Universidade Federal de São Carlos
Centro de Ciências Exatas e de Tecnologia
Departamento de Estatística

Redes Neurais Aplicadas a Grafos: Uma Abordagem Semi-Supervisionada

Samuel Treméa

Trabalho de Conclusão de Curso

Universidade Federal de São Carlos
Centro de Ciências Exatas e de Tecnologia
Departamento de Estatística

Redes Neurais Aplicadas a Grafos: Uma Abordagem Semi-Supervisionada

Samuel Treméa
Orientador(a): Andressa Cerqueira

Trabalho de Conclusão de Curso apresentado ao Departamento de Estatística da Universidade Federal de São Carlos - DEs-UFSCar, como parte dos requisitos para obtenção do título de Bacharel em Estatística.

São Carlos
Abril de 2022

Federal University of São Carlos
Exact and Technology Sciences Center
Department of Statistics

Neural Networks Applied to Graphs: a Semi-Supervised Approach

Samuel Treméa
Advisor: Andressa Cerqueira

Bachelors dissertation submitted to the Department of Statistics, Federal University of São Carlos - DEs-UFSCar, in partial fulfillment of the requirements for the degree of Bachelor in Statistics.

São Carlos
April 2022

Samuel Treméa

Redes Neurais Aplicadas a Grafos: Uma Abordagem Semi-Supervisionada

Este exemplar corresponde à redação final do trabalho de conclusão de curso devidamente corrigido e defendido por Samuel Treméa e aprovado pela banca examinadora.

Aprovado em 18 de abril de 2022

Banca Examinadora:

- Andressa Cerqueira (Orientadora)
- Luis Ernesto Bueno Salazar
- Vera Lucia Damasceno Tomazella

Dedico este trabalho à força e ao espírito interior que me rege e sempre me lembrava que, além da tempestade, sempre há um arco-íris.

Dedico este trabalho aos meus pais, Bernadete Fátima Von Gilsa e Emílio Antônio Treméa, que sempre me incentivaram a estudar, nunca duvidaram da minha capacidade e não mediram esforços para que eu pudesse chegar até aqui.

Por fim, também dedico este trabalho ao Alan Gerardi, à dona Noely e ao seu Haroldo, que possibilitaram que eu pudesse começar a estudar em São Carlos. Sem vocês, minha vida teria sido completamente diferente.

Agradecimentos

Agradeço à Prof. Andressa Cerqueira, por ter aceitado me orientar, pelos ensinamentos, inúmeras reuniões, pela paciência e por ter me transmitido a segurança necessária para que este Trabalho pudesse ser concretizado. Palavras não expressam o quão prazeroso foi realizar este trabalho ao seu lado e que, cada vez mais, surjam mais profissionais como ela.

Agradeço aos grandes amigos que eu fiz na Universidade Federal de São Carlos e na Universidade de São Paulo, pelas inúmeras risadas, por me acolherem em São Carlos no momento que eu mais me senti sozinho e por acreditarem em mim mesmo nos momentos em que eu mesmo não acreditava.

Agradeço ao meu amigo Guilherme Perego que, em um momento de adversidade, aceitou de prontidão o meu pedido de ajuda e me auxiliou a identificar um problema fundamental no código. Sua amizade é um dos presentes mais valiosos que São Carlos me trouxe e tenho muito orgulho de poder dizer que sou seu amigo.

"Sessenta anos atrás, eu sabia tudo. Hoje sei que nada sei. A educação é a descoberta progressiva da nossa ignorância."

(Will Durant)

Resumo

Neste Trabalho de Conclusão de Curso, propõe-se a análise aprofundada das Redes Convolucionais de Grafos, um método de aprendizagem de máquina semi-supervisionada para classificação de nós para dados com estruturas de redes. Para tal, tendo como base o artigo dos seus idealizadores, Thomas Kipf e Max Welling (Kipf e Welling, 2017), serão estudadas suas características, nuances e particularidades e, por fim, ele será colocado em prática, com o auxílio da linguagem *Python*, em dados reais.

Palavras-chave: *aprendizado de máquina, grafos, redes neurais, classificação.*

Abstract

In this work, we propose an in-depth analysis of Graph Convolutional Networks, a semi-supervised machine learning method for node classification in graph-structured data. Based on the seminal work (Kipf e Welling, 2017), proposed by Thomas Kipf and Max Welling, the objective of this work is to evaluate in depth the characteristics, nuances, and particularities of this method. This method is also applied to real data in *Python*.

Keywords: *machine learning, graphs, neural network, classification.*

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Imagine que duas pessoas estão jogando uma partida de damas, uma espécie de xadrez simplificado. É natural pensar que os jogadores se perguntem se é possível, de alguma maneira, quantificar suas respectivas probabilidades de vitória naquele dado turno. Segundo McCarthy (2021), em 1949, o cientista Arthur Samuel teve este mesmo questionamento e, então, desenvolveu um programa que atribuía uma pontuação a cada jogador a partir de uma análise realizada nas posições das peças localizadas no tabuleiro e, depois disso, selecionava o movimento que maximizaria as chances de tal jogador vencer. Ao perceber que seu programa era promissor, Arthur passou a buscar formas de aperfeiçoá-lo. Três anos depois, o cientista aprimorou seu programa, dando-lhe a capacidade de lembrar as posições das peças de partidas passadas e agir de maneira adequada, levando os resultados obtidos previamente em consideração. Tal método foi cunhado como *Rote Learning* e, em 1952, foi falado, pela primeira vez na história, em "aprendizado de máquina".

A história do aprendizado de máquina e sua evolução se confunde com a história da própria inteligência artificial. Na sua busca por fazer com que os computadores consigam, de certa forma, aprender com seus próprios erros e chegar a conclusões, cientistas do mundo inteiro criaram um campo específico da Engenharia e Ciência da Computação (que hoje abrange a própria Estatística e tantas outras áreas) que busca, por meio de modelos matemáticos e métodos computacionais de reconhecimento de padrões, realizar uma vasta gama de aplicações, tais como previsões de bancos de dados, tomada de decisões e classificações de objetos.

Contudo, justamente por se tratar de um campo tão abrangente, é de se esperar que um só tipo de algoritmo não resolva todas as questões que o aprendizado de máquina se propõe a resolver. Deste modo, é possível diferenciar seus algoritmos em três tipos de aprendizado:

supervisionado, **semi-supervisionado** e **não-supervisionado**. A principal diferença entre os tipos se trata na quantidade de dados previamente classificados de forma correta, de modo que o algoritmo possa usar essa informação como "aprendizado" e replicá-la para novos dados.

De acordo com Izbicki e dos Santos (2020), em uma aprendizagem **supervisionada**, o banco de dados inteiro está classificado corretamente com a variável de interesse. Por exemplo, em um banco de dados sobre cachorros em que é especificado qual é a raça de cada um e deseja-se identificar a raça de um animal que não está na base original. Já para uma aprendizagem **semi-supervisionada**, apenas uma parte do banco de dados está classificada, de modo que o algoritmo conseguiria "aprender" com esta parcela do banco, mas teria que utilizar outros métodos para trabalhar com o restante da base. Por fim, a aprendizagem **não-supervisionada** é utilizada quando não há informações sobre a característica de interesse no banco de dados. Assim, resta ao algoritmo buscar algum tipo de associação ou agrupamento entre as observações. Dentre o oceano de aplicações possíveis para tais métodos, estão as **redes neurais**.

Curiosamente, a origem das redes neurais se dá antes mesmo do surgimento do próprio aprendizado de máquina. Em McCulloch (1943), seis anos antes da partida de damas de Arthur Samuel, o matemático Walter Pitts e o neurofisiologista Warren McCulloch trabalharam juntos para tentar descobrir como funcionava os neurônios do cérebro humano. Utilizando diversos algoritmos e modelos matemáticos, eles conseguiram replicar seu funcionamento em circuitos elétricos por meio de um método denominado **lógica de limiar** (*Threshold Logic*), que consiste em transformar um dado de entrada contínuo em um dado de saída discreto. É por isso que "redes neurais" possui este nome: pois seus algoritmos buscam replicar o comportamento dos neurônios no cérebro humano.

De modo geral, os neurônios de uma rede neural são organizados em três camadas: camada de **entrada**, camadas **intermediárias** e camada de **saída**. As camadas de **entrada** são responsáveis por receber os dados a serem analisados de fontes externas. Estes dados podem variar desde vetores numéricos até pixels de uma imagem ou até mesmo arquivos de áudio. Depois da camada de entrada, existem as camada **intermediárias**, também conhecidas como camadas escondidas. Estes neurônios recebem os dados da camada de entrada e realizam conexões e análises de acordo com os parâmetros da rede. Por fim, os neurônios da camada de **saída** são responsáveis por receber as informações que foram trabalhadas nas camadas escondidas e exportar de acordo com a finalidade do

algoritmo. A Figura 1.1 fornece uma representação visual do processo descrito acima.

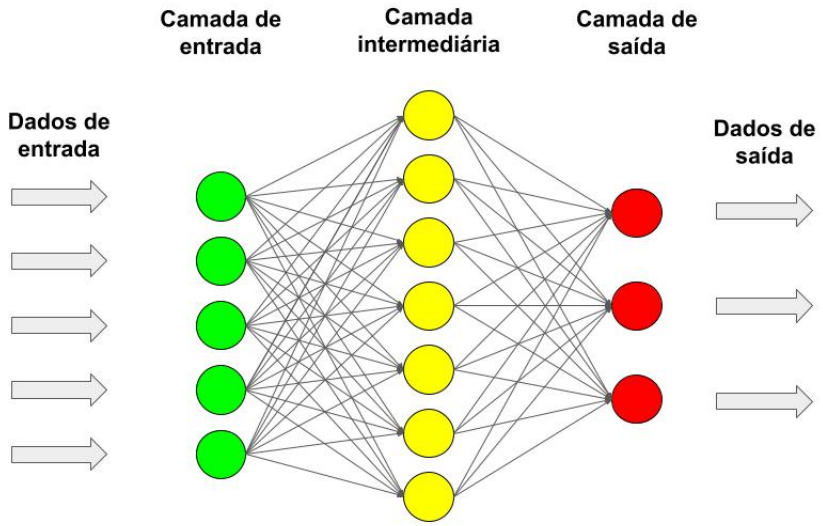

Figura 1.1: Exemplo de Rede Neural

Uma análise mais aprofundada sobre os alicerces matemáticos que suportam este campo do aprendizado de máquina será realizada ao decorrer deste trabalho. Contudo, por se tratar de um tema tão amplo e com possibilidades tão diversas, torna-se difícil navegar por este assunto sem um foco prévio estabelecido de maneira clara. Então, decidiu-se que o presente trabalho irá explorar métodos e utilidades das já mencionadas redes neurais em dados estruturados em **grafos**.

Mas, o que são grafos?

