K^{-1} l = (A^T A)^{-1} A^T b. \tag{1.24}$$

De fato, isto equivale a resolver o sistema $Ax = b$, definido ao lado direito da Equação 1.4, que pode ser regularizado para determinar θ e β da melhor forma, fazendo:

$$
\begin{aligned}
Ax = b &\Leftrightarrow \\
A^T Ax = A^T b &\Leftrightarrow \\
x = (A^T A)^{-1} A^T b &= x^{\star}.
\end{aligned} \tag{1.25}
$$

Este método nada mais é do que o método dos mínimos quadrados, que pode ser encontrado em [13].

Vamos aplicar este método computacionalmente com auxílio do Python, com objetivo de determinar os paramêtros de um ajuste linear, como na Equação 1.1, para o conjunto de dados disposto na Figura 1.1.

Primeiramente, é necessário definir o algoritmo que irá gerar o conjunto de dados e as matrizes A e b, como mostra a Listagem 1.1.1.

Listagem 1.1.1 Algoritmo do Método dos Mínimos Quadrados - Definição Conjuntos

```
# Importando biblioteca
import numpy as np

# Definindo semente para ter reprodutibilidade
np.random.seed(0)

# Gera 45 pontos igualmente espaçados no intervalo de 0.5 a 9.
x = np.linspace(0.5, 9, 45).reshape(-1, 1)

# Gera a matriz A
coluna_1 = np.ones((num_linhas, 1))
matriz_A = np.hstack((coluna_1, coluna_x))

# Gera 45 valores de y, onde cada y éigual ao valor correspondente de x mais um ruído
    normal, com média 0 e desvio padrão 1.
matriz_b = (x + np.random.normal(0, 1, 45)).reshape(-1, 1)
```

Estando a matriz A e b devidamente definidas como matriz_A e matriz_b respectivamente, uma implementação em Python do método dos mínimos quadrados dado na Equação 1.25 pode ser dada por:

Listagem 1.1.2 Algoritmo do Método dos Mínimos Quadrados

```python
# Função para calcular a inversa de uma matriz 2x2
def inversa_matriz_2x2(matriz):
    a = matriz[0][0]
    b = matriz[0][1]
    c = matriz[1][0]
    d = matriz[1][1]
    determinante = a * d - b * c
    inversa_determinante = 1 / determinante
    return [[d * inversa_determinante, -b * inversa_determinante],
            [-c * inversa_determinante, a * inversa_determinante]]

# Define a matriz transposta de A
matriz_A_transposta = matriz_A.T

# Realiza o produto da matriz transposta de A com A
produto_AT_A = np.dot(matriz_A_transposta, matriz_A)

# Realiza o produto da matriz transposta de A com b
produto_AT_b = np.dot(matriz_A_transposta, matriz_b)

# Resolve o sistema considerando produto_AT_A como a matriz dos coeficientes e
    produto_At_b como a matriz das variáveis dependentes
produto_final = inversa_matriz_2x2(produto_At_A)
parametros = np.dot(produto_final, produto_At_b)

# Obtém os parâmetros da solução encontrada
alpha = float(vetor_x[0, 0])
beta = float(vetor_x[1, 0])
```

Arrendondando a solução em três casas decimais, obtemos aproximadamente:

$$\theta = 0.910 \text{ e } \beta = 0.854,\text{[1]} \tag{1.26}$$

e, assim, o ajuste linear pode ser traçado em conjunto com os dados, como na Figura 1.2.

[1] Nesta seção e nas seções subsequentes, será utilizado o arrendodamento dos valores em três casas decimais.

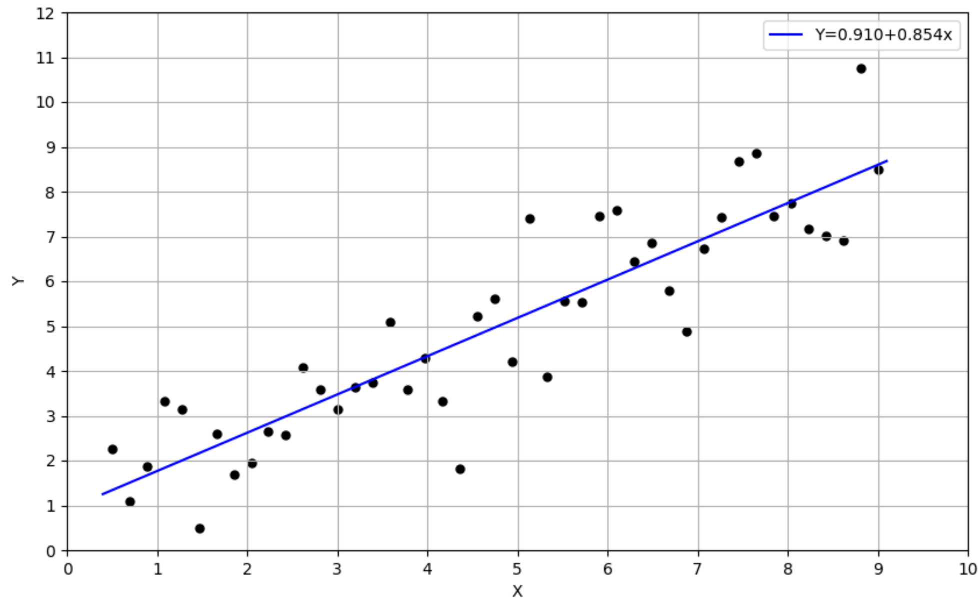


Figura 1.2 Ajuste linear obtido a partir do método dos mínimos quadrados

Quando falamos de ajustes de dados, não é muito comum nos referirmos ao valor do erro como na Equação 1.5, pois não há muita informação neste valor para além de sabermos que é mínimo. Neste caso em particular, o valor foi de $|e|^2 \approx 51.159$. Nos capítulos seguintes, o valor do erro quadrático, que aparecerá com uma nova definição, poderá ser usada de forma comparativa entre diferentes ajustes.

É possível propor uma generalização do método dos mínimos quadrados para qualquer conjunto de funções. Considere um conjunto qualquer de funções

$$H = \{h_1(x), h_2(x), h_3(x), ..., h_m(x)\}, \tag{1.27}$$

tais que:

$$\begin{aligned} Y_1 &= \theta_1 h_1(x_1) + \theta_2 h_2(x_1) + \theta_3 h_3(x_1) + \cdots + \theta_m h_m(x_1) \\ Y_2 &= \theta_1 h_1(x_2) + \theta_2 h_2(x_2) + \theta_3 h_3(x_2) + \cdots + \theta_m h_m(x_2) \\ Y_3 &= \theta_1 h_1(x_3) + \theta_2 h_2(x_3) + \theta_3 h_3(x_3) + \cdots + \theta_m h_m(x_3) \\ &\vdots \\ Y_n &= \theta_1 h_1(x_n) + \theta_2 h_2(x_n) + \theta_3 h_3(x_n) + \cdots + \theta_m h_m(x_n). \end{aligned} \tag{1.28}$$

A partir disto, podemos construir o sistema $Ax = b$, que será dado como a seguir.

$$\begin{bmatrix} h_1(x_1) & h_2(x_1) & \cdots & h_m(x_1) \\ h_1(x_2) & h_2(x_2) & \cdots & h_m(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ h_1(x_n) & h_2(x_n) & \cdots & h_m(x_n) \end{bmatrix}_{n\times m} \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_m \end{bmatrix}_{m\times 1} = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{bmatrix}_{n\times 1} . \tag{1.29}$$

De maneira totalmente análoga aos processos descritos até aqui, encontraremos os melhores valores de $\theta_1,\ldots,\theta_m$ através da resolução do sistema regularizado:

$$A^T Ax = A^T b, \tag{1.30}$$

em que K é a Matriz de Gram, $K = A^t A$, e $l = A^T b$.

Sendo assim, após regularizar o Sistema 1.29, obtemos que:

$$\begin{bmatrix} \sum_{k=1}^{n} h_1(x_k)h_1(x_k) & \sum_{k=1}^{n} h_1(x_k)h_2(x_k) & \cdots & \sum_{k=1}^{n} h_1(x_k)h_m(x_k) \\ \sum_{k=1}^{n} h_2(x_k)h_1(x_k) & \sum_{k=1}^{n} h_2(x_k)h_2(x_k) & \cdots & \sum_{k=1}^{n} h_2(x_k)h_m(x_k) \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{k=1}^{n} h_m(x_k)h_1(x_k) & \sum_{k=1}^{n} h_m(x_k)h_2(x_k) & \cdots & \sum_{k=1}^{n} h_m(x_k)h_m(x_k) \end{bmatrix} \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_m \end{bmatrix} = \begin{bmatrix} \sum_{k=1}^{n} h_1(x_k)Y_k \\ \sum_{k=1}^{n} h_2(x_k)Y_k \\ \vdots \\ \sum_{k=1}^{n} h_m(x_k)Y_k \end{bmatrix} . \tag{1.31}$$

Este é um sistema linear que pode ser facilmente resolvido numericamente, como já mencionado nesta seção. Note, porém, que este sistema deve ser cuidadosamente analisado, visto que dependendo dos tipos das funções de H, é possível que este sistema não seja linearmente independente, ainda que todas as suas variáveis independentes sejam distintas entre si.

Diante de tudo que foi posto até aqui, é possível perceber que os coeficientes a serem determinados pelo método dos mínimos quadrados podem surgir em várias posições na combinação linear que caracteriza o ajuste, isto é, podemos ter algo do tipo:

$$Y_i = \theta_1 h_1(\beta_1 x_i) + \theta_2 h_2(\beta_2 x_i) + \theta_3 h_3(\beta_3 x_i) + \cdots + \theta_m h_m(\beta_m x_i). \tag{1.32}$$

Neste caso, o sistema a ser resolvido para determinar θ_i e β_i, $i = 1,\ldots,m$, é não linear e sobredeterminado, o que o torna muito complicado.

É preciso, portanto, encontrar uma maneira alternativa para resolver o problema de encontrar os coeficientes do ajuste da Equação 1.32 sem termos que passar pelas complicações que o método dos mínimos quadrados, e a necessidade de se encontrar a solução de sistemas

não lineares, trazem. Uma das maneiras mais utilizadas para tal é justamente o método do gradiente descendente, que transforma o problema de resolver o sistema não linear para os coeficientes θ_i e β_i em um problema de otimização. Este método será descrito na próxima seção.

1.2 Ajuste Não Linear e o Método do Gradiente Descendente

A não linearidade é um ponto de bastante atenção quando o assunto é ajuste de curvas, isto porque a maioria dos problemas reais são não lineares. Resolver sistemas não lineares numericamente é uma tarefa muito difícil, em particular, utilizando-se processos iterativos que dependem de técnicas de álgebra linear, como o conhecido Método de Newton-Raphson [13].

O problema se torna ainda mais desafiador quando é necessário determinar parâmetros que estão impostos a recursividade, que é um método matemático onde uma função faz referência a si mesma, como pode ser visto em [16]. Porém, é possível utilizar conceitos do Cálculo, como o de gradiente ou derivada de uma função, para minimizar o erro global do ajuste através de um processo iterativo. Um exemplo de tal processo é chamado de método do gradiente descendente, descrito na Definição 1.2.4.1 e em [17].

O método do gradiente descendente é fundamental na otimização matemática, amplamente utilizado na estatística e também na computação com o aprendizado de máquina, como mostra [2]. O método conhecido atualmente começou a ser formalmente utilizado em 1847 por Augustin-Louis Cauchy [18], que o usou para minimizar uma função de várias variáveis e que vem sendo aprimorado por outros pesquisadores até os dias atuais. Contudo, o conceito de usar o gradiente, ou derivada, de uma função para encontrar seu mínimo local teve os seus primórdios no final do século XVII, sendo implementado inicialmente por Isaac Newton e Gottfried Wilhelm Leibniz. Com o avanço tecnológico, o gradiente descendente vem ganhando grande notoriedade na área da tecnologia. Mais detalhes sobre os primórdios deste método, podem ser encontrados em arquivos de meados do século XIX [19].

Na seção a seguir, apresentaremos este método em detalhes com base nas fundamentações científicas do livro [17].

1.2.1 O Método do Gradiente Descendente

Para entender o gradiente descendente, podemos imaginar o problema de nos colocarmos em uma superfície montanhosa e que nosso objetivo seja descer até o ponto mais baixo dessa

montanha. Matematicamente, isso corresponde à minimização de uma função. O gradiente da função em um determinado ponto é um vetor que aponta na direção de maior aumento da função. Portanto, para minimizar a função, deve ser levado em consideração o negativo do gradiente. Isto é o mesmo que nos mover na direção oposta ao gradiente, ou seja, descemos a montanha.

Nessa seção, vamos definir formalmente o método do gradiente descendente e seus conceitos elementares, iniciando pelo ponto mais básico.

Definição 1.2.1.1. *(Vetor Gradiente) Considere uma função $f : \mathbb{R}^n \longrightarrow \mathbb{R}$. O vetor gradiente de f em um ponto x de $\mathbb{R}^n$ é denotado por $\bigtriangledown f(x)$ e é definido como sendo*

$$\bigtriangledown f(x) = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right). \tag{1.33}$$

Aqui, $\frac{\partial f}{\partial x_i}$ representa a derivada parcial de f em relação à variável x_i. Cada componente do vetor gradiente é, portanto, a taxa de variação de f na direção de cada uma das variáveis independentes.

Observação 1.2.1.2. *É possível mostrar que, a partir da Definição 1.2.1.1, o vetor gradiente é um vetor que indica a direção de maior aumento da função f nesse ponto e cujo comprimento representa a taxa desse aumento. Mais detalhes podem ser vistos em [19].*

Com base na Definição 1.2.1.1, podemos definir formalmente o que é o método do gradiente descendente.

Definição 1.2.1.3. *(Método do Gradiente Descendente) Seja $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ uma função a ser minimizada. O método do gradiente descendente consiste em encontrar o ponto $x^\star \in \mathbb{R}^n$ tal que $f(x^\star)$ seja o mínimo de f. Essa tarefa é realizada atualizando repetidamente o ponto inicial x na direção oposta ao gradiente de f nesse ponto. Isto é, a cada iteração, x é atualizado da seguinte maneira:*

$$x_{k+1} = x_k - \alpha \bigtriangledown f(x_k), \tag{1.34}$$

em que α é o "tamanho" do passo a cada iteração.

Observação 1.2.1.4. *As iterações do método do gradiente descendente podem se repetir até que um critério de parada seja atingido.*

Para resolução de problemas, existem alguns conceitos importantes associados ao gradiente descendente que deve ser explorado. Na próxima seção, detalharemos um dos principais, que é o de função de custo.

1.2.2 Função de Custo

De acordo com [2], podemos introduzir alguns conceitos fundamentais inerentes ao método do gradiente descendente. Um dos mais básicos e principais, é a função de custo, definida a seguir.

Definição 1.2.2.1. *A função de custo, denotada por $\mathcal{C}$, é uma função que agrega os erros individuais em todo o conjunto de dados, transformando-os em uma única medida de desempenho.*

Atualmente, existem algumas funções de custo que são comumente utilizadas. Veremos algumas delas nos exemplos a seguir.

Exemplo 1.2.2.2. *(MAE) A função de custo "Erro Absoluto Médio" ou simplesmente MAE, do inglês "Mean Absolute Error", é dada por:*

$$\mathbf{C} = \frac{1}{n}\sum_{i=1}^{n} |Y_i - Y_i^{previsto}|. \tag{1.35}$$

Exemplo 1.2.2.3. *(MSE) A função de custo "Erro Quadrático Médio" ou simplesmente MSE, do inglês "Mean Squared Error", é dada por:*

$$\mathbf{C} = \frac{1}{n}\sum_{i=1}^{n} (Y_i - Y_i^{previsto})^2. \tag{1.36}$$

Observação 1.2.2.4. *É muito comum encontrar versões modificadas da função de custo MSE, como por exemplo:*

$$\mathbf{C} = \frac{1}{2}\sum_{i=1}^{n} (Y_i - Y_i^{previsto})^2. \tag{1.37}$$

Isto acontece porque dependendo do problema, tal modificação facilita o cálculo do gradiente e a constante não afetará o processo de minimização.

Exemplo 1.2.2.5. *(RSS) A função de custo "Soma dos Quadrados dos Resíduos" ou simplesmente RSS, do inglês "Residual Sum of Squares", é dada por:*

$$\mathbf{C} = \sum_{i=1}^{n} (Y_i - Y_i^{previsto})^2. \tag{1.38}$$

Estas funções de custo geralmente são utilizadas em regressões e, dependendo do contexto, podem não se adequar ao método do gradiente descendente, uma vez que a MAE não é diferenciável em zero e a MSE e RSS são sensíveis a outliers, que são pontos que se

diferenciam significativamente da maioria do conjunto de dados, como descreve [2]. Isso acontece, porque grandes erros são amplificados pela potência das equações. Nestes casos, é possível encontrar [2], outras funções que possam vir a cumprir um melhor desempenho, como a RMSE, do inglês "Root Mean Squared Error", a MSLE do inglês "Mean Squared Logarithmic Error", entre outras.

Resumidamente, o gradiente descendente é um método de minimização, que usa a informação do gradiente para se mover iterativamente em direção ao ponto de mínimo de uma função. Com o objetivo de minimizar o erro de um ajuste, o método utiliza o gradiente de uma determinada função de custo, para se mover iterativamente em direção oposta.

Na próxima seção, abordaremos os detalhes importantes sobre vertentes do gradiente descendente e novos hiperparâmetros que auxiliam a controlar a convergência do método.

1.2.3 Hiperparâmetros

O método do gradiente descendente se distingue por um conjunto de hiperparâmetros que geram novos tipos de algortimos, e governam seu processo de otimização e consequentemente, convergência. Estes parâmetros, definidos externamente à estrutura do modelo, são cruciais para a eficácia do algoritmo durante o treinamento.

No método do gradiente descendente, os parâmetros são os valores que o algoritmo tenta ajustar para minimizar a função de custo. O hiperparâmetro é o "tamanho" do passo, ou seja, é o α descrito na Definição 1.34. O "tamanho" do passo é também conhecido como taxa de aprendizagem, do inglês "learning rate". O valor de α pode ser fixo ou variar a cada iteração, e nesse caso em especial, será um parâmetro. Neste trabalho, a taxa de aprendizagem será sempre um hiperparâmetro fixo, denotado por taxa de aprendizagem.

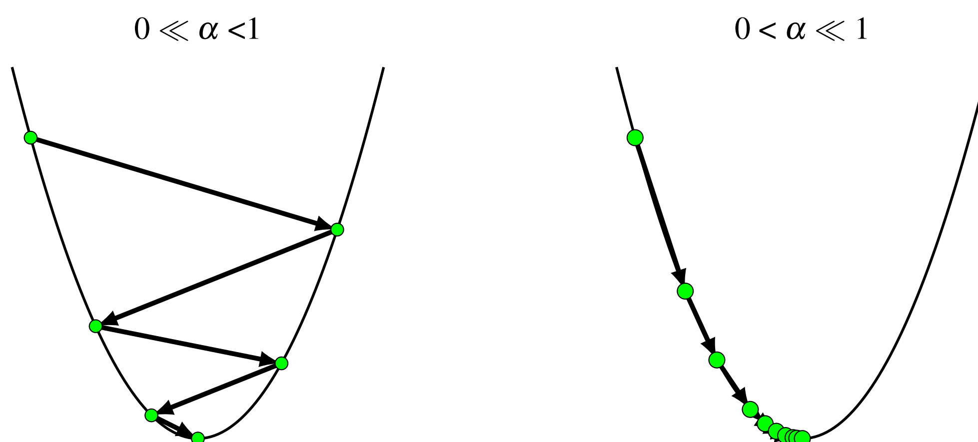


Figura 1.3 Esquema ilustrativo do método do gradiente descendente para diferentes taxas de aprendizado.

Um detalhe importante, é que taxas excessivamente altas podem induzir a oscilações ou divergência, enquanto taxas muito baixas prolongam o processo de convergência, como demonstra a Figura 1.3.

Além da taxa de aprendizado, existe também o número de iterações, ou de épocas, que é o valor que determina quantas vezes o algoritmo processará todo o conjunto de dados. Um número insuficiente de iterações pode impedir que o modelo alcance a convergência, enquanto um número excessivo pode levar a um desperdício de recursos computacionais e potencial sobreajuste, que ocorre quando o modelo se ajusta excessivamente aos dados observados, comprometendo sua capacidade de generalizar para dados não vistos.

O tamanho do lote, mini-lote ou batch size, refere-se ao número de amostras usadas para calcular cada atualização do gradiente. Este tamanho pode influenciar a estabilidade e a velocidade da convergência, onde a estabilidade pode ser definida como uma convergência que não possui muitas oscilações [2]. Lotes menores tendem a oferecer uma convergência mais rápida, mas com mais oscilações, enquanto lotes maiores proporcionam estabilidade, mas podem ser mais lentos.

Por fim, existe um hiperparâmetro pouco conhecido mas de muito interesse, que é o momento, ou momentum. Este hiperparâmetro auxilia na aceleração do gradiente descendente em direções favoráveis, ao integrar a direção dos gradientes anteriores ao cálculo atual, fazendo com que o algortimo evite mínimos locais ou gradientes próximos de zero. Este hiperparâmetro, em especial, será detalhado na seção a seguir.

Os hiperparâmetros do gradiente descendente são cruciais para a convergência eficaz do algoritmo. Eles calibram a sensibilidade e a direção do processo de otimização, equilibrando a precisão e a velocidade. Uma escolha adequada desses parâmetros pode acelerar o aprendizado e assegurar uma generalização robusta do modelo. Sem essa sintonia cuidadosa, o algoritmo pode falhar em aprender ou consumir recursos excessivos, enfatizando a importância de um ajuste fino para um treinamento eficiente e eficaz.

1.2.4 Tipos de Gradiente Descendente

Atualmente, existem múltiplas variantes do gradiente descendente, ajustadas para desafios específicos e conjuntos de dados de distintas magnitudes. Em [2], é possível encontrar algumas dessas variantes.

A primeira delas é a mais simples, chamado de gradiente descendente em lote. Esta abordagem tradicional computa o gradiente para o conjunto de treinamento integral, ou seja, utiliza todo o conjunto de treinamento sem dividi-lo em lotes menores. É fortemente adequado para conjuntos de dados de menor escala, mas menos eficiente para dados com maior volume, visto que todo o conjunto de treinamento é processado de uma vez só.