Um grafo é uma estrutura composta por dois componentes, denominados **nós** e **arestas**. Geralmente, os nós representam objetos individuais e as arestas se tratam das conexões entre estes objetos. Por exemplo, imagine uma sala de aula repleta de alunos. Neste caso, cada nó representaria cada aluno desta classe e, se dois alunos quaisquer são amigos (ou seja, possuem uma conexão entre si), haverá uma aresta conectando tais nós. A Figura 1.2 fornece outro exemplo de grafos, mas de maneira visual.

Atualmente, o volume de dados estruturados deste modo é de uma abundância indescritível. Sua estrutura é compacta e informativa, mas pode ser complexa de ser manipulada e analisada por métodos mais tradicionais de predição e aprendizado de máquina, fazendo-se necessária a reinvenção de um novo leque de possibilidades e ideias que possibilite suprir tal demanda. Uma dessas possibilidades se refere às **Redes Convolucionais de Grafos** (do inglês, *Graph Convolutional Networks*), que busca aliar os dois conceitos apresentados aqui (Redes neurais e grafos), e é o foco principal deste trabalho.

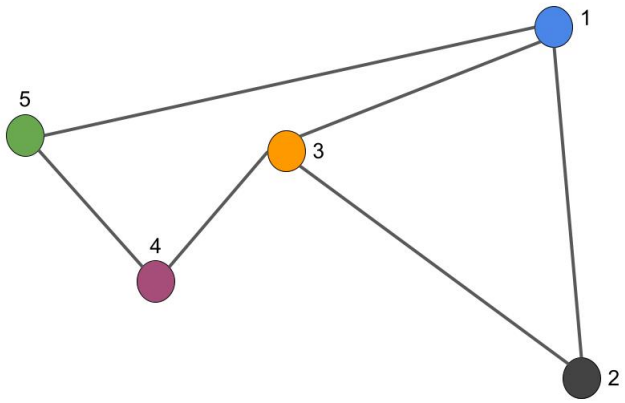

Figura 1.2: Exemplo de Grafo

1.1 Objetivos

Este projeto é alicerçado em dois objetivos principais:

- Aprofundar-nos no método da Redes Convolucionais de Grafos, um procedimento recentemente proposto na literatura para classificação de nós em grafos que alia conceitos de redes convolucionais e aprendizagem semi-supervisionada para utilizar redes neurais e aprendizagem de máquina em um novo tipo de problema.

- Por se tratar de uma metodologia consideravelmente recente, revelou-se uma árdua tarefa encontrar materiais relacionados a ela em português, nossa linguagem materna. Assim, também estabelece-se como um dos principais objetivos deste Trabalho de Conclusão de Curso, a acessibilidade que lhe é devida a um tema tão interessante e atraente para alunos e entusiastas do ramo da Estatística e das Ciências Exatas de modo geral.

1.2 Organização

No Capítulo 2, serão abordados os principais aspectos a respeito da estrutura de grafos, como intuições e aplicabilidades, bem como a base matemática que sustenta seus conceitos e exemplos práticos. No Capítulo 3, será apresentada uma introdução a respeito das Redes Convolucionais de Grafos, explicando a intuição por trás da técnica, bem como suas devidas representações matemáticas. Prosseguindo, no Capítulo 4, serão descritas as análises realizadas com o banco de dados escolhido, juntamente com as informações dos pacotes e procedimentos em *Python* e, por fim, no Capítulo 5, serão realizadas as

considerações finais do projeto.

Capítulo 2

Grafos

A partir deste ponto, serão apresentados capítulos específicos para introduzir os conceitos fundamentais que servirão de alicerce para o restante deste Trabalho de Graduação. Como os dados com os quais trabalharemos estarão organizados em estruturas de **grafos**, é de suma importância que se estabeleça elementos importantes que ajudarão o caro leitor a se familiarizar, caso seja leigo no assunto, com as idéias que serão introduzidas mais adiante.

2.1 Intuição

O objetivo desta seção é fornecer uma visão mais intuitiva de como dados podem ser estruturados em grafos para que o leitor passe a ter mais familiaridade com o conceito. Como já especificado na introdução, um grafo é uma estrutura composta por dois componentes, denominados **nós** e **arestas**. Geralmente, os nós representam objetos individuais e as arestas se tratam das conexões entre estes objetos.

Exemplo 1: Imagine que um pesquisador tenha interesse em analisar uma determinada rede social para detectar qual é a proporção de *fake news* que são compartilhadas por seus usuários. Nesta situação, pode-se estabelecer uma estrutura de grafo, cujos nós serão os usuários desta rede social e as arestas seriam as notícias (falsas ou não) entre os usuários.

Exemplo 2: Dados relacionados a viagens aéreas ou rodoviárias também podem ser interpretados como estruturas de grafos de uma maneira muito intuitiva. Basta considerar que cada nó do grafo é representado por uma cidade, e que as arestas simbolizam a existência de conexões diretas de meios de transporte entre as duas cidades representadas

pelos nós.

2.2 Definições Matemáticas

Agora, serão fornecidas definições mais formais para conceitos que serão repetidos à exaustão no decorrer de todo este Trabalho de Graduação.

2.2.1 Grafos e Matriz de Adjacência

O primeiro passo que precisamos tomar para alcançar o nosso objetivo é definir, de fato, o que são grafos.

Definição 2.1 (Nós e Arestas)

Seja $\mathcal{E}$ um conjunto de arestas e seja $\mathcal{V}$ um conjunto de nós em uma base de dados qualquer. Então, representa-se uma aresta que conecta um nó $u \in \mathcal{V}$ e $v \in \mathcal{V}$ como:

$$(u, v) \in \mathcal{E}$$

Definição 2.2 (Grafos)

Seja $\mathcal{E}$ um conjunto de arestas e seja $\mathcal{V}$ um conjunto de nós em uma base de dados qualquer. Então, um grafo é definido por:

$$\mathcal{G} = (\mathcal{V}, \mathcal{E})$$

Por mais que seja simples, elencar todos os pares possíveis de um grafo da maneira definida acima pode ser uma tarefa árdua e tediosa, principalmente se estivermos falando de dezenas de milhares de nós, por exemplo. Então, surge uma ferramenta de grande importância que possibilita visualizar todas as interações entre os nós e as arestas de um grafo, denominada **matriz de adjacência**.

Definição 2.3 (Matriz de Adjacência)

Seja $\mathcal{G}$ um grafo com conjunto de arestas $\mathcal{E}$ e conjunto de nós $\mathcal{V}$ quaisquer. Então, a matriz de adjacência do grafo $\mathcal{G}$ é uma matriz $A^{|\mathcal{V}|\times|\mathcal{V}|}$, em que, dados dois nós $u \in \mathcal{V}$ e $v \in \mathcal{V}$:

$$A[u, v] = 1; \quad (u, v) \in \mathcal{E} \tag{2.4}$$

e

$$A[u, v] = 0; \quad (u, v) \notin \mathcal{E} \tag{2.5}$$

Na prática, o que a definição acima diz é que cada linha e cada coluna da matriz de adjacência representará um nó do grafo e, caso haja uma aresta conectando dois nós, o valor correspondente na matriz será 1. Caso não haja arestas, o valor será 0.

Exemplo 3: Seja $\mathcal{G}$ um grafo identificado pela Figura 2.1.

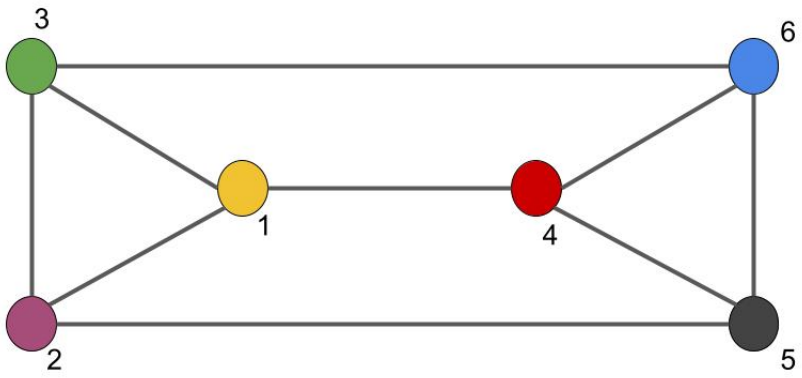

Figura 2.1: Grafo $\mathcal{G}$ com 6 nós e 9 arestas.

Assim, a matriz de adjacência $\mathbf{A}$ do grafo $\mathcal{G}$ é dada por:

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Perceba como identificar as conexões entre os nós se torna uma tarefa mais intuitiva, visual e menos árdua trabalhando com uma matriz ao invés de analisar todas as combinações de nós posíveis um-a-um, sob qual a definição é alicerçada. É claro, o exemplo acima trata de um grafo de dimensões infinitamente menores do que os que normalmente

são tratados mundo afora, mas o seu intuito é mostrar como as visualizações e, posteriormente, os cálculos podem ser facilitados ao trabalharmos, principalmente de maneira computacional, com uma matriz.

2.2.2 Grau de um nó

Ao analisarmos um grafo, um dos primeiros e mais importantes aspectos que podem chamar a nossa curiosidade seria buscar identificar quais seriam os nós mais *importantes* dentro daquele ecossistema, ou seja, quais são os nós que, de uma forma ou de outra, exercem mais influência sobre os demais e são mais relevantes dentro do grafo em questão. À primeira vista, faz sentido pensar que a importância de um nó é diretamente proporcional ao número de conexões que ele possui, ou seja, um nó que possui maior número de conexões possui maior importância do que outro com menos conexões. Assim, nasce o conceito de **grau** de um nó.

Definição 2.6 (Grau de um Nó)

Seja $\mathcal{G}$ um grafo com conjunto de arestas $\mathcal{E}$ e conjunto de nós $\mathcal{V}$ quaisquer. Então, o grau g_u de um nó u é dado por:

$$g_u = \sum_{v \in \mathcal{V}} A[u, v] \tag{2.7}$$

Ou seja, o grau de um nó nada mais é do que a quantidade de arestas ligadas a ele. Analogamente, podemos definir $\mathcal{N}_u$ como a **vizinhança** de u, ou seja, o conjunto de nós que fazem conexão direta com o nó u. Identificar quais são os nós de maior grau em um grafo pode te ajudar a perceber as nuances principais da estrutura daquele grafo e quais são os principais pontos de influência dentro daquele ambiente. Perceba o que acontece no grafo apresentado na Figura 2.1. Ali, todos os nós possuem 3 arestas de ligação, ou seja, todos os nós possuem grau 3. Uma interpretação que podemos ter é que o grafo em questão é consideravelmente homogêneo e estável, sem nenhum ponto particular de influência e também sem pontos aparentes que não possuam importância própria.