Enquanto isso, o gradiente descendente estocástico, conhecido como "SGD" do inglês "Stochastic Gradient Descent", é o método onde os parâmetros são atualizados para cada amostra de treinamento individualmente. Embora sua natureza estocástica possa introduzir flutuações no processo de otimização, como o SGD lida com um exemplo de treinamento por vez, ele requer menos memória, tornando-o adequado, escalável e eficiente para conjuntos de dados grandes que não cabem na memória de uma vez. Para um conjunto de dados de treinamento com N amostras, haverá N atualizações dos parâmetros por iteração.

Por fim, temos o gradiente descendente em mini-lote. Esta técnica intermedia o gradiente descendente em lote e o estocástico, utilizando subconjuntos do conjunto de dados para atualizações, equilibrando eficiência e estabilidade, sendo amplamente adotada na prática atual. Este método tem como base primordial a utilização do tamanho do lote, que muitas vezes é definido como *batch size*. E é a partir deste hiperparâmetro que a base se divide em amostras, de forma que se o conjunto de dados de treinamento tiver N exemplos e o tamanho do mini-lote for M, haverá $\frac{N}{M}$ atualizações dos parâmetros por iteração.

Vale ressaltar que, no contexto de otimização, um vale é uma região do espaço de parâmetros onde o erro é menor do que nas áreas vizinhas. O objetivo do método é encontrar o ponto mais baixo do vale, que é o mínimo global do erro. No entanto, isto nem sempre será possível, pois também podem existir mínimos locais, para os quais o método pode convergir. Enquanto isso, platôs são regiões do espaço de parâmetros onde a função de custo não muda muito ou muda muito lentamente, ou seja, o gradiente é próximo de zero.

A integração dos gradientes anteriores ajuda a suavizar a atualização dos parâmetros, de modo que o algoritmo pode ganhar "inércia". Isso é benéfico pois, se o algoritmo estiver em um platô, a inércia pode ajudá-lo a continuar se movendo e possivelmente sair dessa região. Por outro lado, se o algoritmo estiver em um vale, mas se aproximando de um mínimo local, a inércia pode ajudá-lo a não ficar preso e, talvez, ajude o algoritmo a passar por cima do mínimo local em direção a um mínimo global. Tendo disto isso, podemos definir formalmente o método do gradiente descendente com momento.

Definição 1.2.4.1. *(Método do Gradiente Descendente com Momento) Seja* $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ *uma função a ser minimizada. O método do gradiente descendente consiste em encontrar o ponto* $x^\star \in \mathbb{R}^n$ *tal que* $f(x^\star)$ *seja o mínimo de* f*. Essa tarefa é realizada atualizando repetidamente o ponto inicial* x *na direção oposta ao gradiente de* f *nesse ponto. A atualização dos parâmetros com momento é realizada em duas etapas:*

1. Atualização da Velocidade:

$$v_{k+1} = \beta v_k + (1-\beta)\nabla f(x_k), \tag{1.39}$$

onde:

- v_k *é a velocidade, ou momento, na iteração k.*
- β *é o hiperparâmetro do momento, que determina a fração do momento anterior a ser mantida, geralmente* $0 \leq \beta < 1$.

2. *Atualização do Ponto:*

$$x_{k+1} = x_k - \alpha v_{k+1} = x_k - \alpha[\beta v_k + (1-\beta)\nabla f(x_k)]. \tag{1.40}$$

Observação 1.2.4.2. *Geralmente, a velocidade inicial é definida como zero, para assegurar um estado inicial neutro, sem requerer conhecimento prévio sobre a distribuição dos gradientes.*

O método do gradiente descendente com momento, portanto, suaviza e acelera a trajetória de x em direção ao mínimo de f. A decisão de adotar um momento de, por exemplo, 0.9, indica que 90% do gradiente anterior contribuirá para a atualização corrente.

Na próxima seção, iremos implementar um algortimo com o método do gradiente descendente incluindo a recursão e veremos como a recursividade pode atuar em nosso favor, especialmente em casos não lineares.

1.2.5 Implementação do Gradiente Descendente

Já sabemos que o gradiente descendente emerge como uma técnica fundamental no campo da otimização matemática e aprendizado de máquina, permitindo a minimização de funções de custo de forma eficiente e eficaz. Nesta seção, dedicaremos nosso foco à implementação prática desta metodologia, abordando tanto ajuste lineares quanto não lineares, para demonstrar sua versatilidade e poder em encontrar soluções ótimas. Veremos que, no caso linear, encontraremos exatamente a mesma solução que o método dos mínimos quadrados.

Para iniciar a implementação devemos, antes de mais nada, definir muito bem um problema e a função de custo a ser minimizada. Inicialmente, vamos considerar o mesmo conjunto de dados dispersado na Figura 1.1 do ínicio deste capítulo. Presumindo que desejamos fazer um ajuste linear como na Equação 1.1, a função de custo que iremos minimizar é a função da soma quadrática dos resíduos, proposto no Exemplo 1.2.2.5.

$$\mathcal{C} = \sum_{i=1}^{n}(Y_i - Y_i^{previsto})^2 = \sum_{i=1}^{n}(Y_i - (\theta + \beta x_i))^2, \tag{1.41}$$

em que

$$custo_i = Y_i - (\theta + \beta x_i). \tag{1.42}$$

Como vamos utilizar o gradiente desta função, que claramente é diferenciável, precisamos antes de mais nada efetuar o cálculo da derivada em relação as variáveis de interesse. No nosso caso, são exatamente θ e β. Note que:

$$\frac{\partial \mathcal{C}}{\partial \theta} = \sum_{i=1}^{n} \frac{\partial \mathcal{C}}{\partial \text{custo}_i} \cdot \frac{\partial \text{custo}_i}{\partial \theta}, \tag{1.43}$$

$$\frac{\partial \mathcal{C}}{\partial \beta} = \sum_{i=1}^{n} \frac{\partial \mathcal{C}}{\partial \text{custo}_i} \cdot \frac{\partial \text{custo}_i}{\partial \beta}, \tag{1.44}$$

em que,

$$\frac{\partial \mathcal{C}}{\partial \text{custo}_i} = 2\sum_{i=1}^{n} (Y_i - Y_i^{previsto}) = 2\sum_{i=1}^{n} (Y_i - (\theta + \beta x_i)), \tag{1.45}$$

$$\frac{\partial}{\partial \theta} = -1, \tag{1.46}$$

$$\frac{\partial}{\partial \beta} = -x. \tag{1.47}$$

E portanto,

$$\frac{\partial \mathcal{C}}{\partial \theta} = 2\sum_{i=1}^{n} (Y_i - Y_i^{previsto})(-1) = 2\sum_{i=1}^{n} (Y_i^{previsto} - Y_i), \tag{1.48}$$

$$\frac{\partial \mathcal{C}}{\partial \beta} = 2\sum_{i=1}^{n} (Y_i - Y_i^{previsto})(-x_i) = 2\sum_{i=1}^{n} (Y_i^{previsto} - Y_i)x_i. \tag{1.49}$$

Agora que já temos o gradiente bem definido, é possível realizar as iterações para encontrar o resultado desejado. Para facilitar o processo, faremos a implementação desse algoritmo na linguagem de programação Python. Iniciaremos com uma escolha arbitrária de parâmetros iniciais $\theta = \beta = 1$, e uma taxa de aprendizagem inicial $\alpha = 0.0001$. Para fins de depuração, considere $a = \theta$, $b = \beta$ e taxa_de_aprendizagem = α.

Listagem 1.2.1 Parte I - Algoritmo do Gradiente Descendente

```
1 # Define os parâmetros iniciais
2 a = 1
3 b = 1
```

```
4
5 # Define a taxa de aprendizagem
6 taxa_de_aprendizagem = 0.0001
```

Resta, então, definir a quantidade de iterações e por fim, realizar o loop de treinamento.

Listagem 1.2.2 Parte II - Algoritmo do Gradiente Descendente

```
1 # Define a quantidade de iterações
2 max_interacoes = 10000
3
4 # Loop de treinamento
5 for epoca in range(max_interacoes):
6
7     # Armazena os valores anteriores de "a" e "b" para verificação de critério de parada
8     anterior_a = a
9     anterior_b = b
10
11     # Cálculo das previsões (y_pred) usando o modelo linear atual
12     y_pred = a + b * x
13
14     # Cálculo do gradiente da função de custo em relação a "a"
15     grad_a = 2 * np.sum(y_pred − y)
16
17     # Cálculo do gradiente da função de custo em relação a "b"
18     grad_b = 2 * np.sum((y_pred − y) * x)
19
20     # Atualização dos parâmetros "a" e "b" usando gradiente descendente
21     a = a − (taxa_de_aprendizagem * grad_a)
22     b = b − (taxa_de_aprendizagem * grad_b)
23
24     # Cálculo da função de custo (soma dos quadrados dos resíduos)
25     custo = np.sum((y_pred − y)**2)
26
27     # Critério de parada: se os valores de "a" e "b" se estabilizarem com precisão de três
         casas decimais, o loop éinter
28     if np.all((round(float(a), 3) == round(float(anterior_a), 3)) and
29               (round(float(b), 3) == round(float(anterior_b), 3))):
```

```
            print(f"Critério de parada atingido na iteração {epoca} com a={a} e b={b}")
            break
```

O algoritmo atingiu exatamente o mesmo resultado e mesmo custo que o método dos mínimos quadrados na Equação 1.26, com precisão de 3 casas decimais, na iteração 3002 em apenas 1 segundo e 68 milésimos, numa máquina com 16GB de RAM, processador intel *i*5 de 11ª geração.

Caso a escolha inicial dos parâmetros α e β fossem diferentes, a quantidade iterações para atingir o critério de parada seria diferente. Além disso, a escolha da taxa de aprendizagem também influencia na quantidade de iterações para atingir o critério de parada desejado. De todo modo, isso demonstra a eficiência do gradiente descendente em vários aspectos e a sua facilidade de implementação.

No entanto, para implementar o gradiente descendente com momento, bastaria definir os valores iniciais de velocidade, sua atualização e também dos pesos, além de definir a taxa de momento desejada.

Listagem 1.2.3 Parte III - Algoritmo do Gradiente Descendente com Momento

```
# Define a velocidade inicial igual a zero
v_a = 0
v_b = 0

# Define hiperparâmetro do momento
beta = 0.9

# Loop de treinamento
for epoca in range(epocas):
    ...

    # Atualização das velocidades
    v_a = beta * v_a + (1 - beta) * grad_a
    v_b = beta * v_b + (1 - beta) * grad_b

    # Atualização dos parâmetros "a" e "b" usando gradiente descendente com momento
    a = a - (learning_rate * v_a)
    b = b - (learning_rate * v_b)

    ...
```

O gradiente descendente com taxa de momento em 90%, atingiu os mesmos resultados que o gradiente descendente com 43 iterações a menos. Este valor não parece ser tão considerável, mas pensando em conjuntos de treinamento de larga escala e de complexidade superior, os resultados tendem a ser mais significativos.

A versatilidade do gradiente descendente nos permite aplicar o mesmo processo para determinar soluções em problemas onde a não linearidade é muito expressiva. Para vislumbrar tal feito, vamos considerar o conjunto de dados da Figura 1.4, que possui uma distribuição claramente não linear.

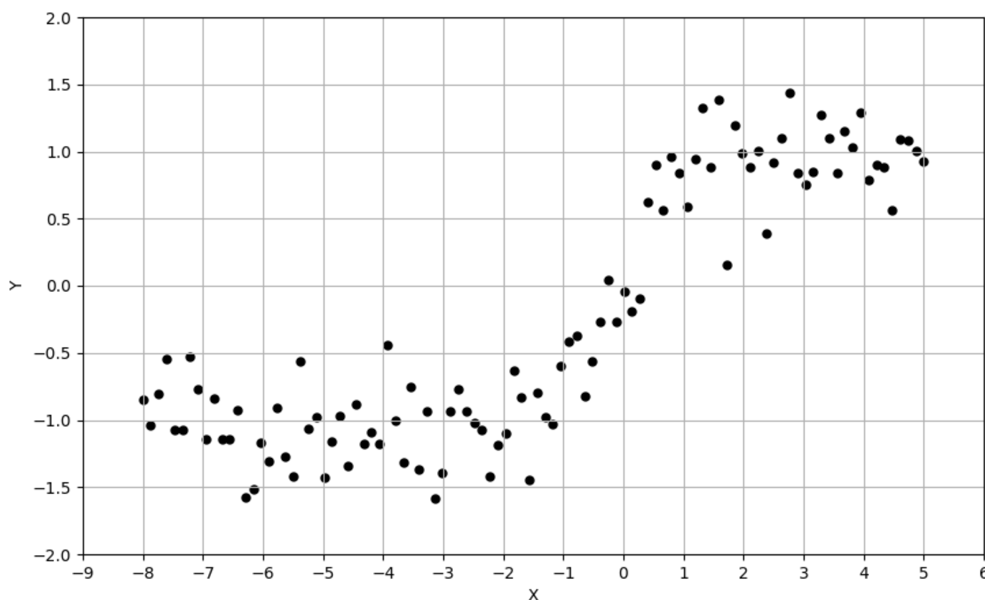


Figura 1.4 Dispersão de Dados Não Linear

Ao realizar este ajuste usando para tal uma equação linear como a Equação 1.1 e o método dos mínimos quadrados, considerando arrendondamentos com três casas decimais, obtemos:

$$\alpha = 0.063 \text{ e } \beta = 0.215, \tag{1.50}$$

e que o valor da soma dos quadrados dos resíduos foi de aproximadente 24.770. O gradiente descendente necessitou de 575 iterações para atingir os mesmos valores com uma taxa de aprendizado de 0.0001.

Agora, vamos realizar o ajuste com a função não linear tangente hiperbólica, dada por:

$$Y(x) = a \cdot tanh(bx + c). \tag{1.51}$$

Com isso, devemos determinar pelo método do gradiente descendente os parâmetros a, b e c que descrevem o conjunto de dados da melhor maneira possível, isto é, visando a minimização da função de custo. De forma análoga ao exemplo anterior, temos que determinar as derivadas parciais da função custo com relação as variáveis a, b e c, isto é:

$$\begin{aligned}
\frac{\partial \text{custo}_i}{\partial a} &= 2\sum_{i=1}^{n}(Y_i^{\text{previsto}} - Y_i) \cdot \tanh(bx_i + c), \\
\frac{\partial \text{custo}_i}{\partial b} &= 2\sum_{i=1}^{n}(Y_i^{\text{previsto}} - Y_i) \cdot a \cdot \text{sech}^2(bx_i + c) \cdot x_i, \\
\frac{\partial \text{custo}_i}{\partial c} &= 2\sum_{i=1}^{n}(Y_i^{\text{previsto}} - Y_i) \cdot a \cdot \text{sech}^2(bx_i + c).
\end{aligned} \tag{1.52}$$

Tendo os gradientes necessários, é possível fazer a implementação desse algoritmo na linguagem de programação Python, de forma análoga ao exemplo anterior. Para uma escolha de parâmetros iniciais $a = b = c = 1$, e uma taxa de aprendizagem inicial $\alpha = 0.01$, o algoritmo atingiu o critério de parada na iteração 69, com custo total de aproximademente 7.334. Os paramêtros determinados pelo método foram aproximadamente $a = 1.032$, $b = 1.038$ e $c = 0.024$.

Os traços desses ajustes podem ser visualizados na Figura 1.5.

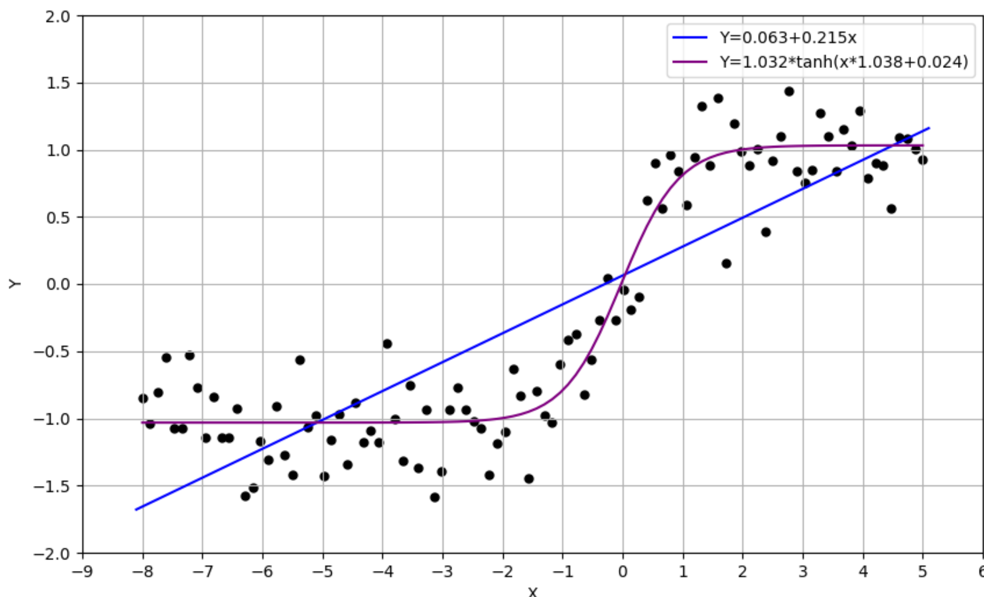


Figura 1.5 Comparação do ajuste não linear com gradiente descendente e ajuste linear com método dos mínimos quadrados.

Note que, apesar dos excelentes resultados, uma limitação do gradiente descendente é a possibilidade de convergência para mínimos locais, principalmente quando há uma quantidade grande de parâmetros a serem ajustados. Isso pode impedir que o modelo atinja o melhor desempenho global possível. Além disso, o cálculo de gradientes pode ser desafiador em algumas situações, como quando a função de custo escolhida não for diferenciável. Se o gradiente for próximo de zero, isso também pode desacelerar a convergência, pois fará com que alguns neurônios não influenciem na saída da rede. Sem contar que, além disso, existe a sensibilidade a inicialização dos parâmetros.

Em problemas reais, a não linearidade é quase instríseca e muitas vezes requer um processo de recursividade no ajuste para otimizar os resultados e fazer com que o modelo se ajuste a dados complexos. Por exemplo, aplicar a recursividade no problema pressuposto acima, nos permite definir o ajuste do seguinte modo:

$$Y(x) = a_1 \cdot tanh[b_1(a_2 \cdot tanh(b_2 x + c_2)) + c_1]. \tag{1.53}$$

Determinando os mesmos valores iniciais que o modelo anterior, isto é, $a_1 = b_1 = ... = c_2 = 1$ e taxa de aprendizagem $\alpha = 0.01$, o algoritmo atingiu o mesmo critério de parada, com precisão de 3 casas decimais na iteração 65, com custo total de 7.303. Isso nos mostra um pequeno avanço em relação ao algoritmo anterior. Veja na Figura 1.6 a comparação dos ajustes realizados até aqui para o conjunto de dados disposto na Figura 1.4.

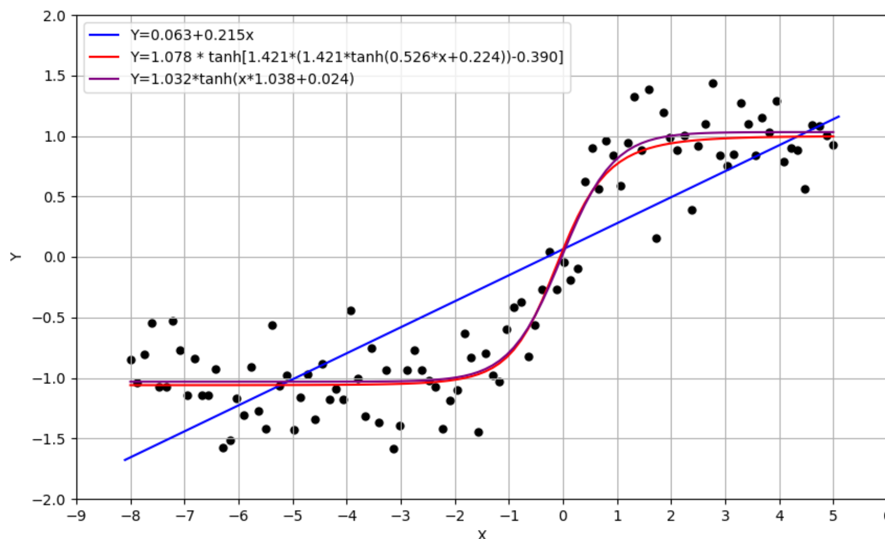


Figura 1.6 Comparação de ajustes não lineares obtidos através do gradiente descendente com e sem recursividade e o método dos mínimos quadrados.

Além disso, note que podemos avaliar também, a evolução do custo ao longo do treinamento, como mostra a Figura 1.7.

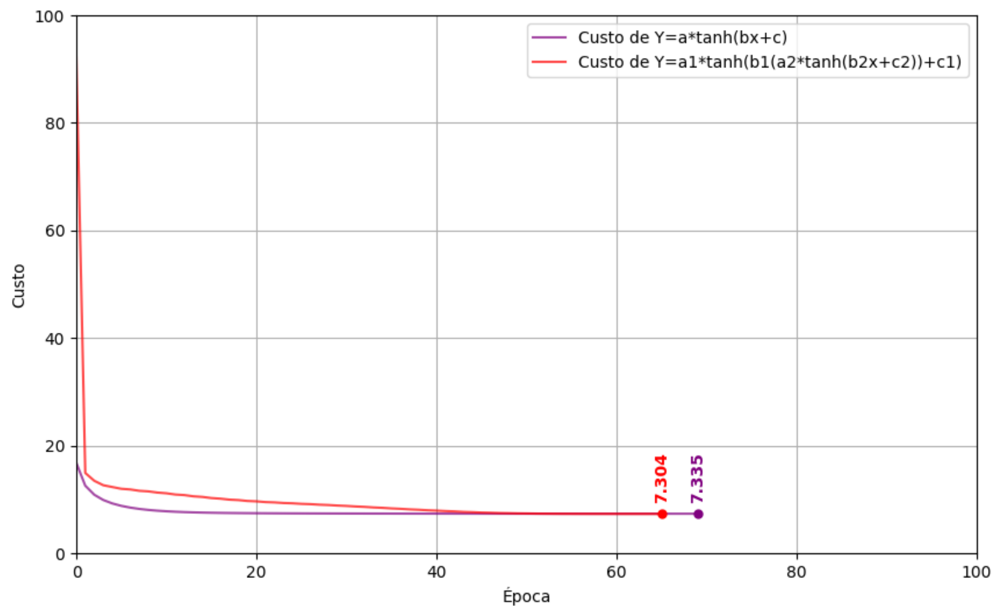


Figura 1.7 Evolução do custo nos treinamentos do gradiente descendente com e sem recursividade nos ajustes não lineares.