2.2.3 Características de um Nó

Em todo estudo realizado em qualquer banco de dados que seja, é de fundamental importância que sejam analisadas, com esmero, os atributos e as características que cada

observação apresenta ao pesquisador. Em outros contextos, estas informações são comumente chamadas de *variáveis* mas, aqui, elas serão tratadas como as *características* de cada uma das nossas observações, ou seja, de cada um dos nossos nós.

No caso do Exemplo 1, uma possível característica de interesse poderia ser a quantidade de amigos ou seguidores que os usuários possuem naquela rede social, o que traria informações mais concretas sobre quão longe uma *fake news* poderia percorrer. Já para o Exemplo 2, pode ser interessante saber o número de passageiros que passam por aquela cidade sob uma determinada frequência.

Tais informações podem ser representadas e visualizadas das mais variadas formas, porém, a exemplo do que acontece com as relações de conexão entre os nós, para o propósito do atual estudo, essas características facilitarão nosso trabalho se também forem escritas no formato matricial.

Definição 2.8 (Características de um Nó)

Seja $\mathcal{G}$ um grafo com conjunto de arestas $\mathcal{E}$ e conjunto de nós $\mathcal{V}$ quaisquer. Também, seja d o número de características presentes em cada nó. Então, a ***matriz de características*** $\mathbf{X} \in \mathbb{R}^{d \times |\mathcal{V}|}$ *é dada por:*

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1|\mathcal{V}|} \\ x_{21} & x_{22} & \cdots & x_{2|\mathcal{V}|} \\ \vdots & \vdots & \ddots & \vdots \\ x_{d1} & x_{d2} & \cdots & x_{d|\mathcal{V}|} \end{bmatrix} \tag{2.9}$$

Interpretando a definição acima, temos que cada linha da matriz representa cada um das características de cada nó e cada coluna se refere a cada um dos nós presentes no grafo.

Exemplo 4: Seja $\mathbf{X}$ a matriz de características identificada abaixo:

$$\mathbf{X} = \begin{bmatrix} 7 & 15 & 2 \\ 1 & 33 & 12 \\ 20 & 9 & 4 \end{bmatrix} \tag{2.10}$$

Perceba que apenas com a matriz de características, já podemos extrair informações relevantes a respeito de cada nó, pois já podemos identificar que $\mathbf{u}_1 = (7, 1, 20)$, $\mathbf{u}_2 = (15, 33, 9)$ e $\mathbf{u}_3 = (2, 12, 4)$.

Capítulo 3

Redes Convolucionais de Grafos

Agora, com os principais conceitos relacionados a grafos devidamente apresentados, pode-se dizer que estamos aptos a começar a aprender como redes convolucionais funcionariam em grafos. A razão pelo qual é necessária a aplicação de um modelo como este ao invés de outros mais conhecidos no ramo de aprendizagem profunda, tais como Redes Neurais Convolucionais (RNC) e Redes Neurais Recorrentes (RNR), se deve pelas particularidades de cada um que não permitem que os mesmos sejam utilizados em grafos. Assim, surge a necessidade da criação de um novo modelo que consiga reproduzir, de maneira satisfatória, os resultados que as ferramentas já consagradas apresentam em suas respectivas áreas de atuação em grafos, sendo fiel à sua estrutura e mantendo a representatividade de seus nós e suas características. Com este objetivo em mente, para absorver e conseguir reproduzir os conceitos e modelos que serão introduzidos ao decorrer deste capítulo, um material em particular nos chamou a atenção em virtude da sua excelente escrita e de sua fantástica didática. O material referido é o Hamilton (2020), que foi utilizado à exaustão e acaba por permear-se ao longo de todas as definições e conceitos que serão apresentados daqui em diante.

3.1 Entrada de dados

Então, pensando em criar um modelo de aprendizagem profunda aplicável para o nosso trabalho, o ideal seria pensar em um modelo cujo método de entrada de dados respeite as regras estruturais de um grafo e também leve em consideração as características associadas a cada um dos nós presentes nele. Logo, faz sentido pensar em um modelo que receba, como entrada, tanto a matriz de adjacência $\mathbf{A}$ quanto a matriz de características $\mathbf{X}$

associadas a um grafo $\mathcal{G}$.

No entanto, um obstáculo aparece ao tentarmos avançar por esta rota. Como sabemos, uma matriz de adjacência $\mathbf{A}$ é representada por seus nós em suas linhas e colunas. Isso significa que, devido à sua estrutura e às propriedades da matriz, um mesmo grafo pode ser representado por diferentes matrizes de adjacência. Em outras palavras, o modelo seria afetado diretamente pela ordem em que os nós estão dispostos na matriz de adjacência, o que não pode acontecer.

Matematicamente, segundo Gilmer et al. (2017), qualquer função f que, tem como entrada, uma matriz de adjacência $\mathbf{A}$ associada a um grafo $\mathcal{G}$, deve satisfazer uma das seguintes propriedades:

$$f(\mathbf{PAP}^\intercal) = f(\mathbf{A}), \tag{3.1}$$

$$f(\mathbf{PAP}^\intercal) = Pf(\mathbf{A}). \tag{3.2}$$

A equação (3.1) está relacionada à propriedade de *Invariância por Permutação*, que aborda que a função f não depende da ordem em que os nós estão dispostos na matriz de adjacência de entrada. Já a equação (3.2) se refere à propriedade de *Equivariância por Permutação* que, mesmo tendo um nome aparentemente complexo, ela apenas assegura que a saída de f é permutada de maneira conjunta com a matriz de adjacência de entrada. Apesar de serem semelhantes, os dois resultados apresentados em (3.1) e (3.2) representam vital importância ao estudarmos modelos baseados em grafos, de modo que a não-completude de um deles em qualquer modelo inviabiliza, por completo, a utilização do mesmo.

3.2 Vetor imerso e passagem de informações

3.2.1 Conceito e Intuição

Apenas por agora, para fins didáticos, suponha que já exista um modelo que satisfaça uma das duas propriedades descritas em (3.1) e (3.2). Como definido anteriormente, temos duas fontes de entrada de dados neste modelo: a matriz de adjacência $\mathbf{A}$, que traz informações a respeito das conexões que os nós possuem uns com os outros, e a matriz de características $\mathbf{X}$, que contém informações das características intrínsecas e individuais de

cada nó relacionadas ao estudo em questão. Daí, surge a necessidade de **combinar** estas informações provenientes de duas fontes distintas em uma estrutura só, de maneira única para cada nó. A esta estrutura, é dado o nome de **vetor imerso**, representado por $\mathbf{h}$, de modo que o vetor imerso relacionado ao nó u é representado por $\mathbf{h}_u$. Aqui, é de extrema importância que fique claro que o vetor imerso $\mathbf{h}_u$ possui dimensão d, sendo d o número de características presentes na matriz de características $\mathbf{X}$.

A troca (ou melhor dizendo, a *atualização*) das informações acontece única e exclusivamente nestes vetores imersos. Mas, como esta troca é realizada? Ora, por meio de iterações.

Quando as duas matrizes são imputadas no modelo, o algoritmo de passagem de informações ainda não começou a funcionar, logo, pode-se dizer que estão na iteração zero. Na iteração zero, o vetor imerso $\mathbf{h}_u$ do nó u nada mais é do que o vetor de características do mesmo nó u. Matematicamente, temos que:

$$\mathbf{h}_u^{(0)} = \mathbf{x}_u, \quad \forall u \in \mathcal{V}. \tag{3.3}$$

Perceba como é mais claro visualizar a dimensão do vetor imerso. Como na iteração zero, o vetor imerso é basicamente o vetor de características, a dimensão do vetor nada mais é do que a quantidade de características.

Agora, imagine que queiramos saber informações a respeito do nó u para que possamos classificá-lo. Para atingirmos este objetivo, o modelo precisa agregar as informações dos nós vizinhos de u; porém, para coletar as informações dos vizinhos de u, é necessário agregar as informações dos vizinhos dos vizinhos de u, e assim o modelo prossegue com uma espécie de recursividade.

Esse processo é repetido quantas vezes forem necessárias. Supondo que ele se repita k vezes, é definido o vetor imerso $\mathbf{h}_u^{(k)}$, que é atualizado com as informações de seus vizinhos.

A meta final desta estrutura de modelo é, depois de K iterações deste método de envio e recepção de mensagens, criar um vetor imerso final $\mathbf{z}_u^{(K)}$ para cada nó $u \in \mathcal{V}$ contendo as devidas informações individuais de cada nó. Convém imaginar que os vetores imersos $\mathbf{h}_u^{(k)}$ são vetores imersos **preliminares** que são atualizados conforme as iterações acontecem, e os vetores $\mathbf{z}_u^{(K)}$ são os vetores imersos finais depois que todas as iterações já ocorreram. Perceba como, nesta estrutura, existem duas atividades distintas que são executadas:

- **Coleta**: Existe a coleta de informações dos vizinhos do nó de interesse u;

- **Atualização**: Depois que as informações são coletadas na iteração k, é necessário substituir o vetor de informações da iteração anterior $\mathbf{h}_u^{(k-1)}$ pelo vetor $\mathbf{h}_u^{(k)}$, correspondente à iteração nova.

Ambas as atividades podem ser interpretadas como duas funções distintas. A função responsável pela **coleta** das informações receberá, como entrada, os vetores de informações $\mathbf{h}_v^{(k)}$, $v \in \mathcal{N}(u)$, onde $\mathcal{N}(u)$ é a vizinhança do nó u, e terá como *output*, a mensagem compilada por estas informações. Em outras palavras, apenas os nós da vizinhança do nó u vão entrar nesta função. Esta mensagem servirá como *input* na função relacionada à **atualização** do vetor de informações que, utilizando o vetor anterior $\mathbf{h}_u^{(k-1)}$ como segundo *input*, fará a atualização e terá como *output*, o vetor $\mathbf{h}_u^{(k)}$. Para uma melhor visualização do processo descrito acima, dispõe-se da figura descrita em 3.1.

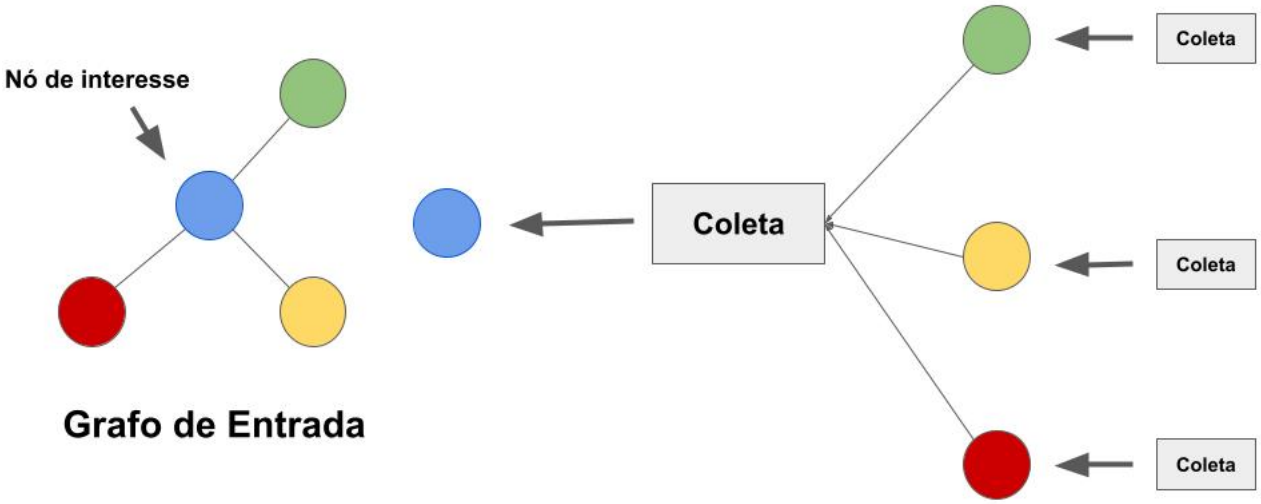

Figura 3.1: Modelo de Transmissão de Mensagem.

3.2.2 Representação Matemática

Defina-se f como a função relacionada à atividade de coleta de informações e g a função relacionada à atualização das informações. Assim, é possível representar, matematicamente, todo o processo descrito na Seção 3.2.1 da seguinte forma:

$$\mathbf{h}_u^{(k+1)} = \underbrace{g^{(k)}\left(\mathbf{h}_u^{(k)}, \underbrace{f^{(k)}(\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u))}_{\text{Coleta da Informação}}\right)}_{\text{Atualização da Informação}}. \tag{3.4}$$

Perceba como o segundo argumento da função g se refere aos vetores imersos (ou a mensagem) que são coletados dos vizinhos v do nó de interesse u. Para simplificar a expressão, é possível definir a mensagem $\mathbf{m}_{\mathcal{N}(u)}^{(k)}$ como sendo:

$$\mathbf{m}_{\mathcal{N}(u)}^{(k)} = f^{(k)}(\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)), \tag{3.5}$$

de modo que:

$$\mathbf{h}_u^{(k+1)} = g^{(k)}(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}). \tag{3.6}$$

Assim, ao final de K iterações, é possível definir o vetor imerso final do modelo como:

$$\mathbf{z}_u = \mathbf{h}_u^{(K)}, \forall u \in V. \tag{3.7}$$

3.3 Passagem de Informações em Grafos

Na seção anterior, foram definidas as funções de coleta e atualização de informações de maneira mais genérica e abstrata. Agora, é necessário pensar em alternativas de transferir tais idéias em funções de modo que sejam possíveis e, mais do que isso, viáveis de serem trabalhadas em dados estruturados em grafos.

Por se tratar de um método de aprendizagem de máquina, é necessário que, no modelo, sejam inseridos **parâmetros** que possam ser treinados pelo algoritmo. Para este caso, será utilizada uma matriz de parâmetros $\mathbf{W}$ que, inicialmente, será dividida em dois termos distintos: uma matriz $\mathbf{W}_a$ referente apenas ao nó de interesse e uma matriz $\mathbf{W}_b$ relacionada aos vizinhos do nó de interesse.

Baseando-nos no modelo descrito em Hamilton (2020), a coleta da mensagem $\mathbf{m}_{N(u)}^{(k)}$ será agregada a partir da soma dos vetores imersos $\mathbf{h}_v^{(k)}$, para todos os nós vizinhos do nó de interesse u. Matematicamente, temos:

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in N(u)} \mathbf{h}_v. \tag{3.8}$$

Assim, já é possível definir uma estrutura cuja aplicabilidade em grafos se torna possível, sendo ela:

$$\mathbf{h}_u^{(k)} = \mathbf{W}_a^{(k)}\mathbf{h}_u^{(k-1)} + \mathbf{W}_b^{(k)} \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v^{(k-1)}. \tag{3.9}$$

Perceba que o primeiro termo se refere ao nó de interesse e a sua matriz de parâmetros relacionada e o segundo termo está relacionado aos vizinhos do nó de interesse e a sua matriz de parâmetros relacionada. Vale relembrar que a matriz de características $\mathbf{X}$ também está presente neste modelo, pois $\mathbf{h}_u^{(0)} = \mathbf{x}_u, \forall u \in E$.

Porém, sob uma análise mais criteriosa, percebe-se que a equação (3.9) poderia facilmente ser trabalhada utilizando métodos lineares, tais como regressão linear, por exemplo. Então, faz-se necessária a inclusão de um termo não-linear na expressão acima para que seja justificada sua aplicabilidade no trabalho em questão. Deste modo, defina-se σ como uma função de ativação e a estrutura de passagem de mensagems em grafos se torna completa, definida como:

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}_a^{(k)}\mathbf{h}_u^{(k-1)} + \mathbf{W}_b^{(k)} \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v^{(k-1)}\right). \tag{3.10}$$

Apesar de o modelo demonstrado em (3.10) já ser aplicável, é possível simplificá-lo ao adicionar o nó de interesse aos seus vizinhos na função de coleta de mensagem, dando origem ao modelo com ciclos. Ao fazê-lo, temos:

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}^{(k)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \mathbf{h}_v^{(k-1)}\right). \tag{3.11}$$

Os benefícios desta alternativa estão no fato de não ser mais necessária a divisão da matriz de parâmetros $\mathbf{W}$ em duas matrizes distintas e na exclusão completa de um termo na expressão. Contudo, deve-se tomar cuidado, pois aqui, não se torna mais possível diferenciar quais informações são provenientes dos vizinhos do nó e quais são do próprio nó.

3.4 Normalização das Vizinhanças

Como retratado em seções passadas, o método de coleta das informações se dá pela soma das informações dos vizinhos do nó de interesse. Porém, conforme apontado em

Hamilton (2020), isso pode ocasionar alguns percalços em nosso modelo. O problema mais evidente se torna fácil de entender ao lembrarmos que os nós de um grafo podem ter diferentes quantidades de vizinhos, a depender de fatores como suas conexões e posições dentro do universo do grafo.

Deste modo, em um cenário hipotético, é possível imaginar que, em um mesmo grafo, o nó u possua 5 vizinhos e o nó v possua 50 vizinhos. Naturalmente, espera-se que a soma das informações dos vizinhos do nó v seja maior do que a soma das informações dos vizinhos do nó u. Tal disparidade entre o número de conexões entre cada nó pode provocar distorções na interpretação das informações, pois uma soma alta pode estar muito mais relacionada à quantidade de conexões do que às informações propriamente ditas.

Tendo tais questões em mente, faz-se necessária a execução de um método que normalize a função de coleta das informações com base nos graus dos nós na função.

Para realizar tal correção, utiliza-se o método sugerido por Kipf e Welling (2017), chamado de *normalização simétrica*, que possui raízes e motivações atreladas à teoria espectral de grafos, que é denotada por:

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}. \tag{3.12}$$

3.5 Modelo final proposto

De acordo com Hamilton (2020), ao combinarmos o método de normalização simétrica da função de coleta, a função de atualização e ciclos ao modelo, acaba-se por criar uma aproximação de primeira-ordem de uma convolução espectral de grafos, cujo modelo é denotado por:

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}^{(k)} \sum_{v \in \mathcal{N}(u) \cup (u)} \frac{\mathbf{h}_v^{(k-1)}}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}\right). \tag{3.13}$$

O modelo descrito em (3.13) foi idealizado em Kipf e Welling (2017) e trata-se de uma das estruturas mais populares e eficientes relacionadas a redes neurais de grafos, de modo que será utilizada como foco deste Trabalho de Graduação a partir de agora.

3.6 Funções de Ativação

Grande parte da performance de um modelo de rede neural é impactada pela escolha de uma função de ativação adequada para os objetivos que se pretende alcançar. De maneira simplificada, uma função de ativação nada mais é do que a maneira que as informações são transmitidas dos neurônios de uma camada da rede neural para os neurônios de outra camada. Tais funções são inúmeras e possuem as mais variadas características, podendo ser limitadas ou infinitas, lineares ou não-lineares, dentre outros atributos que as diferenciam entre si e se estabelecem como mais adequadas para cada tipo de objetivo (como predição ou classificação) que se pretende alcançar ou até mesmo para cada tipo de estrutura da sua rede (como redes convolucionais ou recorrentes).

Diferentes funções de ativação podem ser utilizadas em diferentes camadas da rede neural. Como explicitado anteriormente, de maneira geral, todas as redes neurais são divididas em três camadas: entrada, intermediária e saída. Tem-se o interesse de selecionar uma função de ativação para as camadas intermediárias que melhor consiga se adaptar à estrutura do modelo de Redes Convolucionais de Grafos que está sendo construído e escolher outra função de ativação para as camadas de saída que sejam adequadas para o objetivo proposto para o projeto em questão.

Neste trabalho, serão analisadas como a escolha adequada da função de ativação para as camadas intermediárias de um modelo de grafos possui grande influência na performance e, consequentemente, nos resultados obtidos. Para tal, serão utilizadas três funções de ativação distintas:

- Linear
- Sigmoide (Logística)
- ReLU

A seguir, será dada uma breve introdução sobre cada uma das funções de ativação a serem utilizadas nas análises.

3.6.1 Linear

Considerada a mais simples de todas, a função de ativação linear também é chamada de função identidade e, como o próprio nome sugere, ela apenas multiplica suas entradas por

1. Por não aplicar nenhuma diferença prática nas informações que as camadas recebem, essa função também é considera de "não-ativação". Uma visualização simples desta função pode ser analisada na Figura 3.2:

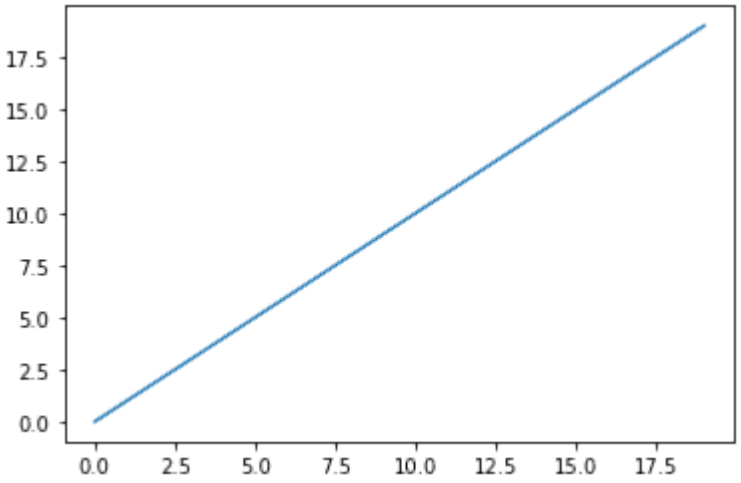

Figura 3.2: Entradas e saídas da função de ativação linear.

Percebe-se que se trata de uma simples reta identidade, com os valores no eixo das ordenadas refletindo os mesmos valores do eixo das abscissas.