No próximo capítulo, entenderemos como uma máquina é capaz de aprender e prever utilizando métodos de otimização como o gradiente descendente para minimização do custo, visando encontrar respostas para um determinado problema independente da natureza dos dados.

Capítulo 2

Redes Neurais Artificiais

Com o avanço científico, métodos matemáticos emergiram fazendo surgir possibilidades jamais imaginadas na ciência da computação, e consequentemente, na atualidade. A combinação entre otimização e poder de alto processamento computacional trouxe à tona novos conceitos, como o de redes neurais artificias, que simulam o processo de aprendizagem do cérebro humano em uma máquina.

O fascínio pela complexidade do cérebro humano e sua capacidade inigualável de aprender, adaptar-se e inovar-se, têm sido fontes de inspiração para diversos estudos ao longos dos anos, se entrelaçando com todos os conceitos vistos até aqui. No centro dessa inspiração está a estrutura neural do cérebro, um emaranhado de bilhões de neurônios interconectados que trabalham em harmonia para processar informações, aprender novas habilidades e formar memórias, mesmo diante de situações adversas. Em um contexto matemático, podem ser interpretados como problemas não-lineares de alta complexidade.

Neste capítulo, vamos entender como funciona o processo de aprendizado de máquina com redes neurais artificiais, através do desenvolvimento de um modelo computacional, utilizando de forma intríseca a minimização de uma elegida função de custo.

As fundamentações teóricas descritas neste capítulo, estão baseadas nos livros [6, 20, 21].

2.1 Estrutura Elementar

Todos nós sabemos que o corpo humano é composto por milhares de células. Umas das mais interessantes é o neurônio, uma célula do sistema nervoso capaz de receber estímulos internos ou externos e, a partir deles, estabelecer conexões entre si. Esta característica peculiar é o que torna possível o processo de aprendizagem humana baseada em sua capacidade de processamento de informações.

Esse processo se inicia nos dentritos, como mostra a Figura 2.1, onde os neurônios recebem os sinais, processando-os no núcleo. Esse processamento é geralmente chamado de "soma". Caso o sinal do processamento seja forte o suficiente, um novo sinal de saída é transmitido através do axônio para outros neurônios. Essa transmissão gera impulsos nervosos por meio das junções entre a terminação de um neurônio e a membrana de outro neurônio e recebe o nome de sinapse. A partir dessas sinapses, obtemos o que é chamado de rede neural, as quais costumam ser tão complexas que fornecem ao ser humano a capacidade de memorização, associação e reconhecimento [1]. Na Figura 2.1, é possível visualizar a representação de um neurônio humano.

Figura 2.1 Imagem adaptada da ilustração de um neurônio humano. Obtida originalmente em [1].

Em [6], Haykin destaca que os neurônios são significativamente mais lentos que os portões lógicos de silício, operando em milissegundos comparados aos nanossegundos dos circuitos de silício. Contudo, o cérebro humano compensa essa lentidão com uma vasta quantidade de neurônios e conexões sinápticas. O córtex humano contém aproximadamente 10 bilhões de neurônios e 60 trilhões de sinapses, resultando em uma estrutura altamente eficiente.

Todo este processo tem como resultado a capacidade de aprendizagem e, do ponto de vista matemático, comporta-se como um entrelaçado de conceitos como o de grafos, e aplicação recursiva de funções atreladas a minimização de uma elegida função de custo. Com a junção destes e de outros conceitos matemáticos, é possível fomentar a definição de redes neurais artificiais, uma poderosa ferramenta da ciência da computação, que ganhou grande

notoriedade na área de aprendizado de máquina e consequentemente, na inteligência artificial [2].

Uma rede neural artificial nada mais é do que um modelo computacional inspirado no cérebro humano capaz de desempenhar várias funções através do processamento de dados. Esses objetos são projetados para reconhecer padrões complexos e realizar tarefas de aprendizado de máquina, como classificação e regressão, por exemplo. Para entender como este modelo é definido, é necessário entender alguns conceitos.

Definição 2.1.0.1. *Um Neurônio Artificial é uma unidade de processamento computacional que processa as informações de um conjunto* $\vartheta = \{x_1, x_2, ..., x_n\}$*, chamado de entradas, de tal forma que dada uma função* $f : \vartheta \subset \mathbb{R} \longrightarrow \mathbb{R}$ *o objeto de saída é dado da seguinte maneira*

$$\eta = f\left(\sum_{i=1}^{n} x_i\, w_i + b_i\right), \tag{2.1}$$

em que os w_i*'s são os pesos e* b_i *são chamados de viés, provendo um ajuste linear seguido da uma aplicação de uma função.*

Definição 2.1.0.2. *A função* f *presente na Definição 2.1.0.1 é denomidada Função de Ativação e geralmente é não linear. É a função aplicada à saída da combinação linear de um neurônio e é utilizada para introduzir a não linearidade dentro do modelo. Isso permite que a máquina compreenda distribuições complexas de dados.*

Exemplo 2.1.0.3. *A função de ativação ReLU, ou Unidade Linear Retificada, é dada por*

$$ReLU(x) = max(0, x). \tag{2.2}$$

Essa função de ativação é caracterizada por sua simplicidade e eficiência computacional. Um aspecto interessante é que, se um neurônio produzir zeros como saída, o gradiente também será zero durante a minimização da função de custo, resultando na ausência de atualização de pesos para esse neurônio, conhecido como "morte do neurônio". Esse fenômeno pode ser visto tanto como um desafio quanto como uma vantagem, dependendo da complexidade dos dados e do modelo em uso.

Definição 2.1.0.4. *Em uma rede neural artificial, uma coleção de Neurônios que operam em conjunto de forma paralela é chamada de camada. Existem três tipos de camadas, sendo elas a de entrada, oculta e saída. A camada de entrada é a primeira camada da rede neural artificial composta pelas variáveis independentes. A camada de saída é a última, sendo responsável por gerar as previsões do modelo e, por fim, todas as camadas intermediárias*

são chamadas de camadas ocultas. Estas, por sua vez, possuem esse nome por não ficarem explicítas durante o processo iterativo, mas são intrísecas e responsáveis pela produção do resultado.

Na Figura 2.2, é possível visualizar uma representação ilustrativa de uma rede neural artificial, onde os neurônios rosas compõem a camada de entrada com quatro neurônios, os azuis a camada de saída com três neurônios e os vermelhos duas camadas ocultas contendo cinco neurônios cada. As linhas, por sua vez, representam as conexões entre neurônios, às quais estão associadas a pesos e viés.

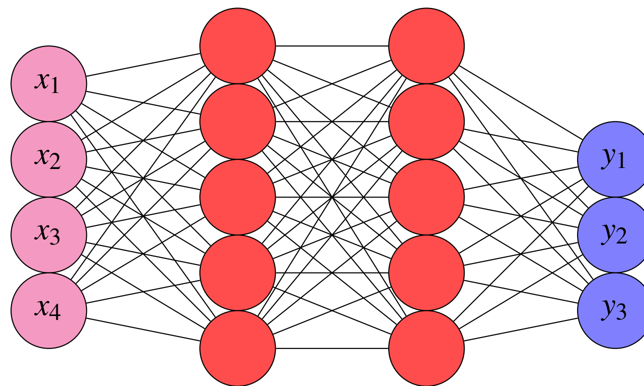


Figura 2.2 Ilustração de uma rede neural artificial com camadas densas.

Definição 2.1.0.5. *Quando uma camada é totalmente conectada a próxima, isto é, todos os neurônios de uma camada conectam-se com todos os neurônios da camada seguinte, ela é chamada de densa. Em uma camada densa, digamos l, com n neurônios, a quantidade total de parâmetros a ser atualizada nesta camada, é dada por*

$$\zeta_l = (\vartheta_n \cdot n) + n, \tag{2.3}$$

em que ϑ_n corresponde a quantidade de entradas.

Observação 2.1.0.6. *A rede apresentada na Figura 2.2 é densa.*

Agora que entendemos os principais conceitos, podemos definir formalmente o que é uma rede neural artificial, ou simplesmente RNA. Um neurônio individual j na camada l realiza um somatório ponderado das saídas dos neurônios da camada anterior com um conjunto de pesos $w_j^{(l)}$ e um viés $b_j^{(l)}$, seguido pela aplicação de uma função de ativação não-linear $f^{(l)}$, de acordo com as Equações 2.4 e 2.5:

$$\eta_j^{(l)} = w_j^{(l)}\, x + b_j^{(l)}, \tag{2.4}$$

$$a_j^{(l)} = f^{(l)}(\eta_j^{(l)}). \tag{2.5}$$

em que $x = a^{(l-1)}$ são as ativações da camada anterior e $a_j^{(l)}$ é a ativação do neurônio j na camada l.

Dito isso, fica evidente que a camada de entrada consiste nos dados de entrada $x = a^{(0)}$. As camadas ocultas são responsáveis por transformar as ativações da camada anterior $a^{(l-1)}$ em ativações intermediárias $a^{(l)}$ na camada l, usando os pesos $w^{(l)}$ e viés $b^{(l)}$. Por fim, a camada de saída é a última camada L e produz o vetor de saída final $Y = a^{(l)}$. Portanto, uma rede neural artificial é uma composição das funções de transformação de cada camada:

$$a^{(l)} = f^{(l)}(w^{(l)}a^{(l-1)} + b^{(l)}), \tag{2.6}$$

para $l = 1, \ldots, L$, com $a^{(0)} = x$ e $Y = a^{(L)}$. Note que, na Equação 2.6, a função f é aplicada componente a componente.

Generalizando, a saída da rede é dada por:

$$Y = F(x) = f^{(L)} \circ f^{(L-1)} \circ \ldots \circ f^{(1)}(x). \tag{2.7}$$

Note que se olharmos para o ajuste proposto na Equação 1.53, incluir a recursividade é o mesmo que adicionar mais uma camada a uma rede neural artificial, dado que ao fazer isso, estamos adicionando mais dados no conjunto de parâmetros do modelo. Dito isto, podemos dizer que uma rede neural aritificial é um processo recursivo de ajustes, utilizando para isso o poder alto de processamento das máquinas existentes na atualidade.

Se uma rede neural fosse composta apenas por operações lineares, não importa quantas camadas fossem empilhadas, a rede inteira ainda poderia ser simplificada para uma única operação linear. Isso significa que, sem as funções de ativação não lineares, a rede seria incapaz de modelar a complexidade encontrada na maioria dos problemas reais, que são intrinsecamente não lineares. O aprendizado está envolvido no ajuste dos pesos $w^{(l)}$ e os vieses $b^{(l)}$ com base nos dados de entrada e saída desejadas, a fim de minimizar uma elegida função de custo. Tal minimização tende a ser feita com métodos utilizando o gradiente da função e um dos métodos mais conhecidos e utilizados já foi discuto no Capítulo 1, a partir da Definição 1.2.1.1.

2.1.1 Alguns Tipos de Redes Neurais

Atualmente, existem vários tipos de redes neurais artificiais que se destacam tanto pelo tipo quanto pela arquitetura, e cada modelo é projetado para resolver um tipo específico de problema [7]. Neste trabalho, utilizaremos a palavra arquitetura de uma rede para referir-nos

à sua estrutura organizacional das camadas e a quantidade de neurônios por camada. No entanto, é comum encontrar na literatura que a arquitetura de uma rede é definida com base no conjunto de características das suas camadas, conexões, funções de ativação e outros componentes que compõem a rede. Neste trabalho, este último aspecto será definido como o tipo de rede neural.

Observe que na rede neural artificial representada na Figura 2.2, todos os neurônios de uma camada, com exceção da camada de saída, estão conectados a todos os neurônios da camada seguinte, gerando dentro da rede uma estrutura auto-associativa. Esse tipo de rede neural artificial é muito comum, sendo conhecida como "Multilayer Perceptron", ou simplesmente MLP, que nada mais é do que uma subclasse de redes neurais do tipo feedforward, ou FFNN do inglês "Feedforward Neural Networks"[7]. Numa FFNN, a informação move-se apenas numa direção, "para frente", através das camadas de entrada, passando pelas camadas ocultas e, finalmente, chegando a camada de saída. Não há ciclos ou laços na rede, o que significa que a saída de qualquer camada não afeta essa mesma camada. Na Figura 2.3, podemos visualizar isto de forma ilustrativa.

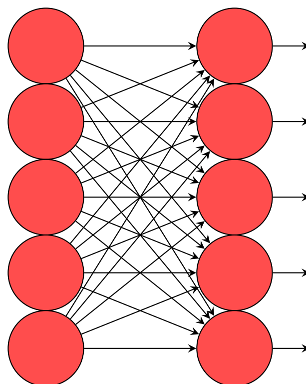


Figura 2.3 Ilustração de uma rede neural artificial feedforward.

Uma MLP por sua vez, é o nome atribuído a redes neurais artificiais onde toda camada é totalmente conectada à próxima camada. É possível encontrar em algumas definições, que em MLP's, as camadas possuem a mesma quantidade de neurônios e/ou a mesma função de ativação para todas as camadas. De todo mondo, perceba que toda MLP é uma FFNN, mas nem toda FFNN é uma MLP.

Quando a rede neural artificial não possui nenhuma camada oculta, é chamada de perceptron simples, ou simplesmente perceptron. Em [22], vemos que o perceptron foi inventado em 1958 por Frank Rosenblatt no Cornell Aeronautical Laboratory. Matematicamente, temos que, sendo f, W e b a função de ativação, pesos e viés respectivamente, a camada de saída de

um perceptron é diretamente dada por:

$$Y = f(Wx + b). \tag{2.8}$$

No entanto, um perceptron possui uma limitação importante. O modelo só é capaz de resolver problemas lineares, ou ao menos linearmente separáveis, isto porque este tipo de rede reduz a entrada a um valor proveniente de um ajuste da forma $Y = Wx + b$, que é aplicado a uma função de ativação. Essa redução a um único valor, faz com que o modelo perca a capacidade de aprender relações não lineares, visto que não possui camadas ocultas [2]. E é por isto que um perceptron não consegue resolver problemas mais complexos, como a maioria dos problemas reais que não são linearmente separáveis. Por esta razão, existem as MLP's, que são capazes de lidar com problemas não lineares de alta complexidade, desde que os dados sejam bem tratados.

Além do perceptron, das MLP's e das FFNN's, existem outros tipos de redes neurais artificiais, que podem ser encontrados em [2, 6, 7]. Dentre as mais comuns, temos as Redes Neurais Convolucionais, ou CNN do inglês "Convolutional Neural Networks", que são excelentes para processar dados em imagens, vídeos e até dados de séries temporais. Em [23], vemos que a característica principal em uma CNN é a convolução em algumas camadas, que pode ser expressada por:

$$(k * g)(i, j) = \sum_m \sum_n k(m,n)g(i-m, j-n), \tag{2.9}$$

em que k é a entrada da rede neural artificial e g é o filtro de convolução aplicado nas coordenadas (i, j) da entrada. O filtro de convolução é responsável por detectar padrões específicos, como bordas ou texturas, nas diferentes regiões da imagem. Essa operação gera o que chamamos de mapa de características, que é uma representação das características extraídas do dado de entrada após a aplicação do filtro. O mapa de características contém informações relevantes para a identificação de padrões, como formas e estruturas, que serão usados para a classificação ou detecção de objetos.

Após a convolução, ainda em [23], é possível observar que uma operação de *pooling* é frequentemente aplicada ao mapa de características. O *pooling* tem o objetivo de reduzir a dimensionalidade do mapa, simplificando a quantidade de dados que a rede processa, mas sem perder as informações mais relevantes. Isso é feito por meio de operações que agregam valores de pequenas áreas do mapa de características, chamadas de regiões. Um exemplo comum de operações de *pooling* é o "Max pooling" , que se seleciona o maior valor dentro de cada região, preservando os elementos mais proeminentes da imagem. Essas operações

são importantes para aumentar a eficiência computacional e a robustez da rede, ao mesmo tempo que preservam as informações essenciais do mapa de características.

Além dos tipos de redes citados até aqui, outro exemplo são as Redes Neurais Recorrentes, ou RNNs, do inglês "Recurrent Neural Networks". Em [24], Hochreiter e Schmidhuber explicam que esse tipo de rede recebe esse nome por possuir conexões cíclicas, tornando-as adequadas para tratar sequências de dados, como no processamento de linguagem natural. De modo geral, as redes neurais artificiais podem ser de vários tipos, cada uma projetada para tarefas específicas.

Neste trabalho, iremos utilizar a MLP como principal tipo de rede para os problemas definidos daqui em diante. Na próxima seção, detalharemos como funciona o processo de treinamento de um modelo, introduzindo novos conceitos como o de propagação e retropropagação.

2.2 Propagação e Retropropagação

Diante da discussão feita até aqui, fica evidente como a rede neural artificial é não só um modelo computacional eficaz, como também um objeto matemático. Porém, não podemos deixar de levar em consideração que este objeto é sensível a parâmetros, hiperparâmetros e iteratividade. Quanto mais o processo se repete, interligado à minimização de uma elegida função de custo para ajustar os pesos, melhor tende a ser o resultado se a rede neural artificial for equiparada com os hiperparâmetros adequados. Isso faz com que uma rede neural artificial possua dois tipos de fase, sendo elas a fase de propagação e a fase de retropropagação.

A propagação nada mais é do que o processo descrito neste trabalho até aqui, onde os dados de entrada passam pelas devidas transformações na rede neural artificial e geram uma saída. Para uma rede com L camadas, vimos que a operação em cada camada l é definida como:

$$\begin{aligned} \eta^{(l)} &= w^{(l)}a^{(l-1)} + b^{(l)}, \\ a^{(l)} &= f^{(l)}(\eta^{(l)}), \end{aligned} \tag{2.10}$$

em que $w^{(l)}$ são os pesos da camada l, $b^{(l)}$ é o viés, $a^{(l-1)}$ são as ativações da camada anterior, e $f^{(l)}$ a função de ativação. Obtemos no fim da propagação que:

$$Y = a^{(L)}. \tag{2.11}$$

Enquanto isso, na retropropagação, é usado o gradiente da escolhida função de custo para corrigir os parâmetros da rede, isto é, os pesos e vieses. Obviamente, o gradiente é para aplicação do método do gradiente descendente, descrito no Capítulo 1. Primeiramente,

vamos entender como funciona a retropropagação na camada de saída e para isso, considere $\delta_j^{(L)}$ como sendo a derivada do custo de um neurônio j, na camada de saída L. Então:

$$\begin{aligned}\delta_j^{(L)} &= \frac{\partial \mathcal{C}}{\partial \eta_j^{(L)}} \\ &= \frac{\partial \mathcal{C}}{\partial a^L} \cdot \frac{\partial a^L}{\partial \eta_j^{(L)}}.\end{aligned} \tag{2.12}$$

Tendo dito isso, pela Equação 2.10 sabemos que a a derivada da função de custo em relação aos pesos é dada por:

$$\frac{\partial \mathcal{C}}{\partial w_{ij}^{(L)}} = \underbrace{\frac{\partial \mathcal{C}}{\partial \eta_j^{(L)}}}_{\delta_j^{(L)}} \cdot \frac{\partial \eta_j^{(L)}}{\partial w_{ij}^{(L)}} = \delta_j^{(L)} \cdot a^{(L-1)}. \tag{2.13}$$

E portanto, pelo método do gradiente descendente, a atualização dos pesos na camada de saída L, será expressada por:

$$w_{ij}^{(L)} = w_{ij}^{(L)} - \alpha \cdot \delta_j^{(L)} \cdot a^{(L-1)}. \tag{2.14}$$

Analogamente, a atualização de viéses na camada de saída é dada por:

$$b_{ij}^{(L)} = b_{ij}^{(L)} - \alpha \cdot \delta_j^{(L)}. \tag{2.15}$$

De forma similar, podemos retropropagar o erro para as camadas ocultas, dando a devida atenção à regra da cadeia no momento de calcular os gradientes, uma vez que as ativações de camada l agora dependem da camada $l+1$. Com isso, a derivada do custo de um neurônio j, em uma camada oculta l, é:

$$\delta_j^{(l)} = \sum_k \frac{\partial C}{\partial \eta_k^{(l+1)}} \cdot \frac{\partial \eta_k^{(l+1)}}{\partial a_j^{(l)}} \cdot \frac{\partial a_j^{(l)}}{\partial \eta_j^{(l)}}, \tag{2.16}$$

em que k corresponde a quantidade de neurônios da camada $l+1$.

Sabendo que $\delta_k^{(l+1)} = \dfrac{\partial C}{\partial \eta_k^{(l+1)}}$ e $\dfrac{\partial \eta_k^{(l+1)}}{\partial a_j^{(l)}} = w_{jk}^{(l+1)}$, podemos reescrever a Equação 2.16 da seguinte maneira:

$$\delta_j^{(l)} = \left(\sum_k w_{jk}^{(l+1)} \delta_k^{(l+1)} \right) \cdot a^{(l)\prime}(\eta_j^{(l)}). \tag{2.17}$$

Isso nos mostra que uma vez que a rede neural faz uma previsão, para atualizar os pesos e viéses na fase de retropopagação, devemos calcular o erro na camada de saída, retropropagar o erro através da camadas ocultas da rede e por fim, atualizar os pesos e vieses usando o método do gradiente descendente.

A partir disso, podemos destacar um fator importante que é a perda, uma medida que quantifica o erro entre as previsões da rede e os valores observados ponto a ponto e é usada para ajustar os pesos da rede durante o treinamento na fase de retropropagação. Já a acurácia é uma métrica que indica a proporção de previsões corretas feitas pelo modelo em relação ao total de previsões. Por fim, o custo como dito anteriormente, refere-se ao valor agregado da função de perda para todo o conjunto de dados, muitas vezes usado como um termo intercambiável com perda, mas aplicado em um contexto mais global ou acumulado.