3.6.2 Sigmoide (Logística)

A função de ativação sigmoide, também conhecida como função logística, é dada pela expressão

$$\sigma(x) = \frac{1}{(1 - e^{-x})}, \tag{3.14}$$

onde e é a base do logaritmo natural.

O papel dessa função é transformar os valores reais que a camada de saída recebe em valores entre 0 e 1 como os valores de saída da camada. Quanto maior o valor de x, mais perto de 1 será o valor de saída; quanto menor o valor, mais o valor de saída se aproxima do zero. Esse comportamento pode ser visualizado na Figura 3.3:

3.6.3 ReLU

Finalmente, a função de ativação ReLU (*Rectified Linear Unit*) é dada pela expressão

$$\sigma(x) = \max(0, x) \tag{3.15}$$

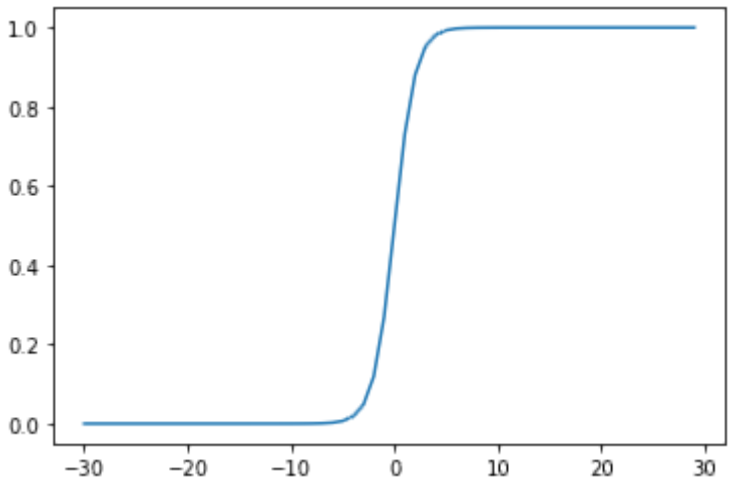

Figura 3.3: Entradas e saídas da função de ativação sigmoide.

De maneira simples, caso o valor de entrada seja negativo, a função assumirá o valor zero e, caso o valor seja positivo, a função agirá de forma idêntica à função de ativação linear. Esse comportamento pode ser visualizado na Figura 3.4.

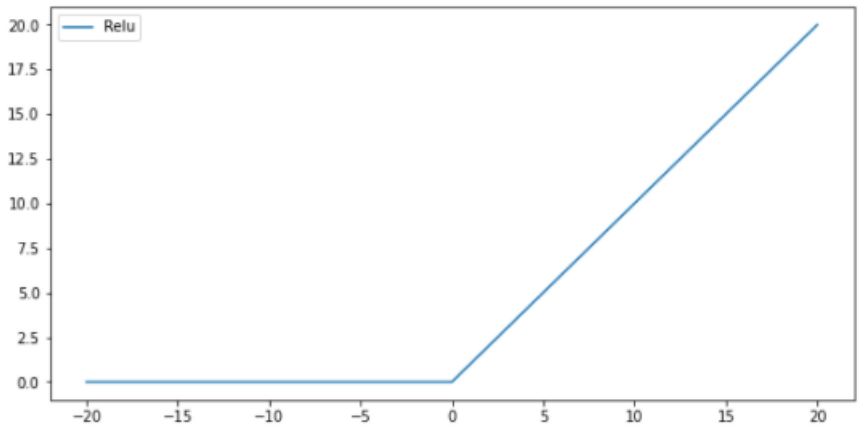

Figura 3.4: Entradas e saídas da função de ativação Relu.

3.7 Praticidade e Aplicações

O foco principal das seções anteriores foi fornecer embasamento teórico e apresentar a estrutura do modelo de aprendizagem de máquina que será utlizado na execução deste trabalho. Assim, faz-se interessante que também seja feita uma introdução e breve análise a respeito dos aspectos práticos desta metodologia, bem como aplicações e técnicas de otimização.

Conforme explicitado no início deste trabalho, o modelo apresentado aqui busca executar a tarefa de aprendizagem de máquina relacionada a *classificação de nós*. Deste modo, para realizar esta classificação e, posteriormente, verificar se ela é eficiente, é preciso aplicar uma função aos vetores imersos finais $\mathbf{z}_u$ para todo nó u. Para esta tarefa, são utilizadas as *funções de perda*. Segundo Chollet (2018), essas funções permitem medir a eficiência de um algoritmo, ao comparar a distância entre o *output* do algoritmo e o *output* esperado. Assim, esta medida é utilizada como *feedback* para realizar ajustes no algoritmo, de modo que uma menor função de perda resulta em um algoritmo mais eficiente. Para tal, é separado uma parte do conjunto de nós (geralmente pequena, cerca de 10% do total) para compor o conjunto de treino, denominado por $\mathcal{V}_{treino}$, e um vetor $\mathbf{y}_u$ via *one-hot encoding* que representa a classe do nó pertencente ao conjunto de treino. Porém, é necessária uma maneira de medir a probabilidade de que o nó pertença à classe $\mathbf{y}_u$. Para isto, existe uma função denominada *softmax*, que normaliza o *output* de uma rede em uma distribuição de probabilidade. Em outras palavras, ela pode receber vetores que não necessariamente seguem as propriedades de uma função de distribuição de probabilidade, tais como valores entre 0 e 1 e a soma de seu componentes ser igual a 1. De acordo com Hamilton (2020), a função *softmax* é definida por:

$$\text{softmax}(\mathbf{z}_u, \mathbf{y}_u) = \sum_{i=1}^{c} \mathbf{y}_u[i] \frac{e^{\mathbf{z}_u^\intercal \mathbf{w}_i}}{\sum_{i=1}^{c} e^{\mathbf{z}_u^\intercal \mathbf{w}_i}}. \tag{3.16}$$

É fácil perceber, pela divisão do termo pela soma do próprio termo, que a função se trata de uma normalização. Além disso, não há problemas com a multiplicação no termo exponencial pois tanto o vetor $\mathbf{w}$ quanto o vetor $\mathbf{z}$ possuem dimensão d, o que torna possível a obtenção de seu produto. Assim, tem-se as ferramentas necessárias para definir a seguinte função de perda:

$$\mathcal{L} = \sum_{u \in \mathcal{V}_{\text{train}}} -\log(\text{softmax}(\mathbf{z}_u, \mathbf{y}_u)). \tag{3.17}$$

Interpretando a (3.15), a função de perda $\mathcal{L}$ percorre todos os nós presentes no conjunto de treino, soma todas as probabilidades provenientes da função *softmax* e aplica-se a função de log-verossimilhança negativa.

Como explicitado, para que tenhamos um algoritmo eficiente, o ideal é de que a sua função de perda assuma o menor valor possível. Uma das técnicas mais utilizadas no ramo da aprendizagem de máquina para otimizar o algoritmo, ou seja, para encontrar

os parâmetros que minimizem a função de perda, é o método do *gradiente descendente*. Explicando de maneira simplificada, esta técnica consiste em analisar a curva da função e "caminhar" ao longo dela na direção oposta à sua inclinação, até que encontre um ponto de mínimo, ou seja, até que ela pare de cair e comece a subir novamente.

De acordo com Hamilton (2020), a tarefa de classificação de nós pode ser considerado um método de aprendizagem de máquina tanto supervisionado quanto semi-supervisionado. Como já explicitado, essa nomenclatura varia de acordo com a quantidade de informações prévias que o modelo possui. Contextualizando para o nosso caso, poderia-se pensar que essa classificação varia de acordo com os **tipos de nós** que o modelo utilizará. De modo geral, é possível identificar três tipos de nós:

- **Nós de treinamento**: são os nós que são incluídos no modelo de transmissão de informação e no cálculo da função de perda $\mathcal{L}$. Em outras palavras, seus nós possuem vetores imersos $\mathbf{h}$ e eles são utilizados no modelo, e seus vetores finais $\mathbf{z}$ são incluídos no cálculo da função $\mathcal{L}$;

- **Nós transdutivos de teste**: são os nós que são utilizados nas operações de transmissão de informação mas não são utilizados no cálculo da função de perda. Em outras palavras, seus nós possuem vetores imersos $\mathbf{h}$ e eles são utilizados no modelo, mas seus vetores finais $\mathbf{z}$ não são incluídos no cálculo da função $\mathcal{L}$.

- **Nós indutivos de teste**: são os nós que não são considerados nem no modelo de transmissão de mensagem, nem no cálculo da função de perda.

Segundo Hamilton (2020), o modelo proposto se encaixa nos moldes de aprendizagem semi-supervisionado pois o modelo será testado nos nós transdutivos, já que, neste caso, o modelo observará os nós de teste durante o treinamento, mas não saberá suas classificações.

Capítulo 4

Aplicação

Neste seção do trabalho, colocaremos em prática os objetivos que foram especificados no Capítulo 1. Reitera-se qye o objetivo desta seção é aplicar, na prática, o método de classificação de nós via Redes Convolucionais de Grafos de modo que seja possível nos familiarizarmos com a prática, bem como verificar como realizar o primeiro contato com a linguagen *Python* e verificar como é possível aplicar o método nesta linguagem. Em sumo, esta seção se propõe mais a apresentar um caráter mais exploratório do método a ser estudado neste projeto do que responder questionamentos provenientes dos bancos de dados a serem analisados.

Dito isso, aplicaremos um modelo de classificação de nós baseado em Redes Convolucionais de Grafos em um banco de dados real para verificar a precisão deste método e, em um segundo momento, para fazer uma avaliação mais voltada para as funções de ativação utilizadas nas camadas intermediárias do modelo. As três funções de ativação citadas no trabalho serão executadas no mesmo modelo nos bancos de dados em questão e seus resultados serão comparados para que possamos melhor entender qual é a mais adequada para a tarefa de classificação de nós.

Todas as análises foram realizadas via *Jupyter Notebook*, um aplicativo web *open-source* em que é possivel utilizar a linguagem *Python*. Para este projeto, além dos pacotes-base de manipulação e preparação de dados que já vem automaticamente com o *Python 3.6*, também foi utilizado o pacote *Stellargraph*. De modo sucinto, este pacote oferece uma vasta gama de aplicações de aprendizado de máquina voltados especificamente para estruturas em grafos. Dentre as suas aplicações, estão:

- Predição de arestas;

- Classificação de grafos;
- Classificação de nós, objetivo deste trabalho.

Para este projeto, o procedimento de análise prosseguirá da seguinte maneira:

- Primeiramente, será necessário que sejam importados os dados do grafo, realizar uma breve manipulação e dividi-los em bases de treino, teste e validação para que o modelo possa ser treinado. Esta etapa pode ser realizada utilizando pacotes padrões de *Python*, como *Pandas* e *scikit-learn*;
- O segundo passo consiste em criar o modelo de fato, isto é, criar as camadas e configurar como os dados serão inseridos no modelo. O pacote *Stellargraph* possui uma variedade de atributos e módulos que realizam essa etapa sem maiores dificuldades;
- Por fim, depois que o modelo é criado, é necessário treiná-lo e, logo após, avaliar a sua precisão. Este passo pode ser realizado com pacotes como *Tensorflow*, *Keras*, *Pandas* e *Scikit-learn*.

4.1 Banco de dados CORA

O banco de dados CORA (Sen *et al.* (2008)) é composto por artigos científicos relacionados a aprendizagem de máquina. Esses artigos foram classificados em sete categorias distintas, sendo elas:

- Baseados em Casos;
- Algoritmos Genéticos;
- Redes Neurais;
- Métodos Probabilísticos;
- Aprendizagem por Reforço
- Baseado em Regras;
- Teoria.

Cada artigo presente nesse banco de dados será representado por um nó deste grafo, sendo que os artigos foram selecionados de modo que cada artigo cita ou é citado pelo menos uma vez por outro artigo. Em outras palavras, as arestas desse gráfico seriam as citações que os artigos fazem entre si. Para identificar a categoria a que cada artigo pertence, foi realizado um pré-processamento em todas as palavras identificadas nos artigos, de maneira que, ao final, houvesse um dicionário composto de 1433 palavras únicas, que serão utilizadas como covariáveis para o modelo.

O banco de dados possui 2710 artigos científicos e a rede possui 5429 conexões distintas. A Figura 4.1 representa, de maneira visual, como se comporta o grafo deste banco de dados.

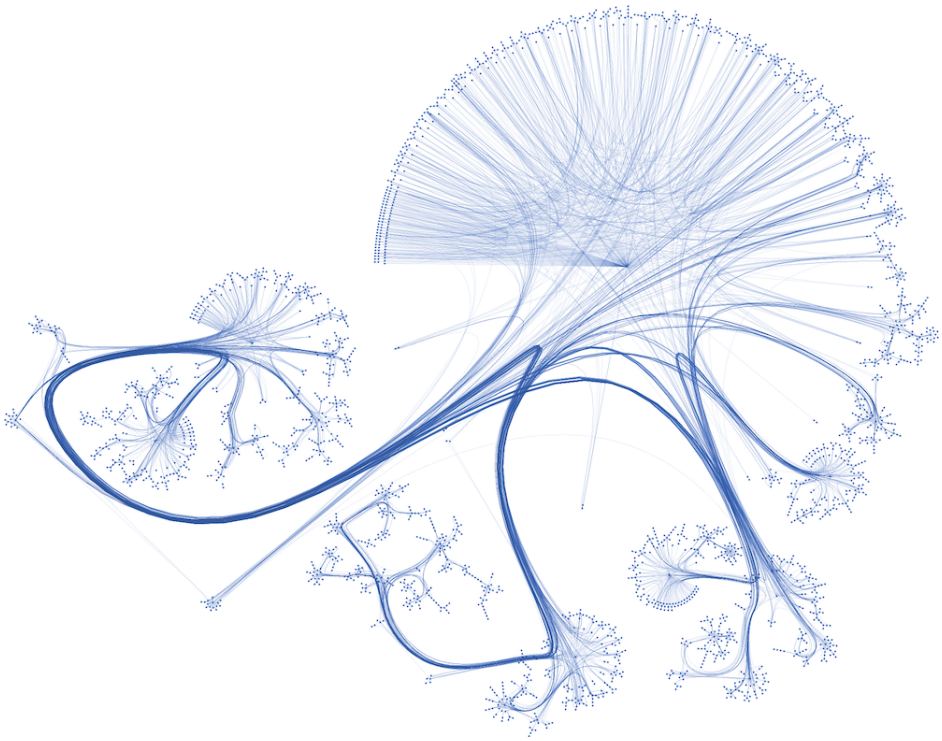

Figura 4.1: Visualização gráfica do grafo referente ao banco de dados CORA. Fonte: https://graphsandnetworks.com/the-cora-dataset

Por se tratar de uma rede de citações, o grafo original é direcionado mas, neste projeto, ele será tratado como não-direcionado.

4.1.1 Preparação dos dados

O primeiro passo é importar as bibliotecas *Python* necessárias para a análise. Instalando no ambiente adequado e utilizando o comando *import*, serão utilizadas as seguintes

bibliotecas:

- **Pandas**: um pacote construído em *Python* que é amplamente utilizado para análise e manipulação de dados;
- **Os**: um pacote importante que fornece funções que interagem de maneira direta com o sistema operacional do dispositivo;
- **Stellargraph**: já descrito anteriormente, é um pacote voltado para a aprendizagem de máquina em dados em estruturas de grafos;
- **Tensorflow**: Tensorflow é um pacote produzido e lançado pela Google para manipulação e execução numérica de maneira ágil. Também serve de base para outros modelos e pacotes de aprendizagem profunda e aprendizagem de máquina;
- **Keras**: é uma interface para aprendizagem profunda elaborada em *Python*, que trabalha conjuntamente com a plataforma Tensorflow;
- **Scikit-Learn**: considerada a biblioteca em *Python* mais utilizada para aprendizagem de máquina, com funcionalidades que abrangem modelos de classificação, regressão e clusterização;
- **Matplotlib**: se trata de uma biblioteca para criar toda sorte de visualizações em *Python*, sejam elas estáticas, animadas ou interativas.

O pacote *Stellargraph* possui um módulo que já inclui uma variedade de *datasets* disponíveis para uso, e dentre eles está o CORA, de modo a tornar sua importação muito mais prática.

Os códigos utilizados neste trabalho também foram baseados na documentação pública do *site*[1] do pacote *Stellargraph*. Lá, além de códigos e conteúdos a respeito de Redes Convolucionais de Grafos e outras aplicações, também são disponibilizadas demonstrações e artigos referentes a cada um dos métodos.

Como queremos elaborar um algoritmo que consiga prever a "categoria"dos artigos, vamos analisar como os nós estão distribuídos entre elas.

[1] https://stellargraph.readthedocs.io/en/v1.2.1/demos/node-classification/gcn-node-classification.html

Tabela 4.1: Número de artigos associados a cada tema de aprendizado de máquina.

Tema	**Nº de artigos**	**Porcentagem relativa**
Redes Neurais	818	30.19%
Métodos Probabilísticos	428	15.79%
Algoritmos Genéticos	418	15.42%
Teoria	351	12.95%
Baseado em Casos	298	11.00%
Aprendizado por Reforço	217	8.01%
Baseado em Regras	180	6.64 %

Algo interessante que a Tabela 4.1 mostra é como o tema de Redes Neurais permanece atual, interessante e *em voga* em comparação com outras áreas que compõem o campo de aprendizado de máquina.

Como acontece em qualquer outro modelo de aprendizado de máquina, seja de predição, classificação ou clusterização, é necessário que uma parte da base de dados seja separada para treinar o modelo, outra parte do banco de dados é utilizado para testar a performance do modelo e o restante da base é alocado em um terceiro grupo para fins de validação. Para esta análise, foi feita a seguinte separação:

Tabela 4.2: Divisão do banco de dados.

Grupo	**Nº de nós**	**Porcentagem**
Treino	140	5,1%
Validação	500	18,5%
Teste	2068	76,4%

Utilizando amostragem estratificada nos nós presentes no grupo de treino para manter a proporção entre as classes, tem-se a seguinte composição:

Tabela 4.3: Nós presentes no grupo de treino.

Tema	**Nº de artigos**	**Porcentagem relativa**
Redes Neurais	42	30%
Métodos Probabilísticos	22	15.71%
Algoritmos Genéticos	22	15.71%
Teoria	18	12.86%
Baseado em Casos	16	11.43%
Aprendizado por Reforço	11	7.86%
Baseado em Regras	9	6.43%

Como a variável de interesse é categórica, será necessário usar vetores *one-hot* para serem comparados aos resultados que sairão na camada de saída do modelo. No banco de dados em questão, há atributos $\mathbf{x}_u$ correspondentes às palavras encontradas no artigo.

Se a palavra ocorre mais do que uma vez no artigo, o atributo relacionado assume o valor um; caso contrário, assume o valor zero.

4.1.2 Criação do modelo

Um modelo de aprendizado de máquina construído no pacote *Stellargraph* é composto de alguns argumentos. São eles:

- As próprias camadas da rede neural;
- Um gerador de dados que consiga converter a estrutura de grafos e as características dos nós (ou seja, a matriz de adjacência e a matriz de características) em um formato que possa ser usado para treino e predição.

Para definir o gerador para o nosso modelo, foi utilizada a função *FullBatchNodeGenerator* para esta tarefa, especificando GCN (*Graph Convolutional Network, Redes Convolucionais de Grafos em inglês*) como o argumento da função. Ao fazê-lo, será utlizada a matriz Laplaciana para que se realize a adequação da estrutura do grafo no modelo. Então, o gerador coletará a informação necessária para produzir os *inputs* do modelo.

Agora, será necessário construir as camadas. Para tal, será utilizada a função **GCN**, também do pacote *Stellargraph*, que pode ser configurada pelos seguintes parâmetros:

- **Tamanho das camadas**: o número de camadas intermediárias e a quantidade de neurônios em cada uma. Neste caso, serão utilizadas duas camadas com 16 neurônios cada.
- **Ativações**: as funções de ativação aplicadas em cada camada intermediária do modelo. Como já dito anteriormente, serão utilizadas as funções *ReLU*, logística e linear.
- ***Dropout***: O *Dropout* nada mais é do que uma técnica para "ignorar" neurônios que estejam poluindo a sessão de treinamento do modelo em alguma das camadas.

O próximo passo é criar um modelo em *Keras*. Para isto, é necessário atribuir as estruturas algébricas de entrada e saída do modelo especificamente para a predição de nós, através do método *gcn.in.out.tensors*. Aqui, vale lembrar que, em *Python*, um *método* é similar a uma função, com a diferença de poder ser associado a objetos e/ou classes.

Essa estrutura algébrica de saída do modelo, conterá um vetor de 16 dimensões para os nós quando estiver treinando ou realizando a predição, de modo que as predições para a classe de cada nó devem ser computadas a partir deste vetor. Por exemplo, no caso da função de ativação *Softmax*, aplicada nas camadas de saída do modelo, ela assegura que a saída para cada nó de entrada seja algo semelhante a um vetor de probabilidades, onde cada vetor assuma valores entre 0 e 1, a soma total seja 1, e a classe com o maior valor é a classe predita pelo modelo.

4.1.3 Treino e avaliação da eficácia (*ReLU*)

Nesta seção, será abordado como o modelo será treinado e avaliado, de acordo com métricas de acurácia e perda. Aqui, será abordado a função de ativação *ReLU*, mas as outras seguirão o mesmo princípio.