Ainda na retropropagação, podemos fazer uso do momento, que é o parâmetro que ajuda a acelerar a convergência e reduzir oscilações durante o treinamento. Para entender como o método do gradiente descendente com momento pode ser aplicado na fase de retropropagação, considere $v_w^{(l)}$ e $v_b^{(l)}$ sendo os vetores de momento para os pesos e vieses da camada l, e $0 < \beta < 1$ o momento. Daí, sendo t equivalente a iteração atual, temos que:

$$v_{W_{ij}}^{(l)}(t) = \beta v_{W_{ij}}^{(l)}(t-1) + (1-\beta)\nabla_{W_{ij}^{(l)}} \qquad \Longrightarrow \qquad W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha v_{W_{ij}}^{(l)}, \tag{2.18}$$

$$v_{b_{ij}}^{(l)}(t) = \beta v_{b_{ij}}^{(l)}(t-1) + (1-\beta)\nabla_{b_{ij}^{(l)}} \qquad \Longrightarrow \qquad b_{ij}^{(l)} = b_{ij}^{(l)} - \alpha v_{b_{ij}}^{(l)}. \tag{2.19}$$

Essa técnica de momento ajuda a suavizar as atualizações dos parâmetros ao longo do treino, o que pode melhorar a convergência e reduzir oscilações durante o treinamento da rede neural. Em termos mais específicos, se $\beta = 0.9$, então teremos 90% de contribuição do vetor de momento, enquanto o gradiente instantâneo da função de custo contribuirá com os 10% restantes.

Um exemplo interessante, mas que exige bastante cuidado na fase de retropropagação, são os problemas classificatórios multiclasse. Para esses casos, onde a previsão trata-se de variedade de classes, efetuar um treinamento satisfatório pode ser custoso se os hiperparâmetros da rede e a estrutura do conjunto de dados não estiver bem definida. Isto porque é necessário haver uma boa representatividade das classes existentes, para não causar desequilíbrio no modelo fazendo com que ele preveja algumas das classes majoritariamente [2].

Para os problemas de classificação, temos a função de custo categóriga de entropia cruzada, também conhecida como "Categorical Cross-Entropy Loss", CE ou "Softmax Loss".

O seu objetivo é minimizar a discrepância entre as distribuições das probabilidades de classes previstas e as reais, como é descrito com detalhes em [2]. Para entender melhor o seu conceito, antes precisamos definir uma função chamada Softmax, que é a função de ativação geralmente aplicada na camada de saída para fins de associação à entropia cruzada.

Definição 2.2.0.1. *(Softmax) Dado um vetor $z = (z_1, ..., z_n) \in \mathbb{R}^n$, a função softmax é dada por*

$$p(z_i) = softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}. \tag{2.20}$$

Ela comprime as entradas do vetor z no intervalo $(0,1)$ e todos os valores resultantes da aplicação em z_i somam 1.

Observação 2.2.0.2. *A função $softmax$ comprime um vetor de valores em um vetor de probabilidades e facilita a computação do gradiente por ser composição de funções infinitamente diferenciáveis.*

Agora, com base na Definição 2.2.0.1 podemos entender o que é a entropia cruzada.

Definição 2.2.0.3. *(Entropia Cruzada) Para um conjunto de dados com m classes, o custo categórica de entropia cruzada de uma única observação é*

$$CE_{zi} = -\sum_{j=1}^{m} y_{ij} \log(p(z_{ij})), \tag{2.21}$$

em que,

$$y_{ij} = \begin{cases} 1, & y_{ij} \in \text{à classe } m, \\ 0, & y_{ij} \notin \text{à classe } m, \end{cases} \tag{2.22}$$

e $p(z_{ij})$ é a probabilidade predita para a classe j na i-ésima observação. O custo total é definido como sendo a média dessas somas. Ou seja,

$$C = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{m} -y_{ij} \log(p(z_{ij})). \tag{2.23}$$

Perceba que se a probabilidade prevista de uma classe verdadeira estiver muito baixa, o logaritmo dessa probabilidade será um número negativo grande, resultando em uma grande penalidade. Além disso, somente o logaritmo da probabilidade da classe verdadeira contribui significativamente para o custo, incentivando o algoritmo a maximizar essa probabilidade. Veja também, que a entropia cruzada transforma a multiplicação de pequenas probabilidades em uma soma de logaritmos, tornando os cálculos numericamente mais estáveis, pois a soma

de logaritmos evita a subestimação de valores, que ocorre quando os números muito pequenos são arredondados para zero após a multiplicação. Associando estes fatos à Observação 2.2.0.2, a sinergia entre a entropia cruzada como função de custo e a softmax como função de ativação se torna evidente e eficaz em problemas de classificação porque enfatiza previsões erradas e confiantes.

Dentro da estatística temos o método da máxima verossimilhança que é uma técnica usada para estimar os parâmetros de um modelo. A ideia central desse método é encontrar os valores dos parâmetros que tornam os dados observados mais prováveis, ou seja, que maximizam a verossimilhança dos dados dados o modelo, mais detalhes podem ser vistos em [25]. Quando minimizamos o custo da entropia cruzada, estamos, na verdade, ajustando os parâmetros da rede para que a distribuição predita seja o mais próxima possível da distribuição verdadeira. Em termos estatísticos, isso corresponde a maximizar a verossimilhança dos dados observados, já que a entropia cruzada pode ser vista como uma medida de verossimilhança negativa. Ou seja, quanto menor o custo da entropia cruzada, maior é a verossimilhança, pois o modelo está sendo ajustado para atribuir maior probabilidade às classes corretas.

Ademais, é necessário estruturar os dados e a arquitetura da rede para que o resultado seja processado de forma correta. Para problemas de classificação multiclasse, devemos escolher de maneira inteligente uma função que converta as saídas em um vetor de probabilidade. Um problema que emerge a partir disto é que podem surgir gradientes menos eficazes para o ajuste do modelo, principalmente se a função custo for definida como na Equação 1.41. Isso ocorre porque, neste caso, a natureza probabilística da saída não é levada em consideração, fazendo com que haja a possibilidade de haver atualizações de peso menos direcionadas e levando talvez a oscilação ou estagnação. Sendo assim, para uma eficiente aplicação entropia cruzada associada a softmax, é necessário ajustar o conjunto de dados para que a interpretação dos resultados ocorra da maneira devida.

Uma das formas de realizar a interpretabilidade correta é utilizando a encodificação one-hot, descrita em [2], onde cada classe é representada por um vetor binário onde apenas a posição correspondente à classe é 1 e todas as outras são 0. Por exemplo, para três classes A, B e C, temos:

$$\begin{aligned} \mathrm{A} &= [1,0,0], \\ \mathrm{B} &= [0,1,0], \\ \mathrm{C} &= [0,0,1]. \end{aligned} \tag{2.24}$$

Dessa forma, o conjunto de dados será mapeado com uma transformação que garante a interpretabilidade correta dos dados.

Agora que temos a estrutura elementar de uma rede neural artificial bem definida, na próxima seção veremos a implementação na prática, associada a uma implementação computacional atrelada a métodos de validação.

2.3 Implementação em Python de Rede Neural Feedforward

Nesta seção, iremos entender os detalhes para implementar uma rede neural artificial feedforward com foco na resolução de um problema de classificação multiclasse, através de um exemplo prático. Para isto, vamos utilizar um conjunto artificial criado pelo python com uso do módulo "make_classification" da biblioteca "Scikitlearn", encontrada em [26].

Essa biblioteca oferece uma ampla gama de algoritmos para tarefas de classificação, regressão, clustering e redução de dimensionalidade, além de ferramentas para pré-processamento de conjunto de dados e avaliação de modelos. Foi criada em 2007 como um projeto "Google Summer of Code" e aprimorada posteriormente por outros pesquisadores, que tornaram o projeto público em primeiro de fevereiro de 2010 [26].

Considere X como sendo o conjunto formado pelas características, ou atributos, e y como sendo o conjunto das classes. Na Listagem 2.3.1 é possível visualizar o algoritmo para definir este conjunto em python.

Listagem 2.3.1 Rede Neural Feedforward Sem Camada Oculta - Gerando conjunto de dados

```
1 # Importando biblioteca
2 from sklearn.datasets import make_classification
3
4 # Gerando conjunto de dados
5 X, y = make_classification(n_samples=1000, n_features=3, n_informative=3,
      n_redundant=0, n_classes=3, n_clusters_per_class=2)
```

A função *make_classification* gera conjuntos de dados aleatórios para problemas de classificação. É altamente configurável, permitindo que você controle aspectos como o número de amostras, características, classes, clusters, e a quantidade de ruído nos dados, sendo um excelente auxílio para testar e validar algoritmos.

A Listagem 2.3.1 gera uma base de dados com 1000 amostras, definido pelo parâmetro *n_samples*, onde cada amostra assume uma de três características, definida pelo parâmetro *n_features*. A base possui três classes, a qual são definidas pelo parâmetro *n_classes*, o que implica em um problema de classificação multiclasse. O parâmetro *n_informative* nos assegura que todas as três características são consideradas informativas para a tarefa de classificação. Enquanto isso, o parâmetro *n_redundant* definido como zero certifica que

não há redundância nos dados, ou seja, cada característica contribui de forma única para a distinção entre as classes.

Cada classe possui dois clusters, como indica o parâmetro *n_clusters_per_class*. Tal escolha foi feita para deixar o conjunto de dados mais próximo dos problemas reais, que geralmente não possuem um único agrupamento por classe. Na figura 2.4 é possível verificar uma visualização tridimensional do conjunto gerado.

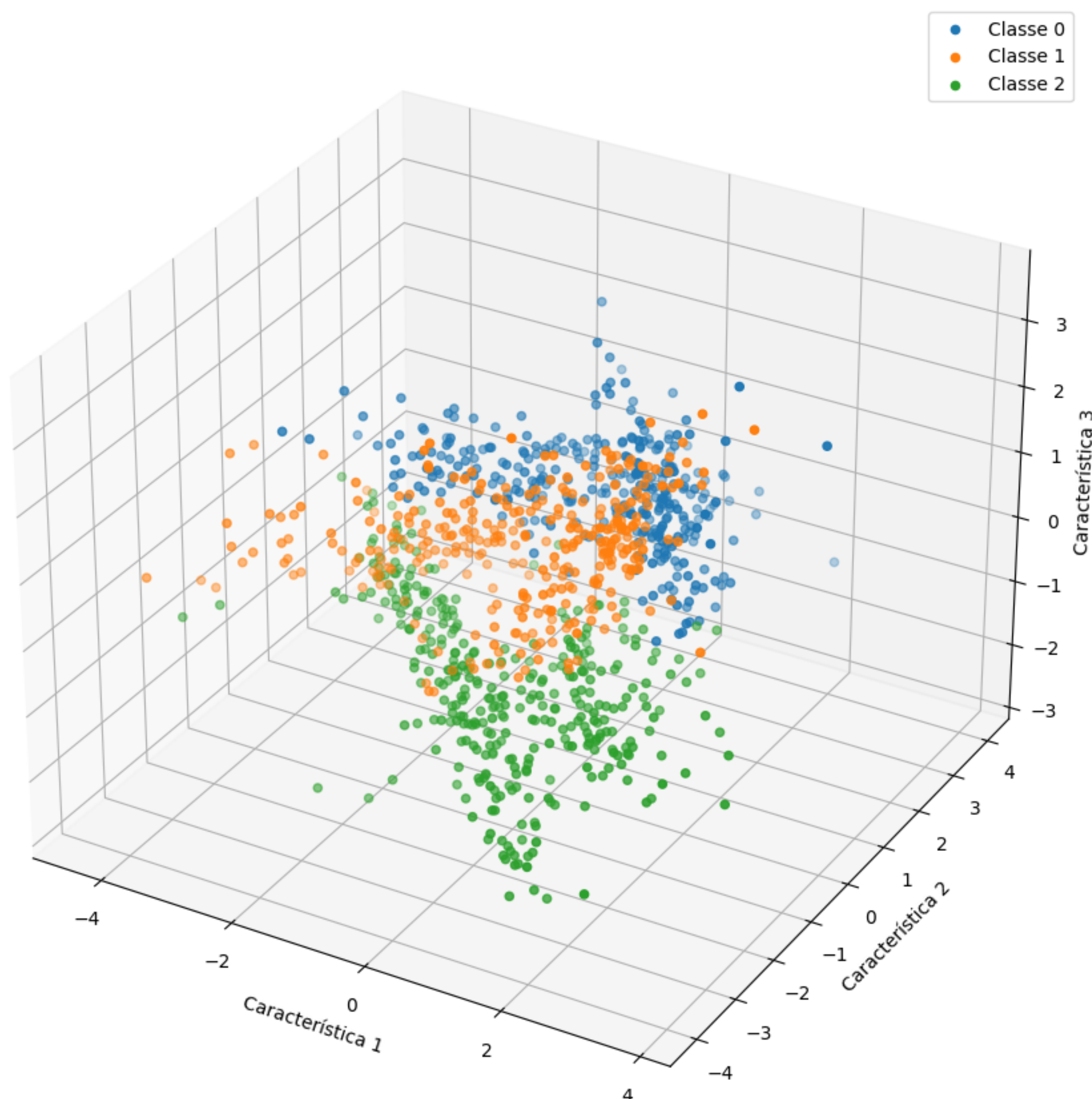


Figura 2.4 Conjunto de dados gerados pela Listagem 2.3.1.

Dada a quantidade de características, devemos ter três neurônios na camada de entrada correspondendo a essas características e um neurônio na camada de saída que irá determinar a previsão da classe. Além disso, faremos a escolha de uma camada oculta contendo quatro

neurônios. Uma representação dessa rede pode ser visualizada na Figura 2.5. Note que, para cada neurônio η_i, é aplicado o ajuste linear com peso e viés atrelado ao neurônio, seguido da função de ativação, como mostra a Figura 2.6.

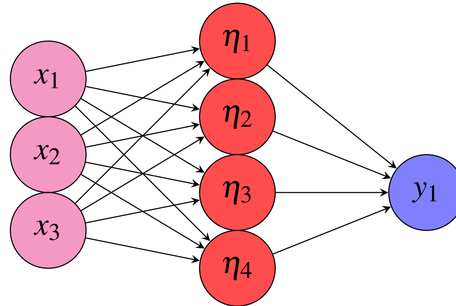


Figura 2.5 Ilustração da rede neural artificial implementada neste trabalho com uma camada oculta contendo quatro neurônios.

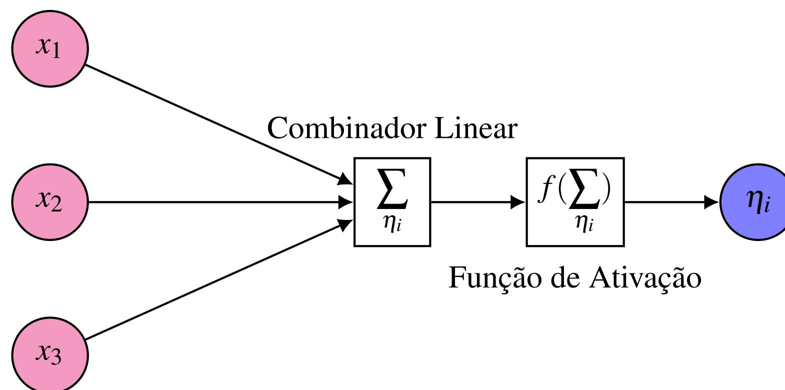


Figura 2.6 Ilustração da aplicação da função de ativação em uma rede neural artificial.

De todo modo, como estamos tratando de um problema de classificação multiclasse, devemos escolher a função de custo e função de ativação, além da arquitetura básica de camadas e neurônios. Como descrito na Seção 2.2, não é muito indicado utilizar a Equação 1.41 como função de custo. Sendo assim, utilizaremos a entropia cruzada associada ao emprego da $softmax$, descrita na Definição 2.2.0.1, na camada de saída.

Devemos lembrar que a entropia cruzada requer que as classes de saída do modelo sejam representadas como uma distribuição de probabilidade. Ao representar cada classe como um vetor binário, facilitamos a aplicação direta da entropia cruzada, permitindo uma comparação efetiva entre a distribuição de probabilidade prevista e a real. Vimos na Seção 2.2, que para um conjunto de classes $C = \{1, 2, ..., n-1, n\}$, esse conjunto encodificado em one-hot seria $C' = \{(1,0,0,...,0,0),(0,1,0,...,0,0),...,(0,0,0,...,1,0),(0,0,0,...,0,1)\}$, que nada mais é do que a base canônica do espaço n-dimensional.

Com isso, nossa rede representada na Figura 2.5 sofre uma transformação, passando a ter três neurônios na camada de saída, assim como está representado na Figura 2.7.

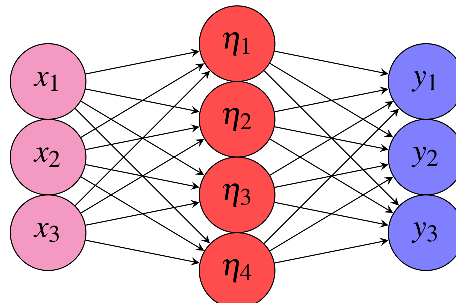


Figura 2.7 Ilustração de uma rede neural artificial com uma camada oculta contendo quatro neurônios com camada de saída encodificada.

A sinergia entre a entropia cruzada (CE) e a $softmax$ como função de ativação, fica bem elucidada no cálculo do gradiente. Considere $p_i = p(z_i) = softmax(z_i)$, e note que:

$$\begin{aligned} \frac{\partial \text{CE}}{\partial z_i} &= \frac{\partial \text{CE}}{\partial p_i} \cdot \frac{\partial p_i}{\partial z_i} \\ &\overbrace{=}^{*} -\frac{y_i}{p_i} \cdot p_i(1 - p_i) \\ &= p_i - y_i, \end{aligned} \tag{2.25}$$

em que o passo $*$ é válido porque y_i é 0 ou 1, dependendo se a classe correspondente é a classe verdadeira ou não. Assim, em resumo, a simplicidade do gradiente da função de custo de entropia cruzada quando usada com a função de ativação $softmax$, deriva da interação especial entre essas duas funções. Quando combinadas com uma codificação adequada, as características matemáticas de ambas se complementam de maneira que o gradiente da função de custo em relação à entrada da $softmax$ se simplifica, reduzindo o gradiente a diferença entre a previsão do modelo e o valor real, o que ajuda bastante computacionalmente falando.

Iniciaremos com a implementação de um perceptron para resolver esse o problema de classificação com os dados dispostos na Figura 2.4. Seguiremos definindo o conjunto de treino e teste e definindo as funções necessárias para a fase de treinamento.

Listagem 2.3.2 Rede Neural Fedforward Sem Camada Oculta - Definindo Funções e Conjunto de Treino e Teste

```
1 # Importando biblioteca
2 import numpy as np
3 from sklearn.model_selection import train_test_split
```

```
4
5 # Para ter reprodutibilidade
6 np.random.seed(0)
7
8 # Função para encodificar as classes em one-hot
9 def one_hot_encode(y, n_classes):
10     return np.eye(n_classes)[y]
11
12 # Encodificando o conjunto de classes
13 y_one_hot = one_hot_encode(y, n_classes=3)
14
15 # Define divisão dos dados em conjuntos de treino e teste
16 X_treino, X_teste, y_treino, y_teste = train_test_split(X, y_one_hot, test_size=0.2,
       random_state=0)
17
18 # Define função de ativação softmax
19 def softmax(z):
20     exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))
21     return exp_z / np.sum(exp_z, axis=1, keepdims=True)
22
23 # Define função de propagacao
24 def propagacao(X, W, b):
25     z = np.dot(X, W) + b
26     return softmax(z)
27
28 # Define função de custo
29 def entropia_cruzada(y, y_previsto):
30     return -np.mean(np.sum(y * np.log(y_previsto), axis=1))
31
32 # Define a função que calcula a acurácia
33 def calcular_acuracia(y_real, y_previsto):
34     classes_previstas = np.argmax(y_previsto, axis=1)
35     classes_reais = np.argmax(y_real, axis=1)
36     acuracia = np.mean(classes_previstas == classes_reais)
37     return acuracia
```

Resta então, definir o loop de treinamento e inicializar os pesos.

Listagem 2.3.3 Rede Neural Feedforward Sem Camada Oculta - Algoritmo de Treinamento

```
# Define função que faz o looping de treinamento
def treinamento(X, y, W, b, taxa_de_aprendizagem, epocas):
    acuracias = []
    perdas = []
    for epoca in range(epocas):
        # Executa a propagação e obtém as ativações
        y_previsto = propagacao(X,W,b)

        # Calcula o custo de entropia cruzada
        custo = entropia_cruzada(y, y_previsto)

        # Armazena o custo atual
        perdas.append((epoca, custo))

        # Inicia a fase de retropropagação com cálculo do gradiente da custo em relação
            as saídas da rede
        gradiente_perda = y_previsto - y

        # Calcula o gradiente em relação aos pesos e vieses da camada de saída
        grad_W = np.dot(X.T, gradiente_perda)
        grad_b = np.sum(gradiente_perda, axis=0)

        # Atualiza os pesos e vieses
        W -= taxa_de_aprendizagem * grad_W
        b -= taxa_de_aprendizagem * grad_b

        # Calcula a acurácia
        acuracia = calcular_acuracia(y_teste, propagacao(X_teste, W, b))

        # Armazena a acurácia atual
        acuracias.append((epoca, acuracia))

        # Define critério de parada com base no custo
        if round(custo, 1) <= 0.1:
            print(f'Critério de parada atingido na época {epoca}')
```

```
35                break
36
37        return W, b, acuracias, perdas
38
39  # Inicialização de pesos e bias
40  W = np.random.randn(3, 3) * 0.01
41  b = np.zeros(3)
```

Os pesos, W, foram inicializados de maneira a definir a quantidade de camadas e de neurônios por camada. Neste exemplo, temos uma matriz $3x3$ com valores aleatórios seguindo uma distribuição normal de média 0 e variância 1. Estes valores por sua vez, são multiplicados por 0.01, inicializando os pesos com valores pequenos. Já os vieses, são inicializados com zeros. Com esse contexto, o algoritmo está configurado para uma rede neural com três neurônios na camada de entrada, que correspondem às características, e três neurônios na camada de saída, representando as classes encodificadas.

Uma vez que o algoritmo está definido, ao realizar o treinamento com uma taxa de aprendizagem igual a 0.0001 e com 500 iterações, o modelo obteve 85% de acurácia no conjunto de teste e posteriormente 87% para todo o conjunto.

Para transformar esse perceptron em uma MLP semelhante a Figura 2.7, com uma camada oculta contendo quatro neurônios, basta ajustar a função de propagação e de treinamento, assim como a inicialização dos pesos. Isto pode ser visto na Listagem 2.3.4 a seguir, que utiliza a *ReLU*, descrita em 2.1.0.3, como função de ativação da camada oculta.

Listagem 2.3.4 Rede Neural Feedforward Com Camada Oculta - Algoritmo de Treinamento

```
1   # Define função que faz o looping de treinamento
2   # Define a função de ativação ReLU para a camada oculta
3   def ReLU(z):
4       return np.maximum(0, z)
5
6   # Define função de propagação considerando a camada oculta
7   def propagacao(X, W1, b1, W2, b2):
8       z1 = np.dot(X, W1) + b1
9       a1 = ReLU(z1)
10      z2 = np.dot(a1, W2) + b2
11      return softmax(z2)
12
13  # Define função que faz o looping de treinamento
```

```
14 def treinamento(X, y, W1, b1, W2, b2, taxa_de_aprendizagem, epocas):
15     acuracias = []
16     perdas = []
17     for epoca in range(epocas):
18         # Executa a propagação direta e obtém as ativações
19         y_previsto = propagacao(X,W,b)
20 
21         # Calcula o custo de entropia cruzada
22         custo = entropia_cruzada(y, y_previsto)
23 
24         # Armazena o custo atual
25         perdas.append((epoca, custo))
26 
27         # Inicia a fase de retropropagação com cálculo do gradiente da custo em relação
28 as saídas da rede
29         gradiente_perda = y_previsto − y
30 
31         # Calcula o gradiente em relação aos pesos e vieses da camada de saída
32         grad_loss_ReLU = np.dot(grad_loss_logits, W2.T)
33         grad_loss_ReLU[z1 <= 0] = 0
34         grad_W1 = np.dot(X.T, grad_loss_ReLU)
35         grad_b1 = np.sum(grad_loss_ReLU, axis=0)
36         grad_W2 = np.dot(a1.T, grad_loss_logits)
37         grad_b2 = np.sum(grad_loss_logits, axis=0)
38 
39         # Atualiza os pesos e vieses
40         W2 −= learning_rate * grad_W2
41         b2 −= learning_rate * grad_b2
42         W1 −= learning_rate * grad_W1
43         b1 −= learning_rate * grad_b1
44 
45         # Calcula a acurácia
46         acuracia = calcular_acuracia(y_teste, propagacao(X_teste, W1, b1, W2, b2))
47 
48         # Armazena a acurácia atual
49         acuracias.append((epoca, acuracia))
```

```
50
51         # Define critério de parada com base na custo
52         if round(custo, 1) <= 0.1:
53             print(f'Critério de parada atingido na época {epoca}')
54             break
55     return W1, b1, W2, b2, acuracias, perdas
56
57 # Inicialização de pesos e bias
58 W1 = np.random.randn(3, 4)
59 b1 = np.zeros(4)
60 W2 = np.random.randn(4, 3)
61 b2 = np.zeros(3)
```

Tendo a mesma taxa de aprendizagem e quantidade de épocas que o modelo anterior, este modelo com camada oculta obteve 88% de acurácia no conjunto de teste e 90% para o conjunto todo. Na Figura 2.8, é possível visualizar a evolução do custo ao longo das épocas para ambos os treinamentos.

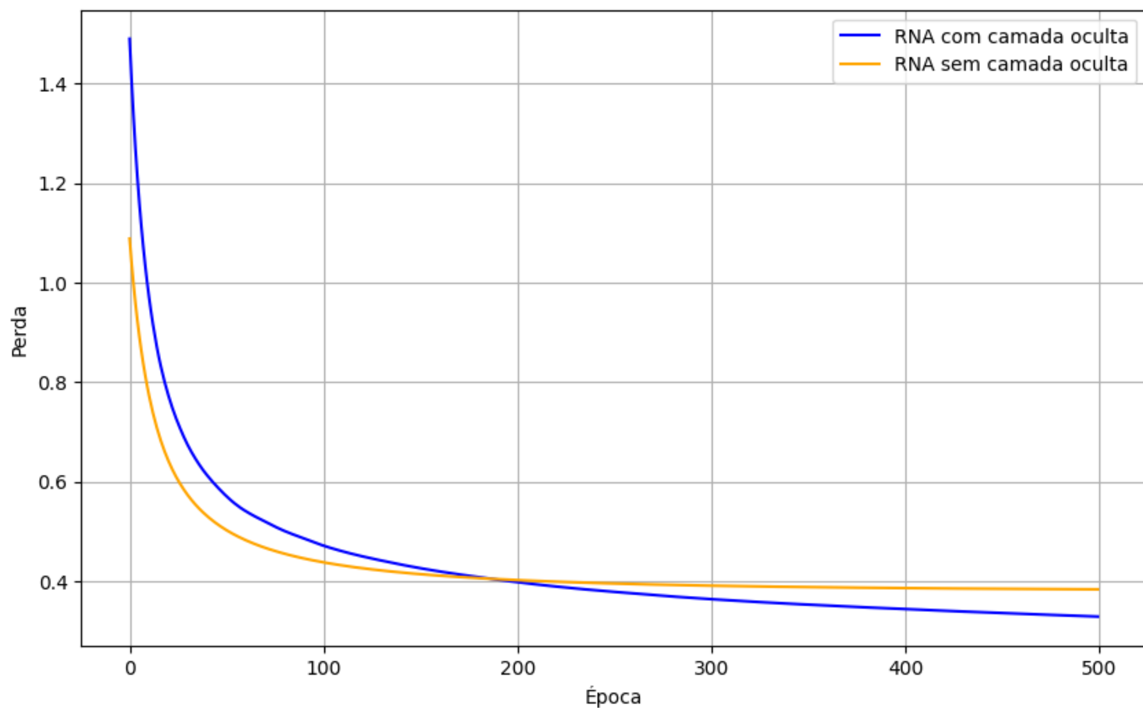


Figura 2.8 Evolução do custo ao longo das épocas para as redes definidas nas Listagens [2.3.3, 2.3.4].

Com base na análise do gráfico acima, durante o treinamento dos dois modelos, é evidente que o modelo com camada oculta superou o modelo sem camada oculta em termos de desempenho. A diminuição mais acentuada do custo nas épocas finais indica uma capacidade de aprendizado mais eficaz por parte do modelo com camada oculta. Essa diferença na trajetória do custo sugere que o modelo com camada oculta foi capaz de capturar representações mais complexas dos dados, permitindo uma melhor adaptação aos padrões subjacentes presentes no conjunto de dados.

É importante ressaltar que, atualmente, existem inicializadores especializados de pesos que podem melhorar o desempenho do modelo, como a inicialização de Xavier, ou Glorot, e a inicialização de He que são descritos em [27, 28]. Esses inicializadores são projetados para manter a estabilidade dos gradientes durante o treinamento mas não serão considerados neste trabalho para manter uma abordagem tradicional.

Uma vez que entendemos como implementar uma rede neural artificial, é imprescindível entender o bom ou mau comportamento do modelo e, para isso, podemos utilizar técnicas que nos permitem identificar não só a acurácia do modelo, mas também, visualizar o desempenho do algoritmo em relação a sua arquitetura. No próximo capítulo, iremos abordar este assunto e realizar análises que nos forneçam algum direcionamento sobre a influência da arquitetura no treinamento das redes.

Capítulo 3

Análise do Desempenho Redes Neurais Artificiais

Como vimos no capítulo anterior, a teoria das redes neurais artificais explora princípios matemáticos de álgebra linear, cálculo diferencial e teoria dos grafos, para representar um processo de aprendizado que é inspirado pelo sistema de conectividade encontrados no cérebro humano, como Haykin descreve em [6].

Vimos também, que a natureza do conjunto de entrada influencia na escolha do tipo de rede neural artificial, como descreve a Seção 2.1 do Capítulo 2. Diferentes tipos de problemas devem ser resolvidos com diferentes tipos de redes neurais, pois a estrutura e o funcionamento das redes neurais podem ser adaptados para melhor se ajustar às características específicas de cada problema.

Neste capítulo, iremos realizar uma análise comparativa da acurácia de diferentes arquiteturas que serão treinadas para resolver um problema de classificação multiclasse. Para isso, utilizaremos múltiplas redes neurais feedforward com hiparâmetros de treinamento como função de ativação, taxa de aprendizagem e outros, fixados. A análise irá abordar desde a evolução da acurácia durante o treinamento, como também em relação à estrutura da rede, como quantidade de neurônios por rede e por camada.

Todos os algoritmos descritos neste capítulo foram executados em uma máquina com 64GB de RAM, utilizando a plataforma CUDA 12.3 e uma placa de vídeo GeForce RTX 1650.

3.1 Descrição do Problema

Ao utilizar redes neurais artificias como ferramenta para resolução de um problema, é tão importante entender o problema quanto como definir a estrutura da rede que irá ser treinada. Uma vez que desejamos realizar uma análise comparativa da acurácia de redes treinadas com diferentes arquiteturas, devemos munir as redes com os mesmos hiperparâmetros, com a mesma quantidade de neurônios nas camada de entrada e saída, e com o mesmo tipo de rede, para que não tenhamos resultados que não possam ser comparados.

Na Subseção 3.1.1, faremos a descrição da base de dados para compreender o problema a ser resolvido e analisado, bem como a estrutura do conjunto de dados que vai alimentar as redes neurais.

3.1.1 Base de Dados

Para atingir os objetivos deste capítulo, iremos utilizar uma amostra do conjunto de dados público "Reconhecimento de Atividades Humanas", do inglês "Human Activity Recognition 70+"ou simplesmente "HAR70+", disponível em [29]. Este conjunto de dados consiste em informações coletadas de 18 idosos, com idades entre 70 e 95 anos, dos quais a partir de determinadas características é possível avaliar se o idoso está andando, deitado, entre outras opções. Tais características são obtidas por meio da utilização de um acelerômetro triaxial fixado na região da coxa direita e outro na região inferior das costas. O acelerômetro triaxial é um dispositivo que mede a aceleração em três direções ortogonais: eixo X, eixo Y e eixo Z, permitindo a detecção de movimento e orientação no espaço tridimensional.

O conjunto de dados original possui 2.259.597 observações, e classifica as atividades dos idosos em diversas categorias, incluindo caminhar, arrastar os pés, subir ou descer escadas, ficar em pé, sentar e deitar. A amostra obtida para fomentar a produção deste trabalho é composta por 16.000 instâncias extraídas do conjunto de dados original, abrangendo informações de 4 idosos. Esta amostra inclui dados das direções ortogonais X e Y, distribuídos em quatro classes: andando (classe 1), descendo escadas (classe 2), em pé (classe 3) e sentado (classe 4). Cada classe, para cada idoso, possui 1.000 instâncias.

A seleção dos critérios para a amostra foi baseada no objetivo deste capítulo, que é comparar o desempenho de redes neurais com hiperparâmetros fixos, mas com diferentes arquiteturas, isto é, diferentes distribuições de neurônios por camada. Por esta razão, a amostra foi extraída de uma forma que a estrutura do conjunto de dados não comprometa o treinamento de forma significativa, o que aconteceria se as classes não tivessem uma distribuição uniforme, pois as redes poderiam favorecer as classes com mais observações, como explica [2].

3.1.2 Escolha da Estrutura das Redes Neurais Artificiais

Uma vez que o problema foi definido na Subseção 3.1.1, vamos à estrutura da nossa rede neural. Definiremos a rede neural do tipo MLP, como descrito na Seção 2.1 do Capítulo 2, com 4 neurônios na camada de entrada, correspondente às direções X e Y dos sensores e 4 neurônios na camada de saída.

Por se tratar de um problema de classificação multiclasse, utilizaremos a função de custo da entropia cruzada, descrita na Definição 2.2.0.3, associada à função de ativação $softmax$ na camada de saída, descrita na Definição 2.2.0.1. Faremos isso para trabalhar em cima da possibilidade de cada classe ocorrer após o treinamento da rede, e tal escolha foi feita de acordo com os fundamentos teóricos discutidos no capítulo anterior.

Antes de prosseguirmos com mais detalhes, também é essencial entendermos como será definida a estrutura do algoritmo de treinamento da nossa rede neural, considerando especialmente que desejamos realizar múltiplos treinamentos. O primeiro passo, após fazer o tratamento do conjunto dados para extrair a amostra, é dividir o conjunto em dados de treino e teste. Para o presente trabalho, utilizamos 80% dos dados do conjunto para treinamento e 20% para teste, que é um senso comum dentro da literatura [6]. Esta divisão não causou nos modelos testados neste trabalho problemas de overfitting e, de fato, a diferença de acurácia entre os resultados obtidos no conjunto de teste e no conjunto compleo são similares.

Na próxima Subseção, vamos definir o conjunto de arquiteturas das camadas ocultas que irão formar o conjunto de redes neurais a serem treinadas para resolução do problema proposto.

3.1.3 Arquitetura das Redes Neurais Artificiais

Visto que o objetivo deste trabalho é realizar uma análise comparativa do desempenho das redes que serão treinadas com base em sua arquitetura, todas as redes devem ter a mesma quantidade de neurônios nas camadas de entrada e saída. Para o problema proposto na Subseção 3.1.1, sabemos que as arquiteturas devem possuir como fator comum quatro neurônios na camada de entrada, que são referentes as direções ortogonais X e Y dos dois acelerômetros utilizados pelos idosos, e quatro neurônios na camada de saída, referente às classes escolhidas para a amostra. Portanto, resta então definir a estrutura que irá armazenar essa arquitetura, incluindo as camadas ocultas. Para isso, vamos utilizar listas. Cada lista dessa, será uma arquitetura de rede neural aritificial para ser treinada com os parâmetros que iremos fixar. Uma rede que possui, por exemplo, quatro neurônios na camada de entrada, duas camadas ocultas contendo três neurônios e quatro neurônios na cada de saída, será

denotada por

$$[4,3,3,4]. \tag{3.1}$$

Como a quantidade de neurônios será fixa nas camadas de entrada e saída para todas as redes, para fins de simplificação vamos considerar apelas os valores da lista que dizem respeito às camadas ocultas. Com isso, a rede $[4,3,3,4]$ pode ser representada simplesmente por $[3,3]$ e iremos utilizar esta notação nesta seção. Dito isso, devemos gerar um conjunto de listas que representarão as camadas ocultas.

Para gerar um conjunto considerável com diferentes arquiteturas, foi feita diferentes combinações e permutações dos números de 1 a 6, onde são consideradas repetições apenas para valores iguais. Para facilitar o entendimento, veremos na prática como o conjunto é formado. Primeiramente, definimos as combinações simples, que contém todas as combinações de um único elemento dos números de 1 a 6:

$$A_1 = \{[1],[2],[3],[4],[5],[6]\}. \tag{3.2}$$

Depois, incluímos todas as permutações possíveis, sem repetições, para combinações de elementos de tamanhos crescentes, variando de 2 até 6:

$$A_2 = \{[1,2],[1,3],...,[2,1],[2,3],...,[6,4],[6,5]\}, \tag{3.3}$$

$$A_{3,6} = \{[1,2,3],[1,2,4],...,[1,2,3,4],[1,2,3,5],...,[1,2,3,4,5],\dots,[6,5,4,3,2,1]\}. \tag{3.4}$$

Por fim, incluímos também as combinações onde um número é repetido de 2 até 6 vezes:

$$A_i = \{[1,1],[1,1,1],\dots,[2,2],[2,2,2],\dots,[6,6,6,6,6,6]\} \tag{3.5}$$

Portanto, o conjunto de arquiteturas das camadas ocultas é formado por:

$$A = A_1 \cup A_2 \cup A_{3,6} \cup A_i. \tag{3.6}$$

Todas essas combinações e permutações são formatadas para garantir que nenhuma apareça mais de uma vez, resultando em uma lista de 1986 arquiteturas diferentes. Com isso, temos no mínimo uma camada oculta e no máximo seis. Note que as camadas de entrada e saída não foram definidas neste conjunto, mas serão definidas posteriormente na Listagem 3.1.12.

Na Listagem 3.1.1, este conjunto é definido e armazenado em um dataframe, que é uma estrutura de dados bidimensional, semelhante a uma tabela, disponível na biblioteca *pandas* em [30].

Na próxima Subseção, estaremos dedicados a implementação computacional de todos os algortimos necessários, desde a definição do conjunto de arquiteturas a implementação do treinamento das redes.

3.1.4 Implementação Computacional

No Capítulo 2, vimos como é feita a implementação de uma rede neural artificial sem recorrer a bibliotecas e módulos que já possuam estruturas definidas para facilitar implementação do algoritmo de treinamento. Uma vez que este capítulo tem o objetivo de treinar múltiplas redes para avaliar o desempenho posteriormente, precisamos definir um algoritmo que possa ser executado com a melhor eficiência possível, uma vez que os nossos recursos computacionais são limitados. Por esta razão, para fins de simplicidade e eficiência, iremos utilizar algumas bibliotecas já desenvolvidas do python, que possuem fundamentações teóricas comprovadas para o desenvolvimento de algoritmos de redes neurais artificiais.

Antes de iniciar, devemos primeiramente definir o algoritmo que gera conjunto de arquiteturas descrito na Subseção 3.1.3. Para isso, utilizaremos os pacotes *permutations* e *combinations* da biblioteca *itertools*, que permitem a geração eficiente de permutações e combinações de elementos de uma lista. A função *combinations* gera todas as combinações possíveis de um conjunto de elementos, onde a ordem não importa. E em permutations, é gerado todas permutações possíveis de uma combinação, onde a ordem importa. Na Listagem 3.1.1, é possível visualizar uma implementação python que gere o conjunto de arquiteturas descrito na Equação 3.6.

Listagem 3.1.1 Gerando Conjunto de Arquitetura das Camadas Ocultas

```
# Importa as bibliotecas necessárias
import pandas
from itertools import permutations, combinations

# Define a lista de números para gerar as combinações e permutações
numeros = [1, 2, 3, 4, 5, 6]

# Lista para armazenar todas as combinações únicas e permutações
lista = []

# Gera arquiteturas de 1  6 camadas ocultas, onde cada camada possui a mesma
    quantidade de neurônios
tamanho_lista = len(numeros)
```

```
for tamanho in range(1, tamanho_lista + 1):
    for numero in numeros:
        lista_numero = [numero] * tamanho
        if combinacao not in lista:
            lista.append(lista_numero)

# Gera arquiteturas de 1  6 camadas ocultas, onde cada camada possui um valor diferente
for tamanho in range(1, len(numeros)+1):
    for combinacao in combinations(numeros, tamanho):
        for permutacao in permutations(combinacao):
            if list(permutacao) not in lista:
                lista.append(list(permutacao))

arquiteturas = pandas.DataFrame({"arquitetura": lista})
```

Uma vez que temos a estrutura inicial das arquiteturas definidas, vamos iniciar o desenvolvimento dos algortimos que irão gerar e treinar as redes neurais artificiais. Para isso, vamos utilizar uma biblioteca de código aberto, chamada PyTorch, para criar as funções que realizarão nosso treinamento.

O PyTorch foi originalmente desenvolvido pela Meta AI (anteriormente conhecida como Facebook AI) em 2016 [31]. Foi criado como uma biblioteca de aprendizado de máquina baseada na biblioteca Torch, com o objetivo de facilitar a pesquisa e o desenvolvimento em inteligência artificial. Em setembro de 2022, a governança do PyTorch foi transferida para a recém-criada PyTorch Foundation, que é parte da Linux Foundation [31]. Esta mudança visou fortalecer a colaboração aberta e o desenvolvimento de software de código aberto.

O PyTorch visa atender tanto às necessidades de pesquisa quanto de produção na área de aprendizado de máquina, incluindo fáceis e versáteis aplicações de redes neurais aritificiais com suporte a GPU's. A linguagem de implementação predominante nos dias atuais é o python, mas o PyTorch também oferece suporte a outras linguagens. Mais detalhes podem ser encontrados em [32].

Antes de prosseguir, devemos entender o que é "class". Uma *class* define um tipo de objeto, especificando os atributos, que são os dados, e métodos, que são as funções, que os objetos desse tipo terão. Em python, uma *class* pode ser utilizada para realizar várias operações e isso ficará bem claro com o Exemplo 3.1.4.1.

Exemplo 3.1.4.1. *Considerando um sistema de gerenciamento de veículos, vamos criar a class Carro para realizar várias operações, como adicionar novos carros, listar todos os carros disponíveis, atualizar informações de carros existentes, etc.*

Listagem 3.1.2 Parte I - Exemplificando Uso do Class

```
class Carro:
# Método inicializador (construtor) da classe
def __init__(self, marca, modelo, ano):
    self.marca = marca # Atributo de instância
    self.modelo = modelo # Atributo de instância
    self.ano = ano # Atributo de instância

# Método da classe
def descricao(self):
    return f"{self.ano} {self.marca} {self.modelo}"
```

Os atributos são variáveis que pertencem a uma classe e descrevem as propriedades de um objeto. Eles são definidos dentro do método inicializador "(__init__)" e são acessíveis através do "self".

Na Listagem 3.1.3, é possível verificar as saídas da class Carro.

Listagem 3.1.3 Parte II - Exemplificando Uso do Class

```
carro1 = Carro("Marca_x", "Modelo_y", 2024)
print(carro1.marca) # Saída: Marca_x
print(carro1.modelo) # Saída: Modelo_y
print(carro1.ano) # Saída: 2024
```

Já os métodos, são funções definidas dentro de uma classe que descrevem o comportamento de seus objetos. Eles também são acessíveis através do self., como descrito na Listagem 3.1.4.

Listagem 3.1.4 Parte III - Exemplificando Uso do Class

```
carro1 = Carro("Toyota", "Corolla", 2020)
print(carro1.descricao()) # Saída: 2020 Toyota Corolla
```

A herança é um princípio que permite criar uma nova classe baseada em uma classe existente. A nova classe (subclasse) herda os atributos e métodos da classe existente (superclasse), conforme apresentado na Listagem 3.1.5:

Listagem 3.1.5 Parte IV - Exemplificando Uso do Class

```
class CarroEletrico(Carro):
def __init__(self, marca, modelo, ano, autonomia_bateria):
    super().__init__(marca, modelo, ano)
    self.autonomia_bateria = autonomia_bateria

def descricao_bateria(self):
    return f"Este carro tem uma autonomia de {self.autonomia_bateria} km com uma ú
        nica carga."
```

Por fim, a intaciação é o processo de criação de um objeto a partir de uma classe. Cada objeto é uma instância única da classe e isso pode ser facilmente compreendido na Listagem 3.1.6.