Agora, para criar de fato o modelo em uma estrutura *Keras*, será necessário utilizar as estruturas de saída do modelo, que serão as classificações da última camada. Nesta etapa, é necessário definir uma função de perda para o modelo, como descrita durante a metodologia. Como a função do modelo é classificar uma categoria, faz sentido utilizar uma função de perda de entropia cruzada categórica. A beleza e a vantagem de utilizar a linguagem *Python* neste projeto é que, apesar de estarmos utilizando o pacote *Stellargraph* para montar o modelo propriamente dito, a parte preditiva do modelo será construída em uma estrutura *Keras*, que já possui a função de perda desejada embutida em seu pacote.

Contudo, conforme o modelo é treinado, é interessante que a performance do conjunto de validação seja verificada. Para ajudar nesta tarefa, o Keras possui uma função denominada *EarlyStopping*, que interrompe o treinamento assim que for identificado que a acurácia parou de crescer.

Finalmente, com o modelo e os dados devidamente tratados, pode-se começar a treinar o modelo, através do método *fit*, também incluso no *Keras*. Para este treino, serão utilizadas 30 iterações (ou *epochs*). A Figura 4.2 mostra um pouco de como o código realiza este procedimento.

Durante o treino, como mostra a Figura 4.2, alguns dados aparecem, sendo eles:

- *Loss*: se refere à "perda" do modelo. Quanto maior a perda, mais longe o valor predito está do valor real;
- *Accuracy*: trata da acurácia do modelo. É uma métrica que varia entre 0 e 1, e

```
history = model.fit(
    train_gen,
    epochs=30,
    validation_data=val_gen,
    verbose=2,
    shuffle=False,  # Tem que ser falso, pois ''embaralhar'' os dados implica em ''embaralhar'' o grafo
    callbacks=[es_callback],
)
```

```
Epoch 1/30
1/1 - 2s - loss: 1.9492 - acc: 0.0500 - val_loss: 1.9136 - val_acc: 0.3020 - 2s/epoch - 2s/step
Epoch 2/30
1/1 - 0s - loss: 1.9072 - acc: 0.3286 - val_loss: 1.8816 - val_acc: 0.3040 - 255ms/epoch - 255ms/step
Epoch 3/30
1/1 - 0s - loss: 1.8638 - acc: 0.2929 - val_loss: 1.8473 - val_acc: 0.3040 - 237ms/epoch - 237ms/step
Epoch 4/30
1/1 - 0s - loss: 1.8216 - acc: 0.3000 - val_loss: 1.8072 - val_acc: 0.3020 - 273ms/epoch - 273ms/step
Epoch 5/30
1/1 - 0s - loss: 1.7547 - acc: 0.3000 - val_loss: 1.7649 - val_acc: 0.3020 - 221ms/epoch - 221ms/step
Epoch 6/30
1/1 - 0s - loss: 1.6655 - acc: 0.3143 - val_loss: 1.7235 - val_acc: 0.3020 - 222ms/epoch - 222ms/step
Epoch 7/30
1/1 - 0s - loss: 1.6127 - acc: 0.3000 - val_loss: 1.6779 - val_acc: 0.3020 - 239ms/epoch - 239ms/step
```

Figura 4.2: Início do treino do modelo de classificação com a função ReLU no banco de dados CORA.

quanto maior seu valor, maior o índice de acertos do modelo.

- *Val.loss*: mesma interpretação da perda, mas aplicada ao conjunto de validação;
- *Val.accuracy*: mesma interpretação da acurácia, mas aplicada ao conjunto de validação.

Para visualizar como os índices de acurácia e perda se comportaram ao longo do treinamento, pode-se dispor de gráficos como os presentes na Figura 4.3.

A Figura 4.3 mostra que o conjunto de treino começou com uma acurácia mínima, teve um salto assim que os treinos começaram e se comportou de maneira parecida com o conjunto de validação até, aproximadamente, a metade do treinamento, quando passou a apresentar uma acurácia maior e se estabilizou próximo ao 0.9.

Algo importante a se notar ao avaliar a acurácia de um modelo é de que, se identificar que a acurácia estiver estagnada a partir de um certo número de iterações (ou *epochs*), o modelo já atingiu o ápice e, qualquer trabalho a partir dali, acaba se tornando perda de esforço, tempo e dinheiro (a depender do projeto) do pesquisador.

Já com relação à perda, percebe-se que, como espera-se de um modelo de aprendizagem de máquina, conforme o modelo é treinado, espera-se que o modelo acerte cada vez mais conforme ele aprende mais sobre o banco de dados. Neste caso, percebe-se que o conjunto de treino apresentou um decaimento na perda do modelo mais acentuada do que o conjunto

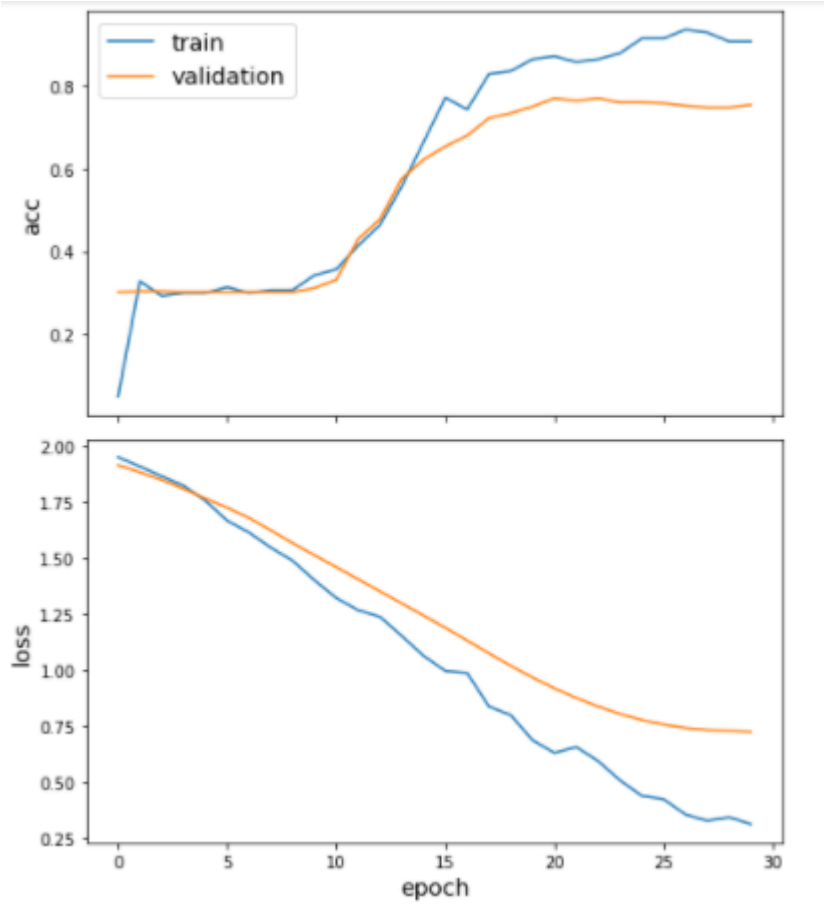

Figura 4.3: Acurácia e perda do modelo de classificação com a função ReLU.

de validação, terminando o treinamento com um valor próximo a 0.25, enquanto o conjunto de validação finalizou seu treino com uma perda quase três vezes maior.

Finalmente, a última parte da avaliação do modelo consiste em utilizá-lo no conjunto de teste, separado previamente. Então, para criar os dados de teste, é utilizado o método *Flow*, junto com o método *Evaluate* para gerar as métricas necessárias, como mostrado na Figura.

```
test_metrics = model.evaluate(test_gen)
print("\nTest Set Metrics:")
for name, val in zip(model.metrics_names, test_metrics):
    print("\t{}: {:0.4f}".format(name, val))
```

```
1/1 [==============================] - 0s 135ms/step - loss: 0.6805 - acc: 0.7872

Test Set Metrics:
        loss: 0.6805
        acc: 0.7872
```

Figura 4.4: Acurácia e perda do modelo de classificação no conjunto de teste com a função ReLU.

Percebe-se que o resultado do modelo junto ao conjunto de teste foi similar ao identificado pelo conjunto de validação durante o treino do modelo, com uma perda próxima de 0.7 e uma acurácia próxima de 0.8.

4.1.4 Classificações e Resultados

Para realizar a classificação, também será utilizado o método *flow* mas, dessa vez, serão utilizados os nós sem suas classes, pois são estas que o modelo tem que prever. No caso da função *Softmax*, como a camada final do modelo é um vetor de probabilidades, é necessário utilizar o método *inverse.transform* para transformar esses valores de volta em categorias. Depois de realizados esses passos, a Tabela 4.4 apresenta algumas das predições do modelo.

Tabela 4.4: Classificações dos 20 primeiros nós, com a função ReLU no banco de dados CORA.

Categoria predita	**Categoria Real**
Redes Neurais	Redes Neurais
Baseado em Regras	Baseado em Regras
Aprendizagem Reforçada	Aprendizagem Reforçada
Aprendizagem Reforçada	Aprendizagem Reforçada
Métodos Probabilísticos	Métodos Probabilísticos
Métodos Probabilísticos	Métodos Probabilísticos
Aprendizagem Reforçada	Teoria
Redes Neurais	Redes Neurais
Redes Neurais	Redes Neurais
Teoria	Teoria
Métodos Probabilísticos	Redes Neurais
Redes Neurais	Algoritmos Genéticos
Métodos Probabilísticos	Métodos Probabilísticos
Teoria	Baseado em Casos
Redes Neurais	Redes Neurais
Redes Neurais	Redes Neurais
Aprendizagem Reforçada	Aprendizagem Reforçada
Redes Neurais	Redes Neurais
Redes Neurais	Redes Neurais
Redes Neurais	Redes Neurais

Da Tabela 4.4, tem-se que o modelo conseguiu um bom trabalho ao prever as categorias dos artigos científicos mas, aliado às métricas apresentadas anteriormente, também é interessante que essas classificações sejam representadas de maneira visual, de maneira a verificar como cada um dos artigos se localizam em um espaço em duas dimensões e se é possível agrupá-los em grupos semelhantes.

Para isso, serão utilizados os vetores imersos relacionados a cada um dos artigos, de modo que cada "imersão" seja um ponto em um gráfico de dispersão.

Para realizar esta tarefa, será necessário substituir a camada relacionada à predição do modelo para a estrutura algébrica de saída da rede convolucional.

O problema é que a última camada da rede convolucional possui dimensão 16, ou seja, a imersão relacionada a cada nó possui 16 números. Para visualizar essas imersões, seria necessário um gráfico de 16 dimensões, algo inviável para este projeto. Então, é preciso realizar uma redução de dimensionalidade nesses vetores, de 16 números para 2, para que seja possível visualizá-los em um gráfico 2D.

Apesar do método de redução de dimensionalidade mais utilizado ser a Análise de Componentes Principais, como essa técnica foi bastante abordada ao longo do curso, optou-se por utilizar um novo método, denominado "Incorporação de vizinhos estocásticos com distribuição t".