Listagem 3.1.6 Parte V - Exemplificando Uso do Class

```
carro2 = CarroEletrico("Tesla", "Model S", 2021, 600)
print(carro2.descricao()) # Saída: 2021 Tesla Model S
print(carro2.descricao_bateria()) # Saída: Este carro tem uma autonomia de 600 km com
    uma única carga.
```

Uma vez que temos uma class bem definida, podemos visualizar na Listagem 3.1.7 como usar os métodos e funções dessa class.

Listagem 3.1.7 Parte VI - Exemplificando Uso do Class

```
 # Lista para armazenar os carros
lista_carros = []

# Função para adicionar um carro  lista
def adicionar_carro(marca, modelo, ano):
    novo_carro = Carro(marca, modelo, ano)
    lista_carros.append(novo_carro)

# Função para listar todos os carros
def listar_carros():
    for carro in lista_carros:
        print(carro.descricao())

# Adicionando carros  lista
```

```
15 adicionar_carro("Toyota", "Corolla", 2020)
16 adicionar_carro("Honda", "Civic", 2019)
17
18 # Listando todos os carros
19 listar_carros()
20 # Saída:
21 # 2020 Toyota Corolla
22 # 2019 Honda Civic
```

Em resumo, uma classe em programação orientada a objetos, como o python, é uma estrutura que permite agrupar dados e comportamento relacionado, facilitando a criação de objetos complexos e reutilizáveis.

Agora, podemos então definifir a *class* RNA, de rede neural artificial, que herda de *nn.Module*, que é a *class* base para todos os pacotes de rede neural no PyTorch. Para isso, devemos importar os pacotes *torch.nn* e *torch.optim* do PyTorch. Na Listagem 3.1.8, é possível visualizar a implementação python para criar a nossa class RNA, que receberá como parâmetros a arquitetura da rede e as suas respectivas funções de ativação.

Listagem 3.1.8 Definindo Class RNA

```
1 # Importa o pacote torch.nn
2 import torch.nn as nn
3
4 # Define da class python RNA que herda de "nn.Module" do PyTorch
5 class RNA(nn.Module):
6     # Método inicializador da classe, que recebe os tamanhos das camadas e funções de
          ativação
7     def __init__(self, camadas, funcoes_ativacao):
8         # Chama o método inicializador da classe base
9         super(RNA, self).__init__()
10        # Inicializa uma lista de pacotes para armazenar as camadas da rede
11        self.camadas = nn.ModuleList()
12        # Itera sobre os tamanhos das camadas para criar as conexões
13        for i in range(len(camadas) - 1):
14            # Adiciona uma camada Linear  lista de camadas com camadas[i+1]
                  neurônios conectados  camada camadas[i]
15            self.camadas.append(nn.Linear(camadas[i], camadas[i+1]))
16        # Armazena as funções de ativação
```

```
17            self.funcoes_ativacao = funcoes_ativacao
18
19        # Define a fase de propagação
20        def propagacao(self, x):
21            # Itera sobre todas as camadas exceto a última
22            for i, camada in enumerate(self.camadas[:-1]):
23                # Aplica a combinação linear da camada
24                x = camada(x)
25                # Se houver uma função de ativação definida para a camada
26                if self.funcoes_ativacao[i]:
27                    # Aplica a função de ativação
28                    x = self.funcoes_ativacao[i](x)
29            # Aplica a última camada linear
30            x = self.camadas[-1](x)
31            # Retorna a saída da rede
32            return x
```

A última camada consiste apenas na aplicação da combinação linear, que é a mesma descrita na Equação 2.1, porque a função de custo escolhida é a entropia e cruzada e, no PyTorch, a função que faz essa aplicação é a CrossEntropyLoss, que espera receber valores brutos da saída da rede, em vez de probabilidades, que é o que aconteceria se aplicassemos a softmax na última camada. A questão é que a função CrossEntropyLoss aplica internamente a função softmax para converter esses valores brutos em probabilidades e só depois calcula a perda de entropia cruzada com os rótulos fornecidos. Este procedimento auxilia na aceleração do treinamento.

Perceba que definimos a class da nossa rede na Listagem 3.1.8, de forma que os parâmetros são as camadas, que esperam receber lista do python onde por exemplo,

$$[4,6,2,5,4], \tag{3.7}$$

representa uma rede contendo a camada de entrada com quatro neurônios, três camadas ocultas contendo seis, dois e cinco neurônios respectivamente, e a camada de saída com quatro neurônios. Esta notação está perfeitamente alinhado com a definição do nosso conjunto de arquiteturas das camdas ocultas descrito na Equação 3.6.

Além disso, a *class* RNA recebe também uma lista semelhante, onde cada entrada corresponde a uma função de ativação a ser aplicada, e pela forma como o algoritmo está sendo definido, para n camadas, devemos ter $n-2$ funções de ativações.

Para fins de otimização, na fase de retropropagação, vamos utilizar o gradiente descendente em mini-lote, descrito na Subseção 1.2.4 do Capítulo 1, que também irá permitir uma execução mais rápida do algoritmo, além de consumir menos memória. Vale ressaltar, também, que o PyTorch trabalha com tensores e por isso devemos converter os conjuntos para tensores antes de iniciar o treinamento. Utilizaremos para isso, os pacotes *DataLoader* e *TensorDataset*, encontrados em [32], que irão efetuar a divisão do conjunto em mini-lote e a conversão para tensores, respectivamente.

Os tensores no PyTorch são estruturas de dados fundamentais que representam *arrays* multidimensionais e são representados de forma semelhante à representação matemática de uma matriz ou um vetor, mas com suporte a múltiplas dimensões. Por exemplo, um tensor de ordem zero é um único número, um tensor de primeira ordem é uma sequência de números organizados em uma linha ou coluna, como vetor, e um tensor de segunda ordem é uma grade retangular de números organizados em linhas e colunas, como matrizes. Um tensor de ordem superior, são como matrizes de matrizes.

Uma importante etapa no construção do treinamento de redes neurais é a fase de treinamento, que foi discutida no capítulo anterior. Na Listagem 3.1.9, a função de treinamento é implementada.

Listagem 3.1.9 Definindo Função de Treinamento

```
1  # Importação de pacotes
2  import torch.optim as optim
3  from torch.utils.data import DataLoader, TensorDataset
4
5  # Define função para treinar o modelo
6  def treinar_modelo(modelo, X_treino, y_treino, taxa_aprendizado, tamanho_lote,
       otimizador, momento, criterio_parada, limiar_parada, max_epocas):
7      # Cria um conjunto de dados de tensores a partir das amostras de treino
8      conjunto_treino = TensorDataset(X_treino, y_treino)
9
10     # Cria um carregador de conjuntos que geram os mini−lotes, embaralhando−os para
           que a rede não aprenda a sequência dos conjuntos
11     carregador_treino = DataLoader(dataset=conjunto_treino, batch_size=tamanho_lote,
           shuffle=True)
12
13     # Verifica o otimizador escolhido
14     if otimizador == "SGD":
15         if momento is not None:
```

```
16              # Cria o otimizador SGD com momentum
17              otimizador = optim.SGD(modelo.parameters(), lr=taxa_aprendizado,
                    momentum=momento)
18          else:
19              # Cria o otimizador SGD sem momentum
20              otimizador = optim.SGD(modelo.parameters(), lr=taxa_aprendizado)
21      else:
22          # Lança um erro se o otimizador não for suportado
23          raise ValueError("Otimizador não suportado")
24
25      # Define o critério de perda como CrossEntropyLoss
26      criterio = nn.CrossEntropyLoss()
27
28      # Define dicionário para armazenar os dados do treinamento
29      dados_treinamento = {"epoca": [], "perda": [], "acuracia": [], "duracao_epoca": []}
30
31      # Loop de treinamento por época
32      for epoca in range(max_epocas):
33          # Coloca o modelo em modo de treinamento
34          modelo.train()
35
36          # Inicializa variáveis de acúmulo
37          perda_total, acuracia_total, amostras_total = 0, 0, 0
38
39          # Fase de treinamento
40          for X_lote, y_lote in carregador_treino:
41              # Zera os gradientes do otimizador
42              otimizador.zero_grad()
43
44              # Calcula a saída do modelo
45              saida = modelo(X_lote)
46
47              # Calcula a perda do lote
48              perda = criterio(saida, y_lote)
49
50              # Calcula os gradientes
```

```
51                 perda.backward()
52
53                 # Atualiza os pesos do modelo
54                 otimizador.step()
55
56                 # Acumula a perda ponderada pelo tamanho do lote
57                 perda_total += perda.item() * X_lote.size(0)
58
59                 # Conta as previsões corretas
60                 previsoes_corretas = (saida.argmax(dim=1) == y_lote).float().sum()
61
62                 # Acumula o número de previsões corretas
63                 acuracia_total += previsoes_corretas.item()
64
65                 # Acumula o número de amostras processadas
66                 amostras_total += X_lote.size(0)
67
68             # Calcula a perda média por amostra
69             perda_media = perda_total / amostras_total
70
71             # Calcula a acurácia média em porcentagem
72             acuracia_media = (acuracia_total / amostras_total) * 100
73
74             # Armazena os dados do treinamento
75             dados_treinamento["epoca"].append(epoca)
76             dados_treinamento["perda"].append(perda_media)
77             dados_treinamento["acuracia"].append(acuracia_media)
78
79             # Verifica o critério de parada
80             if criterio_parada == "perda" and perda_media < limiar_parada:
81                 break
82             elif criterio_parada == "acuracia" and acuracia_media > limiar_parada:
83                 break
84
85         # Retorna o modelo treinado e os dados do treinamento
86         return modelo, dados_treinamento
```

A Listagem 3.1.9 cria mini-lotes de dados de treinamento, configura um otimizador de gradiente descendente estocástico, com ou sem momento, calcula o custo usando CrossEntropyLoss, e atualiza os pesos do modelo ao longo das épocas, até que um critério de parada escolhido seja atendido.

É importante ressaltar que todos os pesos das redes foram inicializados com a mesma semente de geradores de números aleatórios e, portanto, não estudaremos os efeitos dos inicializadores das redes nos resultados que obtivermos.

Uma outra função que devemos definir no algoritmo é a que irá efetuar o cálculo da acurácia. Para tal, basta converter os conjuntos em tensores e comparar as previsões da rede com as reais. A Listagem 3.1.10 define tal função.

Listagem 3.1.10 Definindo Função Para Calcular Acurácia

```
# Define função para calcular a acurácia do modelo
def calcular_acuracia(modelo_treinado, X, y):
    # Desabilita o cálculo de gradientes
    with torch.no_grad():
        # Converte os dados de entrada para tensor
        X_tensor = torch.tensor(X)
        # Calcula as previsões do modelo
        previsoes = modelo_treinado(X_tensor)
        # Obtém as classes previstas e converte para numpy
        classes_previstas = torch.argmax(previsoes, dim=1).numpy()
        # Se y for um tensor, converte para numpy
        if isinstance(y, torch.Tensor):
            y = y.cpu().numpy()
        # Calcula a acurácia comparando as previsões com os rótulos reais
        acuracia = np.mean(classes_previstas == y)
    # Retorna as classes previstas e a acurácia
    return classes_previstas, acuracia
```

Como nosso objetivo é treinar múltipas arquiteturas de redes neurais diferentes e analisá-las posteriomente, devemos definir a função que orquestrará a execução do treinamento dessas redes, passando os parâmetros adequados para todas as funções definidas até aqui e armazenando os resultados, que para este trabalho será em um json, que chamaremos de "configuracoes". Um json é um formato de texto que usa a notação de objetos da linguagem de programação JavaScript, em que os objetos são representados por chaves e contêm pares de chave-valor, onde cada chave é seguida por seu valor correspondente. Além disto os

valores são representados por colchetes e contêm uma lista ordenada de observações, como mostra o Exemplo 3.1.4.2.

Exemplo 3.1.4.2. *Um json contendo a chave arquitetura, cujo valor é a lista contendo a arquitetura, é dado por:*

$$\{''arquitetura'' :'' [4,3,2,4]''\} \tag{3.8}$$

O algoritmo python desta função de orquestramento para treinar várias redes e armazenar as informações pode ser visualizado na Listagem 3.1.11.

Listagem 3.1.11 Definindo Função Para Treinar Múltiplas Redes

```
# Define função para treinar e avaliar redes neurais artificiais com diferentes configurações
def testar_varios_modelos(configuracoes):
    # Lista para armazenar os resultados
    resultados = []

    # Loop sobre cada configuração
    for config in configuracoes:
        # Cria o modelo RNA com a configuração atual
        modelo = RNA(config["camadas"], config["funcoes_ativacao"])

        # Treina o modelo
        modelo_treinado, historico_treinamento = treinar_modelo(modelo, X_treino,
            y_treino, **config["parametros_treino"])

        # Calcula a acurácia do modelo treinado nos conjuntos de teste e também em
            todo o conjunto (treino + teste)
        _, acuracia_conjunto_teste = calcular_acuracia(modelo_treinado, X_teste,
            y_teste_acuracia)
        # Calcula a acurácia do modelo treinado em todo o conjunto de dados
        _, acuracia_conjunto_completo = calcular_acuracia(modelo_treinado, X,
            y_indice)

         # Armazena os resultados para a configuração atual com precisão de 3 casas
             decimais
        resultados.append({
            "nomenclatura": config["camadas"],
            "config": config,
```

```
                "total_parametros": total_params,
                "acuracia_final_teste": f"{acuracia_conjunto_teste * 100:.3f}%",
                "acuracia_final_completo": f"{acuracia_conjunto_completo * 100:.3f}%",
                "historico_treinamento": historico_treinamento
            })
    # Retorna a lista de resultados
    return resultados
```

Durante a fase de treinamento da Listagem 3.1.11, note que são armazenados tanto a acurácia no conjunto de teste, representado pela variável "acuracia_conjunto_teste", quanto o conjunto completo incluindo os dados do conjunto de treino, representado pela variável "acuracia_conjunto_completo". Agora, devemos montar nosso json com as configurações desejadas.

Listagem 3.1.12 Definindo Configurações de Treinamento das Redes

```
# Inicializa uma lista vazia para armazenar as configurações dos modelos
configuracoes = []

# Define um dicionário com os parâmetros de treinamento, que serão fixados para todas as
    configurações
parametros_treino = {
    "taxa_aprendizado": taxa_escolhida,
    "tamanho_lote": tamanho_escolhido,
    "escolha_otimizador" == "SGD", # único suportado
    "momento": momento_escolhido,
    "criterio_parada": criterio_escolhido,
    "limiar_parada": limiar_escolhido,
    "max_epocas": quantidade_escolhida
}

# Itera sobre cada linha do dataframe arquiteturas
for index, row in arquiteturas.iterrows():
    # Define a arquitetura final das camadas de cada rede
    arquitetura_final = [4] + row["arquitetura"] + [4]
    # Adiciona a arquitetura final na em configuracoes
    configuracoes.append({
        "camadas": arquitetura_final
```

```
22      })
23
24  # Itera sobre cada configuração para completar as definições necessárias
25  for config in configuracoes:
26      # Calcula o número de funções de ativação (uma para cada camada oculta)
27      num_funcoes_ativacao = len(config["camadas"]) - 2
28      # Define a lista de funções de ativação (ReLU) para todas as camadas ocultas
29      config["funcoes_ativacao"] = [torch.nn.ReLU()] * num_funcoes_ativacao
30      # Adiciona os parâmetros de treino a configuração
31      config["parametros_treino"] = parametros_treino
32
33  # Aplica a função testar_varios_modelos para treinar as redes
34  resultados = testar_varios_modelos(configuracoes)
35  df_resultados = pd.DataFrame(resultados)
```

Uma vez que o algoritmo está definido, é possível iniciar a análise dos resultados. Nas próximas seções, iremos abordar análises dos resultados dos processos descritos aqui. Para isso, faremos um comparativo em relação à arquitetura das redes definidas na Listagem 3.1.1, e à acurácia das redes após o treinamento, que por sua vez teve todos os seus hiperparâmetros fixados em todas as configurações, tal como foi descrito na Listagem 3.1.12.

3.2 Análise da Acurácia

A acurácia é uma métrica fundamental na avaliação desempenho de ajustes, algoritmos e modelos. A medida quantifica a capacidade do modelo de prever corretamente os resultados de um dado problema. No contexto do problema que está sendo descrito neste capítulo, a medida quantificará o quanto o modelo prevê corretamente as classes no conjunto de dados de teste.

Neste capítulo, faremos as análises com base na acurácia do conjunto completo, a qual chamaremos apenas de acurácia para simplificação. Foram aproximadamente oito dias e duas horas para concluir o treinamento de todos os 1986 modelos, sem levar em consideração o tempo de leitura, tratamento e armazenamento de dados.

Para iniciar as análises, vamos avaliar a evolução da acurácia ao longo das épocas de treinamento. Os valores para os quais os hiperparâmetros foram fixados podem ser vistos na Tabela 3.1. Essa fixação nos permite garantir a compatibilidade entre as redes, que possuem características de treinamento semelhantes, exceto pela distribuição de neurônios e

quantidades de camadas. Assim, podemos comparar diretamente o impacto dessas variações na performance das redes. Além disso, essa abordagem ajuda a minimizar a influência de outros fatores que poderiam distorcer os resultados, permitindo uma análise mais precisa do comportamento do desempenho das redes.

Tabela 3.1 Parâmetros Fixos de Treinamento.

Parâmetro	Valor
Taxa de Aprendizado	0.01
Tamanho do Lote	500
Escolha do Otimizador	SGD
Momento	0.9
Critério de Parada	Perda
Limiar de Parada	0.1
Máximo de Épocas	5000

Na Figura 3.1, é possível observar a evolução da média da acurácia de todos os treinamentos ao longo das épocas. Note que há uma evolução positiva no treinamento, embora seja lenta. Isso indica a necessidade de treinamentos mais longos para as redes estudadas neste trabalho, os quais não puderam ser realizados por limitações computacionais.

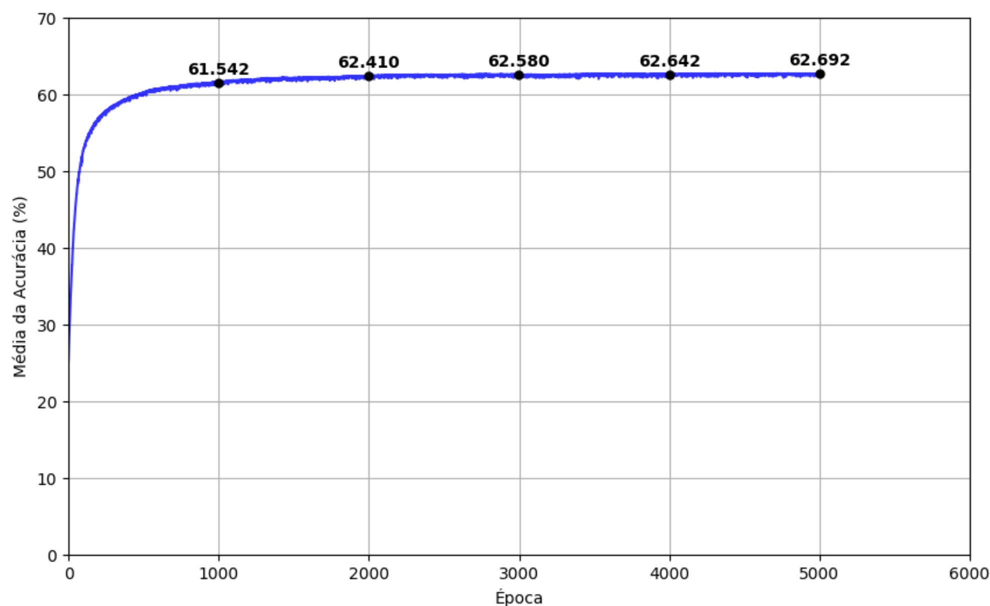


Figura 3.1 Evolução da acurácia média de todos os treinamentos ao longo das épocas.

É fundamental considerar a quantidade de modelos treinados, pois modelos com desempenho abaixo do esperado podem comprometer a média geral. Analisar esses aspectos pode ajudar a entender melhor o comportamento e a eficácia dos modelos ao longo do tempo. Ajustes adicionais nos parâmetros de treinamento também podem ser viáveis para otimizar os resultados. Discutiremos um pouco sobre isto na Seção 3.4.

Dito isso, vamos definir um limiar para determinar modelos com boa e má performance. Para este presente trabalho, vamos usar o limiar de 50% em relação a acurácia, onde estritamente abaixo de 50% significam um desempenho ruim, e caso contrário, um bom desempenho.

Na Tabela 3.2 é possível visualizar os indicadores que fornecem uma visão abrangente do desempenho dos modelos com base na acurácia, permitindo uma comparação clara entre os dois grupos. Os indicares de média e desvio padrão foram arredondados com precisão de três casas decimais e indicam que aproximadamente 43.35% das redes treinadas não tiveram boa perfomance, cuja média da acurácia é 25.263% e um desvio padrão de 2.525%, o que demonstra uma quantidade considerável de modelos com desempenho baixo em que há pouca variação da acurácia. Isso nos mostra que houve algum problema no treinamento desses modelos e isso será discutido com mais detalhes na Seção 3.4. Por enquanto, iremos seguir as análises focando nos modelos que tiveram bom desempenho. Utilizaremos estes modelos para extrair informações da acurácia em relação às suas arquiteturas.

Tabela 3.2 Tabela de desempenho de modelos.

Desempenho	Quantidade	Média	Desvio Padrão	Mínimo	Máximo
Ruim	861	25.263	2.525	25.00	49.97
Bom	1125	91.114	5.855	50.00	95.93

Nos indicadores da Tabela 3.2, podemos ver que há 1125 modelos com bom desempenho e com desvio padrão de 5.855%, o que demonstra variações não tão grandes da média. De fato, vemos que algumas redes que têm acurácia de 50%, porém a acurácia máxima obtida está bem próxima da média.

Na Figura 3.2, podemos acompanhar a evolução da média da acurácia ao longo das épocas para os modelos com bom desempenho. Perceba que a média das redes com boa acurácia é bastante elevada, indicando uma boa escolha, em geral, da configuração dos modelos para tratar os dados. Já a evolução da acurácia ao longo do treinamento para os modelos com a melhor e pior performance, apontadas pela Tabela 3.2, é apresentada na Figura 3.3, juntamente com a única rede que obteve exatamente 50% de acurácia.

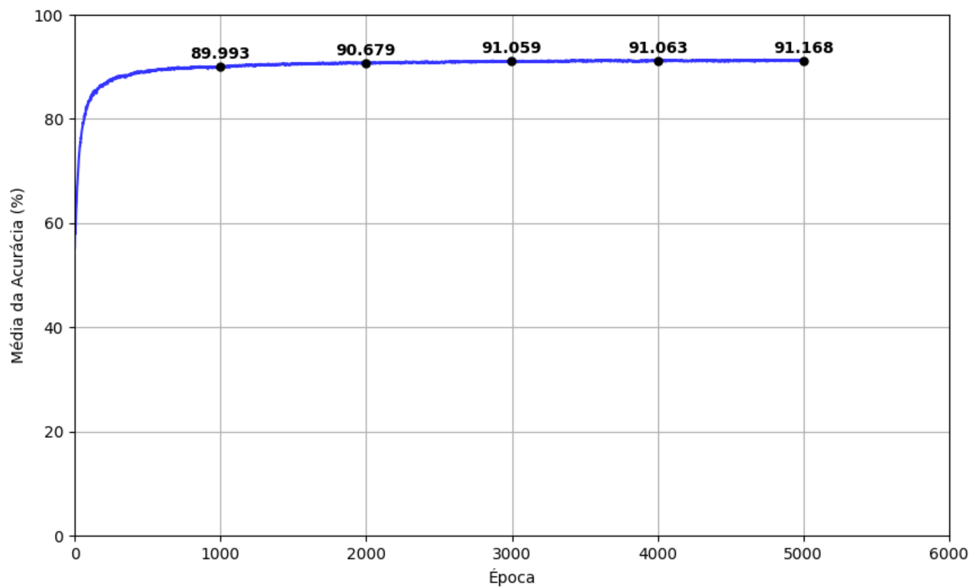


Figura 3.2 Evolução da acurácia média do treinamento de modelos considerados com bom desempenho ao longo das épocas.

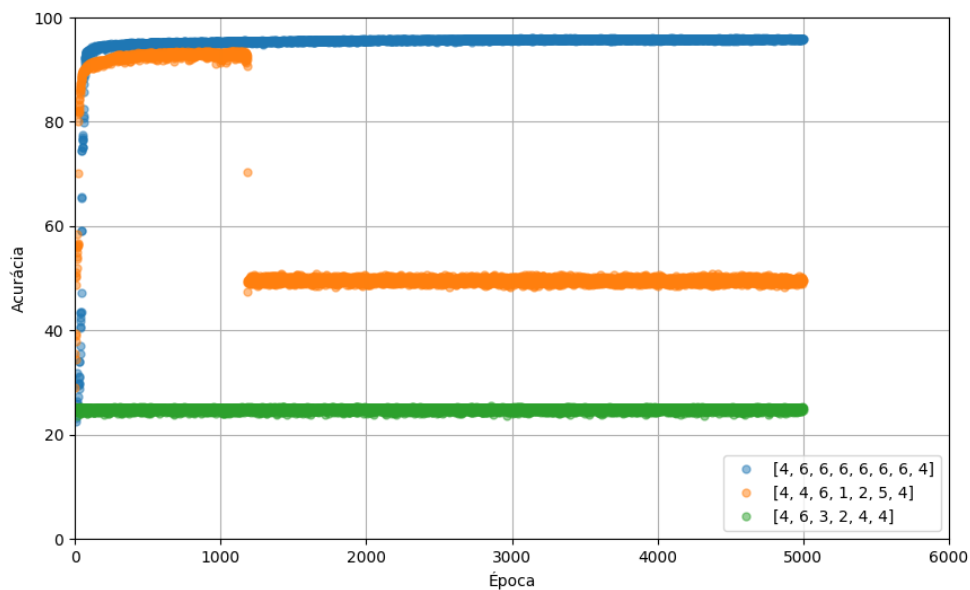


Figura 3.3 Evolução da acurácia ao longo do treinamento de modelos com melhor, pior e intermediária performance.

Ainda sobre a Figura 3.3, é importante ressaltar que, neste caso, há mais de um modelo com a pior acurácia de 25% e por esta razão, foi escolhido aleatoriamente um desses modelos. Além disso, note que temos três pontos de discussão. O primeiro é que o modelo com melhor perfomance foi aquele que contém a maior quantidade de camadas e a maior quantidade de neurônios. O modelo intermediário, por sua vez, passou por cerca de 1000 épocas com acurácia acima de 90%, mas não atingiu o critério de parada. Isso pode ter levado o gradiente descendente a sair do rumo certo e convergir para um mínimo local, em vez de um mínimo global. Por fim, vemos que o modelo com má performance foi prejudicado durante o treinamento e não conseguiu melhorar sua acurácia, provavelmente atingindo um mínimo local ruim logo ínicio do treinamento.

O modelo [4,4,6,1,2,5,4], traçado em laranja, foi acrescentado ao gráfico para mostrar que, durante o treinamento, é possível "estragarmos" a acurácia da rede neural. Discutiremos isso com mais detalhes posteriormente na Seção 3.4 junto com modelos que foram prejudicados de alguma forma durante a fase de treinamento. Quanto ao modelo de melhor performance, é válido questionar se mais camadas e/ou neurônios contribuíram para isso. Observando os modelos de bom desempenho, não foi possível concluir sobre o número de camadas, mas o aumento de neurônios mostrou tendência de crescimento da acurácia média, como mostra a Figura 3.4.

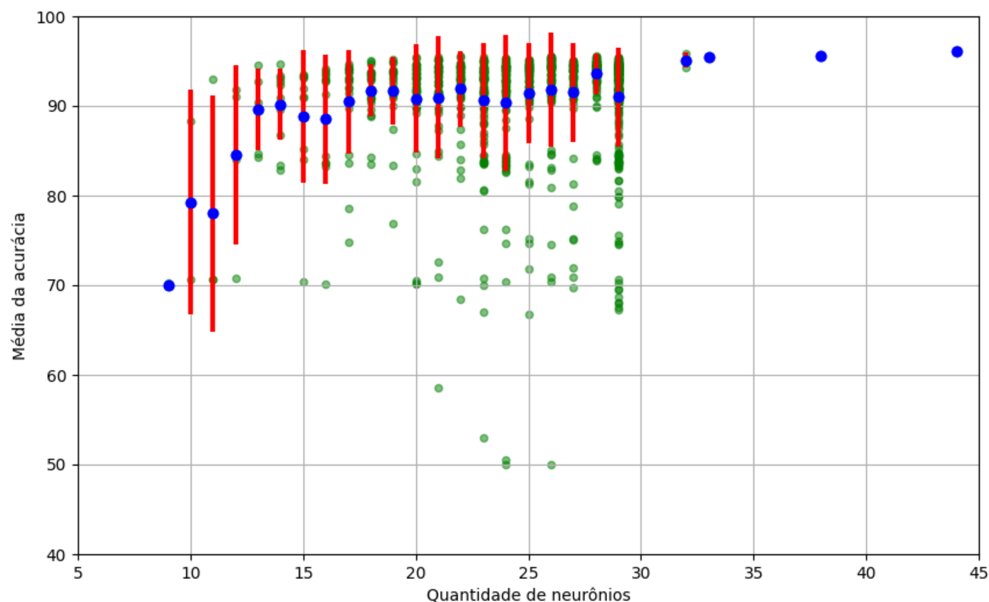


Figura 3.4 Média da acurácia com base na quantidade de neurônios das redes com barra de erro do desvião padrão.

Os pontos na cor verde representam a dispersão de valores da acurácia das redes avaliadas de cada grupo de quantidade de neurônios. Note que, para as o grupos com menos e mais neurônios, existe apenas uma rede de cada e, portanto, não há uma barra de erro associada à acurácia.

Resultados semelhantes podem ser obtidos em relação ao total de parâmetros de cada modelo, como descrito na Observação 2.1.0.5. A quantidade total de parâmetros está diretamente ligada à quantidade de neurônios da rede, influenciando de maneira direta no recurso computacional que é necessário ter disponível para o processamento dos algoritmos, pois quanto mais parâmetros, mais lento será o treinamento.

Na Figura 3.5 podemos analisar o comportamento da média da acurácia em relação a essa variável para os modelos com bom desempenho. No entanto, as barras de erro não são mostradas por questões de simplicidade. Podemos notar que na Figura 3.5 parece existir um valor crítico de número de parâmetros, a saber 142, a partir do qual a acurácia assume valores 95%. Porém, devido a limitações computacionais, não foi possível gerar muitas redes com uma grande quantidade de parâmetros e, portanto, estatisticamente não é possível afirmar que seja, de fato, o caso. Esse ponto crítico indica um possível limite de eficiência a partir do qual a adição de mais parâmetros traz benefícios significativos. A compreensão desses aspectos pode ajudar na escolha de modelos mais eficientes.

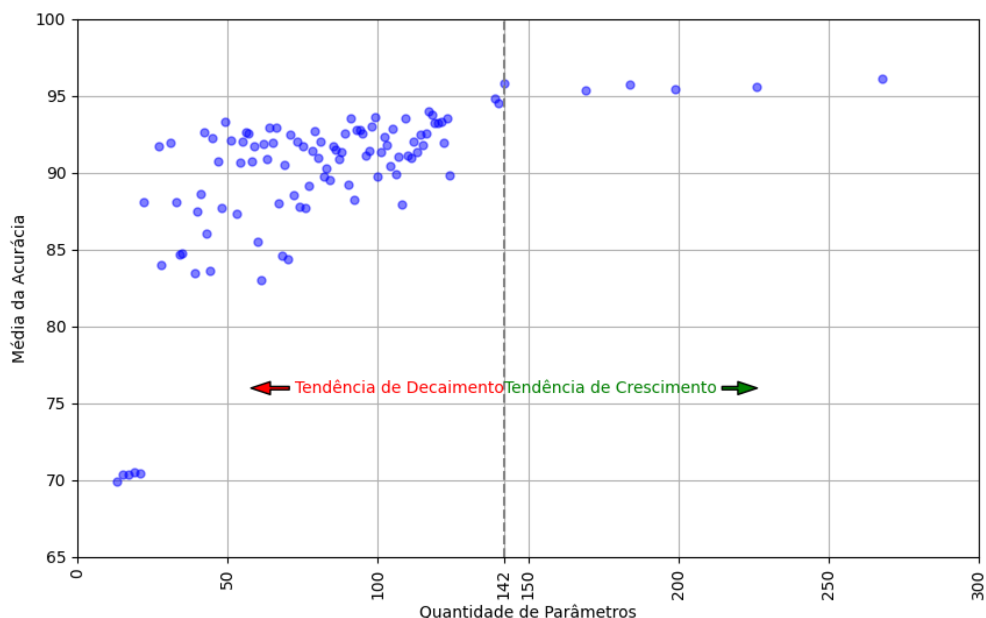


Figura 3.5 Média da acurácia com base na quantidade de parâmetros.

Como veremos mais para frente, a grande flutuação de acurácia para redes que tenham menos parâmetros que o número crítico 142 se deve a problemas de treinamento da redes, similares ao que aconteceu com a rede representada pela cor laranja na Figura 3.3.

Portanto, os resultados obtidos até agora indicam que é possível que arquiteturas que tenham um número significativo, ou suficiente, de parâmetros se comportarão de maneira mais robusta na fase de treinamento.

Na seção seguinte, apresentaremos análises detalhadas da acurácia em relação às lacunas existentes entre as camadas. Tais lacunas são determinadas pela quantidade de neurônios em cada camada. Essas análises nos permitirão verificar como a acurácia dos modelos com bom desempenho se comporta em relação à arquitetura da rede.

3.3 Análise da Acurácia em Relação a Disparidade de Dimensionalidade

Embora seja desafiador, durante o processo de análise foi possível notar algumas características que puderam ser associadas ao comportamento observado dos resultados. Uma delas é a disparidade de dimensionalidade em relação as camadas, que nada mais é do que as lacunas que existem entre uma camada e outra, determinada pela diferença de neurônios existentes camadas subsequentes.

Considere a rede neural [4, 8, 2]. Entre a camada de entrada e a única camada oculta há uma disparidade de tamanho quatro. Similarmente, da camada oculta para a de saída há uma disparidade de tamanho seis. Note, que a rede recebe inicialmente quatro informações e deve expandi-las para oito neurônios. No final, a rede deverá contrair as oito informações para apenas duas. Isto gera uma sequência do tipo expansão-contração na rede. Portanto, podemos questionar quando é possível obter um melhor desempenho do modelo em termos da acurácia: seguindo uma sequência de expansão-contração ou contração-expansão.

Partindo desta ideia, nesta seção faremos uma análise sobre o comportamento da acurácia dos modelos treinados em relação a disparidade de dimensionalidade existentes. Salvo lapso da autora, até o presente momento é desconhecida uma análise similar na literatura. O conceito de expansão e contração são bastante intuitivos, mas antes de prosseguir, iremos definir formalmente quando temos expansão e contração local em uma rede neural artificial. Primeiramente, vamos entender o que é o tamanho de uma lacuna.

Definição 3.3.0.1. *Dada uma rede neural artificial, seja ϑ_i a quantidade de neurônios da camada i. O tamanho de uma lacuna na camada $i+1$, é definido como sendo a diferença entre ϑ_i e ϑ_{i+1}, podendo assumir valores negativos e positivos.*

Definição 3.3.0.2. *Dada uma rede neural artificial, seja ϑ_i a quantidade de neurônios da camada i. Diremos que existe uma contração local quando existir um i tal que*

$$v_i - v_{i+1} = m,\ m > 0.$$

Quando $m < 0$, dizemos que há uma expansão local. Se $m = 0$, não há expansão, nem contração.

Observação 3.3.0.3. *Note que uma rede neural artificial possui uma contração local sempre que existir um tamanho de lacuna positivo. Analogamente, uma rede neural possui uma expansão local sempre que possuir um tamanho de lacuna negativo.*

Definição 3.3.0.4. *Seja uma rede neural artificial, que possui contração e/ou expansão local. Seja $A = \{m_1, \ldots, m_n\}$ o conjunto formado pelos tamanhos de lacunas. O tamanho de maior disparidade dimensional dessa rede é dado por*

$$\max_{i=1,\ldots,n} |m_i|. \tag{3.9}$$

Exemplo 3.3.0.5. *Considere a rede neural artificial, [4, 9, 5]. Nela, ocorre uma expansão local com tamanho de lacuna -5 entre a camada de entrada e a camada oculta, e uma contração local com tamanho de lacuna 4 entre a camada oculta e a camada de saída. O tamanho de maior disparidade dimensional neste caso, é 5.*

Agora, vamos analisar a média da acurácia dos modelos em relação aos tamanhos de maior disparidades existentes nas redes neurais que foram treinadas na seção anterior. Na Figura 3.6, podemos ver que a tendência é que quanto mais alto for o maior tamanho de disparidade, menor tende a ser a acurácia média. Isso nos mostra que, com expansão ou contração impactante na arquitetura, o treinamento da rede demonstrou uma tendência a ser prejudicado.

Os dados da Figura 3.6 mostram que deve-se evitar, dentro do contexto das redes tratadas neste trabalho, um tamanho de maior disparidade maior ou igual a três. Isso é evidenciado pela queda na média da acurácia e pelo aumento do desvio padrão à medida que o tamanho da lacuna aumenta para três ou mais, conforme mostrado na figura.

De forma natural, o leitor deve se perguntar o que acontece quando não há nenhum tipo de disparidade dimensional na rede. As redes sem disparidade são aquelas sem contração e nem expansão local. No contexto deste trabalho, estas são as redes que contêm todas as camadas com quatro neurônios, que obtiveram acurácia acima de 90%, como pode ser visto na Tabela 3.3.

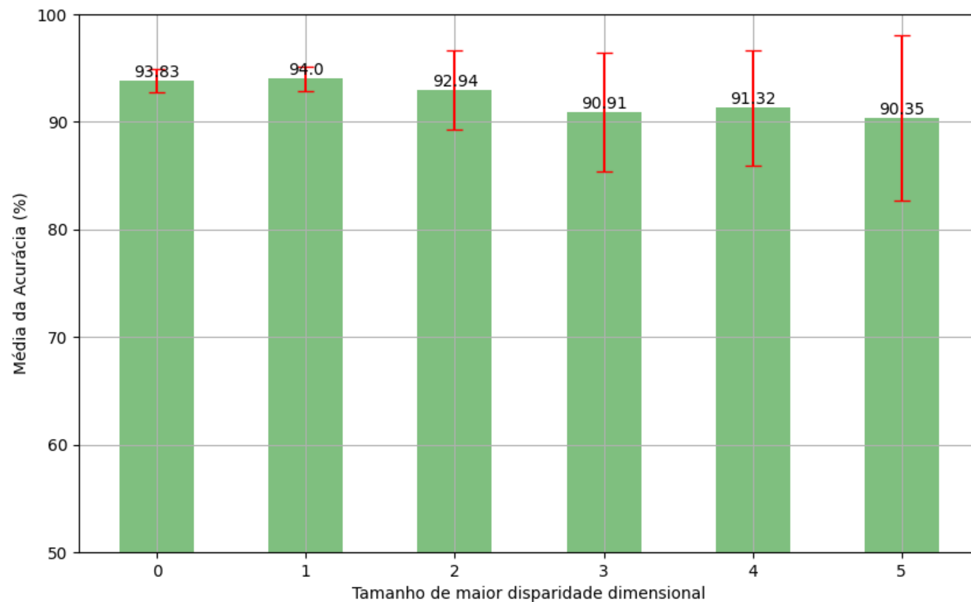


Figura 3.6 Média da acurácia em relação ao tamanho de maior disparidade dimensional das redes com barra de erro do desvio padrão.

Tabela 3.3 Acurácia das redes neurais sem contração nem expansão.

Arquitetura	Acurácia
$[4,4,4]$	91.77%
$[4,4,4,4]$	93.40%
$[4,4,4,4,4]$	94.44%
$[4,4,4,4,4,4]$	94.36%
$[4,4,4,4,4,4,4]$	94.59%
$[4,4,4,4,4,4,4,4]$	94.31%

Vamos investigar, agora, casos de contração e expansão locais existentes nas extremidades da rede, isto é, investigar o comportamento da acurácia em relação às disparidades existentes nas primeiras e últimas camadas. Na Figura 3.7, podemos analisar tal comportamento na primeira camada oculta. Note que ao compararmos modelos com contração e expansão local na primeira camada oculta da Figura 3.7, vemos que é mais vantajoso expandir do que contrair. Uma contração de tamanho 3 já causa uma perda significativa na acurácia média.

Por outro lado, de forma oposta ao ponto observado na Figura 3.7, vemos que há uma tendência a ser melhor contrair na última camada, como mostra a Figura 3.8, indicando que as previsões melhoram quando concentramos a informação contida em mais neurônios.

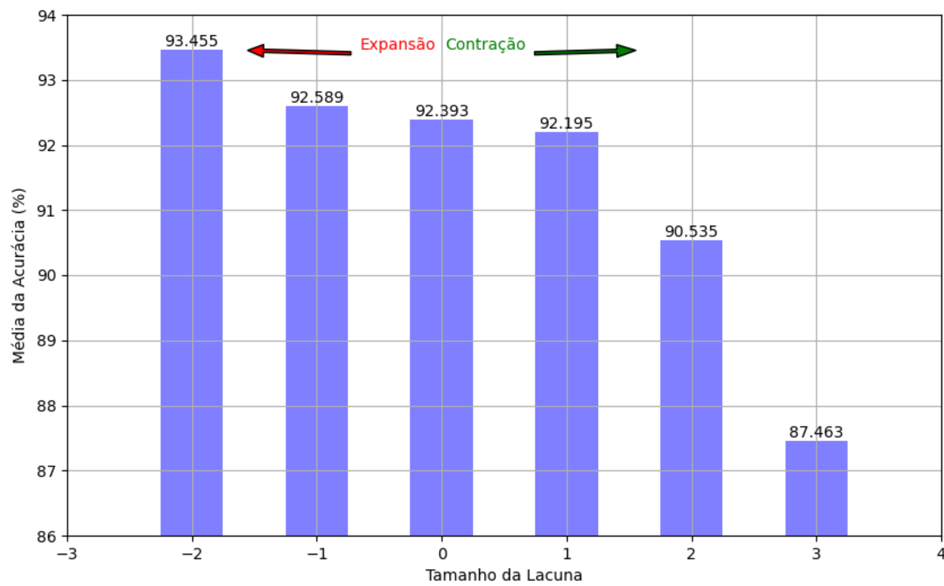


Figura 3.7 Média da acurácia em relação ao tamanho das lacunas na primeira camada oculta das redes neurais treinadas.

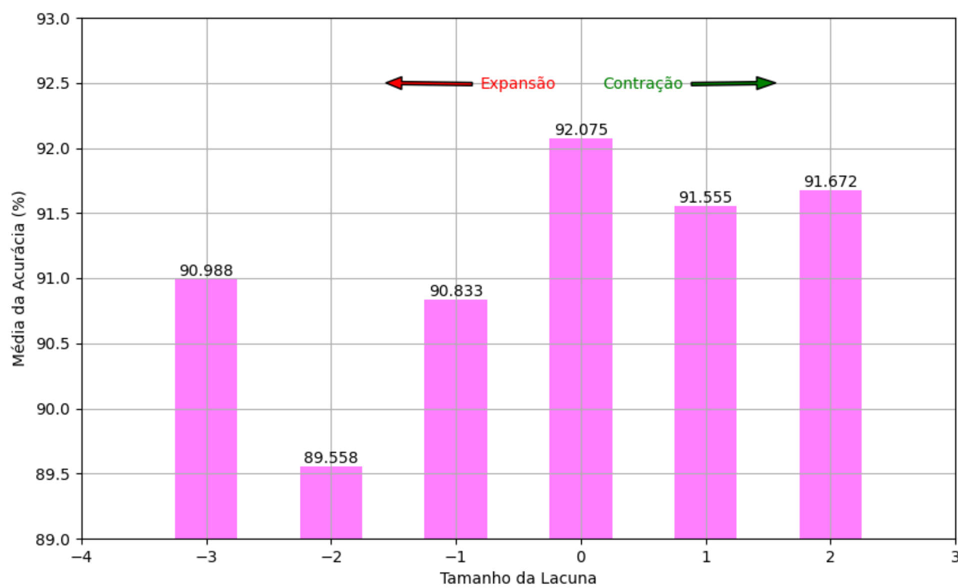


Figura 3.8 Média da acurácia em relação ao tamanho das lacunas na camada de saída das redes neurais treinadas.

Diante da discussão nesta seção, ficou claro que a arquitetura da rede tem uma influência significativa nos resultados dos modelos com bom desempenho. Embora as redes neurais treinadas neste trabalho tenham enfrentado limitações devido a restrições computacionais, foi possível observar tendências relevantes. Apesar de não terem sido utilizadas arquiteturas com grande quantidade de neurônios, parâmetros e camadas, os resultados obtidos indicam direções que podem ser confirmadas por estudos futuros.

Antes de finalizar, perceba que se consideramos o sinal original do valor que define o tamanho de maior disparidade de uma rede, podemos dizer que em uma rede o tipo de disparidade predominante foi a contração ou expansão local. Por exemplo, na rede [4, 9, 5] o tamanho de maior disparidade é 5, mas o tipo de disparidade predominante da rede seria expansão visto que o valor de tamanho da lacuna original -5 é negativo.

Na Figura 3.9, podemos avaliar como a média da acurácia das redes treinadas se comportou em relação aos tipos de disparidade predominantes. Fica claro que, se possível, deve-se construir redes com expansão local na primeira camada e contrair a dimensionalidade na camada de saída.

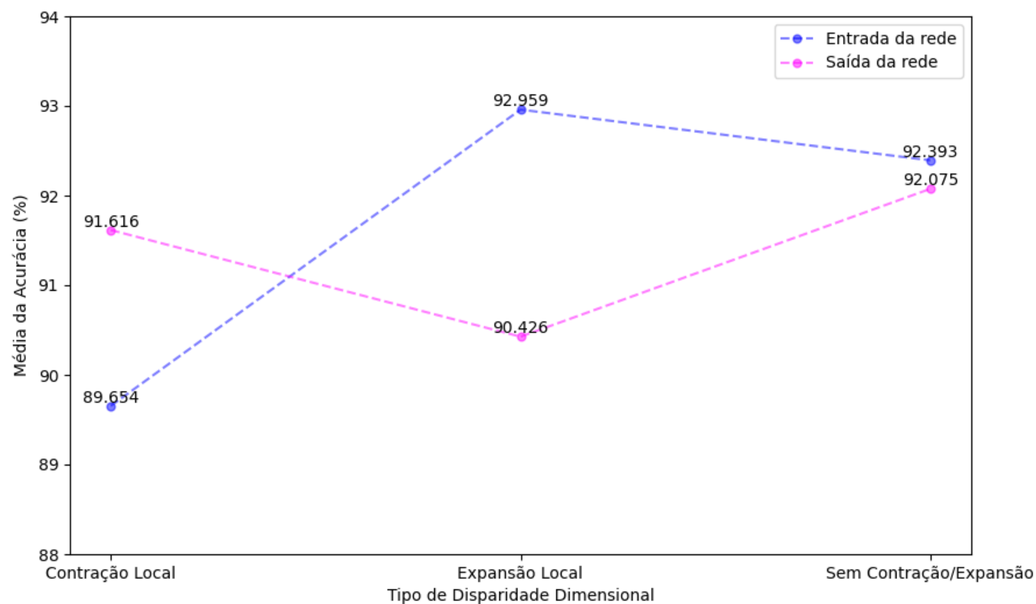


Figura 3.9 Média da acurácia em relação ao tipo de disparidade predominante nas extremidade das redes neurais treinadas.

Na próxima seção, faremos análise dos modelos que tiveram mau desempenho, buscando entender as razões que levaram à falha no treinamento, bem como verificação de possibilidade de ajuste para obter uma acurácia.

3.4 Análise de Modelos com Baixo Desempenho

Analisar a acurácia de modelos com baixo desempenho é crucial para identificar pontos na fase do treinamento passíveis de melhoria. Redes neurais artificiais com baixo desempenho indicam que há falhas na captura dos padrões subjacentes do conjunto de treinamento, podendo ser resultado de insuficiência de dados, qualidade dos dados inadequada, escolha inadequada do modelo, ou hiperparâmetros mal ajustados. A análise detalhada desses aspectos permite identificar a raiz do problema e direcionar os esforços de otimização.

Na seção 3.2, vimos que das 1986 redes treinadas, cerca de 43.35% não obtiveram um bom desempenho, e que a maioria dos seus modelos não havia avançado positivamente em relação a acurácia em nenhum momento. Isso nos leva a constatar que esses modelos tiveram sua fase de treinamento prejudicada, seja por um hiperparâmetro inadequado, pelo tipo de rede, ou qualquer outra eventual situação.

Ainda na Seção 3.2, podemos relembrar na Figura 3.3, a arquitetura e acurácia das redes escolhidas para obter uma análise mais detalhada. Utilizaremos esse mesmo conjunto de arquiteturas para realizar treinamentos ligeiramente diferentes do realizado com os hiperparâmetros fixados na Listagem 3.1.12.

É natural se perguntar se o problema das redes que não tiveram boa performance surgiu por alguma razão ligada à arquitetura da rede, mas não obtivemos fatos que pudessem comprovar isso pelo fato do treinamento ter sido prejudicado, como discutido na seção anterior. Diante disso, iremos explorar algumas possibilidades de ajuste nos hiperparâmetros.

Na Tabela 3.4, é possível visualizar a escolha dos hiperparâmetros dos novos modelos que treinaremos para verificar a possibilidade de corrigir o problema na fase de treinamento. Ademais, manteremos o modelo que obteve um bom desempenho para usarmos como critério de comparação em relação à perfomance das redes para diferentes escolhas de hiperparâmetros ao decorrer desta seção.

Ainda sobre a Tabela 3.4, o primeiro bloco que contém o modelo 1, se refere ao modelo utilizado para treinar as redes analisadas nas seções anteriores, cujos hiperparâmetros foram definidos na Tabela 3.1. No segundo bloco, temos os modelos com hiperparâmetros semelhantes ao modelo 1, exceto pelo fato de que não possui momento. O terceiro, possui variações da taxa de aprendizagem e momento fixo de 50%. Já o terceiro bloco, possui estrutura semelhante ao segundo, mas com tamanho do lote de 1000 observações. Por fim, o último bloco não possui mudanças como essas em relação ao modelo 1, mas possui uma peculiaridade que é a mudança da função de ativação nas camadas ocultas, variando de ReLU, descrita na Definição 2.1.0.3, para tangente hiperbólica.

Tabela 3.4 Dados de Treinamento dos Novos Modelos - Redes com Baixo Desempenho

	Id Modelo	Função Ativação	Taxa Aprendizagem	Momento	Tamanho Lote
i	1	ReLU	0.01	0.9	500
ii	2	ReLU	0.1	-	500
	3	ReLU	0.01	-	500
	4	ReLU	0.001	-	500
	5	ReLU	0.0001	-	500
iii	6	ReLU	0.1	0.5	500
	7	ReLU	0.01	0.5	500
	8	ReLU	0.001	0.5	500
	9	ReLU	0.0001	0.5	500
iv	10	ReLU	0.1	-	1000
	11	ReLU	0.01	-	1000
	12	ReLU	0.001	-	1000
	13	ReLU	0.0001	-	1000
v	14	Tanh	0.1	0.9	500
	15	Tanh	0.01	0.9	500
	16	Tanh	0.001	0.9	500
	17	Tanh	0.0001	0.9	500

A tangente hiperbólica mapeia os valores de entrada para um intervalo entre -1 e 1. Quando as ativações estão centralizadas em torno de zero, os gradientes tendem a ser mais equilibrados, evitando valores extremos que podem levar a explosões ou desaparecimento dos gradientes [2]. Por outro lado, a ReLU é amplamente utilizada e apreciada por sua simplicidade e eficiência computacional. A ReLU ativa os neurônios apenas quando a entrada é positiva, o que ajuda a reduzir a possibilidade de gradientes que desaparecem de forma descontrolada sendo uma escolha robusta e eficiente para muitas arquiteturas de redes neurais, e principalmente para ser aliada quando não há muito recurso computacional disponível.

Visto que atestamos um problema durante a fase de treinamento, a escolha de mudança dos hiperparâmetros para este novo treinamento foi feita visando analisar o comportamento da acurácia mediante variação da taxa de aprendizagem em diferentes cenários. Para isso, utilizaremos as redes descritas na Figura 3.3.

Os modelos identificados com "-", não tiveram nenhuma taxa de momento para o treinamento. Além disso, todos modelos foram treinados com no máximo 5.000 épocas e o critério de parada foi modificado para ser com base na acurácia, finalizando o treinamento

no momento em que atingisse um limiar acima de 90%. Isso se deu pelo fato que na Figura 3.3, vimos que é possível "estragar" a rede ao analisar a evolução da acurácia do modelo [4, 4, 6, 1, 2, 5, 4], que atingiu um determinado limiar de acurácia, mas no fim não teve um desempenho tão considerável. Acreditamos que isso se deu pelo fato do critério de parada ter sido com base na perda, para uma taxa muito pequena e exigente, obrigando o modelo a buscar um mínimo da função de custo que talvez ele não seja capaz de atingir nas condições dadas.

Diferente do treinamento do modelo 1, que tinha critério de parada com base perda para um limiar de 0.1, dos 48 modelos que possuem novo critério de parada com base na acurácia, vinte e quatro atingiram o critério. Analisando as diferentes arquiteturas, dos 16 modelos com [4, 6, 6, 6, 6, 6, 6, 4], onze modelos atingiram o novo critério de parada. Para a arquitetura [4, 4, 6, 1, 2, 5, 4], 10 dos 16 modelos atingiram o novo critério. Por fim, na arquitetura [4, 6, 3, 2, 4, 4], apenas três dos dezesseis modelos conseguiram atingir o novo critério de parada baseado na acurácia.

A arquitetura [4, 6, 3, 2, 4, 4] não apresentou bom desempenho com o ajuste de nenhum hiperparâmetro, exceto pela modificação da função de ativação de ReLU para tangente hiperbólica. Esta mudança específica na função de ativação resultou em uma melhoria significativa no desempenho de todos os modelos que a engloba. Na Figura 3.10, é possível analisar isso em detalhes observando os modelos de 14 à 17. Este resultado mostra que as redes que possuem algum grau elevado de disparidade dimensional nas camadas exigem atenção em seu treinamento.

Os resultados mostram que a rede com melhor desempenho no modelo 1, a saber [4,6,6,6,6,6,6,4], obteve 25% de acurácia nos modelos 2, 9 e 13, demonstrando que uma taxa de acurácia muito baixa exige ajustes muito além dos que os citados no presente trabalho, além de maior tempo de treinamento para acrescer a quantidade de épocas. E uma taxa de aprendizagem mais alta, sem o momento, pode provocar um desempenho muito ruim, fazendo possivelmente o modelo convirja para um mínimo local.

Note que a ausência de momento nos modelos 2 a 5 e 10 a 13 não necessariamente resulta em pior desempenho, mas há instâncias de baixa acurácia, demonstrando que o momento pode ajudar em certas configurações. No entanto, o tamanho do lote pode influenciar na taxa de aprendizagem, uma vez que os modelos na rede [4,6,6,6,6,6,6,4] com lote de tamanho 500 apresentaram menor acurácia para a maior taxa de aprendizagem, enquanto os modelos com lote de tamanho 1000 apresentaram a menor acurácia com taxas de aprendizagem menores.

Ainda sobre a Figura 3.10, vemos que a mudança de hiperparâmetros como taxa de aprendizagem, momento e tamanho do lote, auxiliaram na otimização de alguns modelos e

também em decaimento significativo para alguns casos, como podemos ver nos modelos 2, 10 e 13, por exemplo.

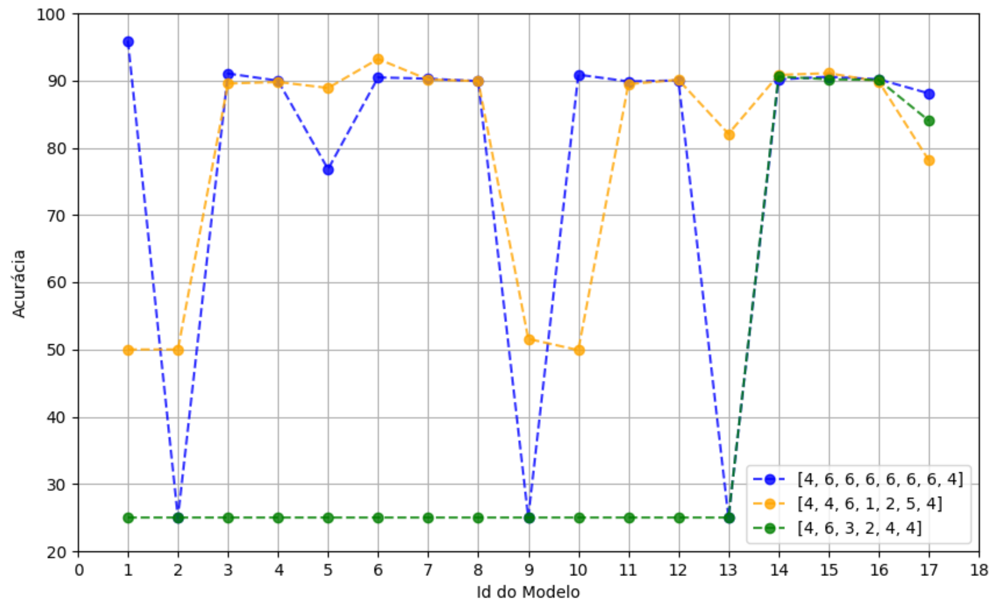


Figura 3.10 Comparação da Acurácia de Modelos com Hiperparâmetros ajustados.

É natural se perguntar como as redes descritas na Figura 3.10 se comportaram em relação à acurácia quando os modelos possuem a mesma taxa de aprendizagem, mas com hiperparâmetros diferentes. Ao considerar

$$\begin{aligned} \mathrm{I} &= [4,6,6,6,6,6,6,4], \\ \mathrm{II} &= [4,4,6,1,2,5,4], \\ \mathrm{III} &= [4,6,3,2,4,4], \end{aligned} \tag{3.10}$$

podemos analisar este fato com mais evidência na Tabela 3.5, levando em consideração a acurácia em todo o conjunto de dados, incluindo os dados de treino e de teste.

Os resultados demonstram que, apesar da taxa de aprendizagem influenciar no resultado, a escolha dos demais hiperparâmetros também pode ter um impacto significativo. Isso é evidenciado de maneira clara pelo bloco dos novos treinamentos da rede III e reforçado pelas discussões anteriores. Assim, uma abordagem experimental para a escolha de hiperparâmetros pode ser necessária em alguns casos visando otimizar o desempenho do modelo e alcançar melhores resultados.

Tabela 3.5 Dados da Acurácia dos Novos Modelos

	Id Modelo	Acurácia
I	1	95.93
	3	91.04
	7	90.27
	11	89.86
	15	90.56
II	1	50.00
	3	89.55
	7	90.14
	11	89.46
	15	91.09
III	1	25.00
	3	25.00
	7	25.00
	11	25.00
	15	90.18

Durante a produção do presente trabalho, não foi possível realizar o treinamento de múltiplas redes com a tangente hiperbólica sendo a função de ativação das camadas ocultas, por questões relacionadas a limitação computacional. Porém, utilizar os resultados que vimos até aqui demonstram ser um bom ponto de partida para estudos e análises futuras que englobem um bom treinamento para expandir análises como as descritas neste trabalho.

Em conclusão, a arquitetura da rede e os hiperparâmetros são componentes essenciais que influenciam significativamente o desempenho de modelos. Um entendimento profundo e uma abordagem meticulosa para ajustar essas configurações podem levar a melhorias substanciais na acurácia e na robustez dos modelos, resultando em soluções mais eficientes e eficazes para os desafios propostos.

Conclusão

Este trabalho explorou a aplicação e otimização de redes neurais, começamos com o processo de otimização de ajustes através do método de gradiente descendente. Definimos as redes neurais e detalhamos a fase de treinamento, utilizando algoritmos específicos para este fim. Testamos múltiplas redes neurais *multilayer perceptron* para resolver um problema de classificação multiclasse, mantendo os hiperparâmetros fixos para realizar uma comparação justa da acurácia obtida ao final do processo.

A análise focou na estrutura da arquitetura das redes, particularmente nas disparidades dimensionais entre as camadas ocultas, refletidas pela quantidade de neurônios em cada camada. Os resultados demonstraram que redes com menor disparidade nas camadas tendem a apresentar maior acurácia. Este achado sugere que a uniformidade na distribuição de neurônios entre as camadas pode facilitar o aprendizado da rede, resultando em previsões mais precisas. No entanto, também ficou evidente que redes neurais podem apresentar problemas durante a fase de treinamento, indicando a necessidade de ajustes contínuos de parâmetros para otimização.

Outro ponto crucial identificado é a relação entre o número de neurônios e a demanda por recursos computacionais. Redes mais complexas, com maior número de neurônios, requerem mais parâmetros a serem ajustados, o que, por sua vez, demandam maior capacidade computacional. Por fim, a importância do tratamento adequado dos dados antes do treinamento foi reafirmada, uma vez que dados bem processados são fundamentais para garantir um treinamento eficiente e resultados mais confiáveis.

Em suma, a escolha da arquitetura ideal e o tratamento dos dados são determinantes para o desempenho das redes neurais. Este trabalho reforça a necessidade de uma abordagem experimental e iterativa na otimização de redes neurais para problemas de classificação, destacando a importância de parâmetros bem ajustados e uma estrutura de rede balanceada, desde a arquitetura da rede à escolha dos hiperparâmetros.

Bibliografia

[1] R. Hildreth. Bio-logical: Java implementation of a neural network. https://medium.com/codex/bio-logical-java-implementation-of-a-neural-network-e9080f8f6b67, 2021. Acessado em: 30 de maio de 2024.

[2] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[3] P. C. T. Gomes. Redes neurais: Desvendando o cérebro artificial, 2024. https://www.datageeks.com.br/redes-neurais/. Acessado em: 8 de abril de 2024.

[4] SAS. Redes neurais - o que são e qual sua importância?, 2023. https://www.sas.com/pt_br/insights/analytics/neural-networks.html. Acessado em: 10 de março de 2024.

[5] Wikipedia contributors. Rede neural artificial. https://pt.wikipedia.org/wiki/Rede_neural_artificial. Acessado em: 01 de junho de 2024.

[6] S. Haykin. *Neural networks and learning machines*. Pearson Prentice Hall, 3rd edition, 2009.

[7] Asimov Institute. The neural network zoo, 2016. https://www.asimovinstitute.org/neural-network-zoo/. Acessado em: 11 de julho de 2024.

[8] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, 1989.

[9] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[10] Amazon Web Services. What is a neural network? https://aws.amazon.com/pt/what-is/neural-network/, 2023. Acessado em: 01 de julho de 2024.

[11] Dendrites. O que é classificação multiclasse? https://dendrites.io/glossario/o-que-e-classificacao-multiclasse/, 2023. Acessado em: 05 de maio de 2024.

[12] P. J. Olver. Lecture notes on numerical analysis. https://www-users.cse.umn.edu/~olver/, 2008.

[13] Y. D. Sobral. Notas de aula de cálculo numérico. Universidade de Brasília, 2020.

[14] F. Coelho and M. L. Lourenço. *Um Curso de álgebra Linear*. Edusp, São Paulo, 2018.

[15] M. A. G. Ruggiero. Ajuste de curvas. https://www.ime.unicamp.br/~marcia/AlgebraLinear/ajuste_curvas.html, 2023. UNICAMP. Acessado em: 01 de abril de 2024.

[16] P. Miranda. Princípios de desenvolvimento de algoritmos. http://www.vision.ime.usp.br/~pmiranda/mac122_2s18/page/recursao.pdf. Acessado em: 14 de junho de 2024.

[17] R. L. Burden, J. D. Faires, and A. M. Burden. *Numerical Analysis*. PWS Publisher company, 5th edition, 1993.

[18] C. Lemaréchal. Cauchy and the gradient method. *Doc Math Extra*, pages 251–254, 2012. https://www.mathunion.org/fileadmin/IMU/Prizes/Abel/2012/docmath-extra-251-254.pdf. Acessado em: 25 de maio de 2024.

[19] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. https://web.stanford.edu/~boyd/cvxbook/.

[20] I. N. da Silva, D. H. Spatti, and R. A. Flauzino. *Redes neurais artificiais: para engenharia e ciências aplicadas*. Artliber, São Paulo, 2010.

[21] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986. https://www.nature.com/articles/323533a0.

[22] A. Longarini. Redes neurais artificiais | as máquinas pensam?, 2019. https://www.lambda3.com.br/2019/09/redes-neurais-artificiais-as-maquinas-pensam/. Acessado em: 02 de abril de 2024.

[23] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf. Acessado em: 01 de julho de 2024.

[24] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. https://www.bioinf.jku.at/publications/older/2604.pdf. Acessado em: 01 de julho de 2024.

[25] U. Triacca. *Maximum Likelihood Theory*. Department of Computer Engineering, Computer Science and Mathematics, University of L'Aquila, L'Aquila, Italy. https://edisciplinas.usp.br/pluginfile.php/5525752/mod_resource/content/1/triacca.pdf. Acessando em: 20 de abril de 2024.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[27] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 249–256, 2010. https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf. Acessado em 20 de agosto de 2024.

[28] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034, 2015. https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/He_Delving_Deep_into_ICCV_2015_paper.pdf. Acessado em 20 de agosto de 2024.

[29] A. Ustad, A. Logacjov, S. Ø. Trollebø, P. Thingstad, B. Vereijken, and K. Bachand N. S. Maroni. Validation of an activity type recognition model classifying daily physical behavior in older adults: The har70+ model. *Sensors*, 23(5):2368, 2023. https://doi.org/10.3390/s23052368.

[30] W. McKinney. Pandas. https://pandas.pydata.org. Acessado em: 02 de julho de 2024.

[31] PyTorch Foundation. Pytorch foundation. https://pytorch.org/foundation. Acessado em: 02 de julho de 2024.

[32] META AI. Pytorch. https://pytorch.org/docs/stable/torch.html. Acessado em: 02 de julho de 2024.