Assim, depois de realizar as imersões e reduzir a dimensionalidade dos vetores, pode-se utilizar os pacotes *Pandas* e *Matplotlib* para definir cores às categorias e, por fim, chegar em um resultado semelhante ao da Figura 4.5. Percebe-se uma clara distinção entre sete cores diferentes, representativas das sete categorias de artigos de aprendizagem de máquina identificados no banco de dados.

4.1.5 Resultados com a função de ativação linear

Agora, como proposto anteriormente, será utilizado o mesmo banco de dados para verificar como diferentes funções de ativação interferem na performance do método e, para este caso, na classificação adequada dos artigos científicos.

Os procedimentos utilizados nesta seção serão idênticos aos mostrados e descritos na seção anterior. A única diferença será a troca da função de ativação das camadas intermediárias do modelo. Neste caso, será usada a função linear, cujos gráficos de acurácia e perda estão descritos na Figura 4.6.

Percebe-se pelos gráficos de acurácia e perda presentes na Figura 4.6 que a função de ativação linear performou de maneira muito semelhante em termos de métrica com a função de ativação ReLU, terminando a sessão de treinamento com uma acurácia próxima do valor máximo para o conjunto de treino e com um valor quase mínimo de perda para este mesmo conjunto. Talvez, o motivo por trás de performances tão semelhantes possa ser explicado pelo fato das duas funções terem comportamento semelhante para valores

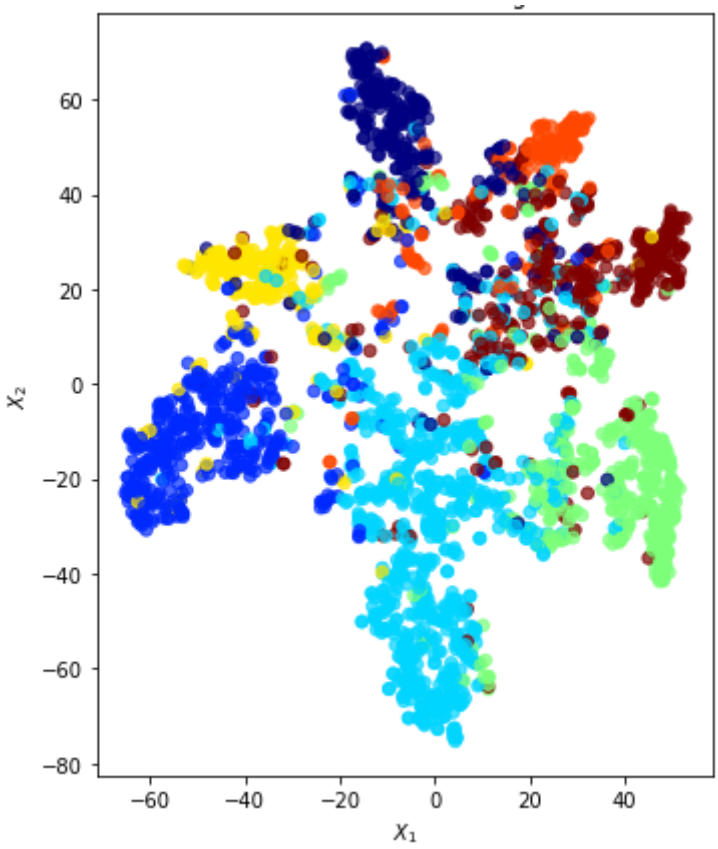

Figura 4.5: Visualização das classificações do banco de dados CORA com a função ReLU.

positivos de entrada na função.

Para termos um critério de comparação com a função de ativação anterior, esse modelo também foi utilizado no conjunto de teste do banco de dados, e os valores finais de acurácia e perda foram plotados na Tabela 4.5.

Tabela 4.5: Acurácia e perda das funções de ativações ReLU e linear.

Função de Ativação	**Acurácia**	**Perda**
ReLU	0.7872	0.6805
Linear	0.7737	0.7769

Como a Figura 4.6 já apontou, a diferença de performance do modelo utilizando as duas funções de ativação testadas é mínima, sendo mais percebida na comparação dos valores de perda do que na acurácia propriamente dita. Para checar se essa diferença é ainda mais ressaltada ou diminuída, também foi aplicarda a redução de dimensionalidade e construído o gráfico de visualização das classificações, denotado na Figura 4.7.

Analisando a Figura 4.7, observa-se um comportamento muito parecido com o identificado com a função de ativação ReLU. Aqui, também percebe-se sete grupos distintos de cores agrupados, porém, com uma intersecção maior e um maior agrupamento de pontos

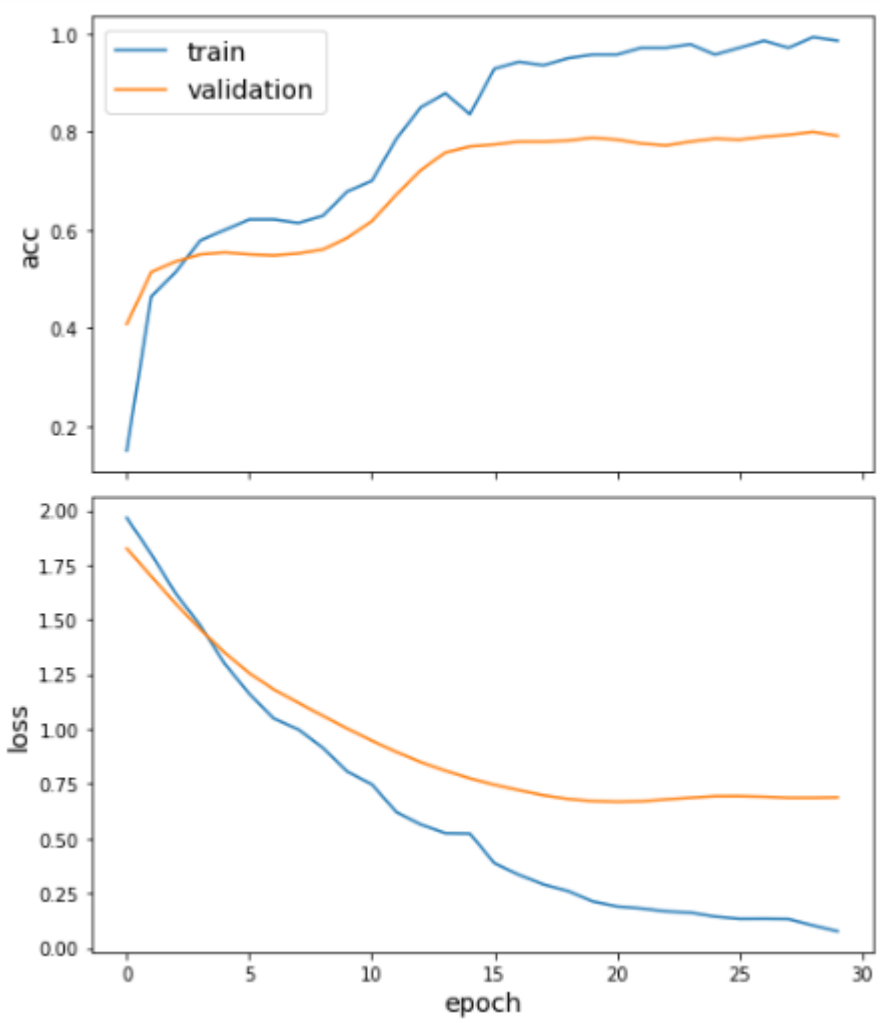

Figura 4.6: Acurácia e perda do modelo de classificação com a função linear.

na região inferior do gráfico.

4.1.6 Resultados com a função de ativação *Sigmoide*

Utilizar a função de ativação linear mostrou que pode ser uma alternativa viável para este conjunto de dados caso o projeto de aprendizagem de máquina seja voltado para classificação de nós. Com o intuito de descobrir se outras funções de ativação aplicadas nas camadas intermediárias do modelo também apresentam resultados promissores, optou-se por utilizar uma terceira alternativa para verificar se ela consegue performar tão bem quanto as duas funções anteriores. Então, o modelo foi testado utilizando a função de ativação sigmoide (ou logística), cujos gráficos de acurácia e perda estão dispostos na Figura 4.8.

Diferentemente das duas funções anteriores, já é possível perceber que a função sig-

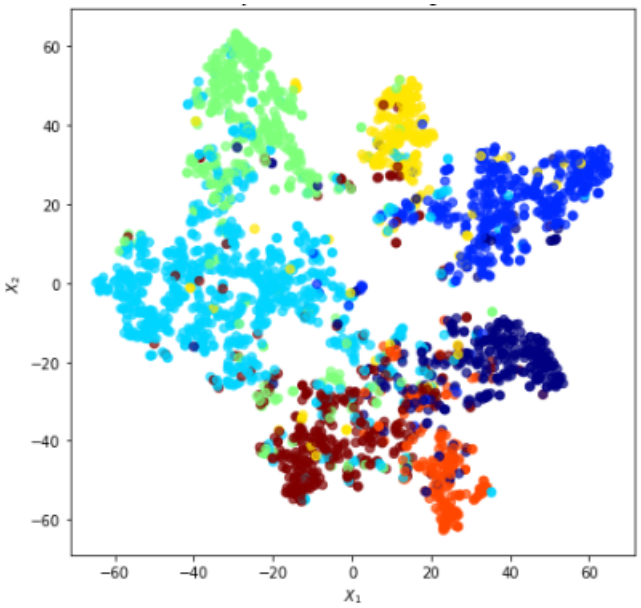

Figura 4.7: Visualização das classificações do banco de dados CORA com a função linear.

moide talvez não seja uma alternativa tão viável para projetos de classificação de nós. Apesar da acurácia e da perda apresentarem, respectivamente, comportamentos de aumento e decrescimento ao longo da sessão de treinamento, ambas terminaram com valores muito aquém daqueles apresentados pelas funções anteriores, com a acurácia em torno de 0.4 e a perda próxima de 1.6.

Também utilizando este modelo com o conjunto de teste para verificar a métrica de acurácia e perda, chegou-se nos valores apontados na Tabela 4.6, que também mostra as métricas com as funções de ativação anteriores para fins de comparação.

Tabela 4.6: Acurácia e perda das funções de ativações ReLU, linear e sigmoide.

Função de Ativação	**Acurácia**	**Perda**
ReLU	0.7872	0.6805
Linear	0.7737	0.7769
Sigmoide	0.3477	1.6109

Pela Tabela 4.6, nota-se ainda mais a discrepância entre as performances entre a função de ativação sigmoide e as outras duas, com a primeira apresentando menos da metade da acurácia e mais do que o dobro da perda em comparação com as funções ReLU e linear.

Finalmente, a visualização das predições do banco de dados também corroboram a interpretação de que a função de ativação sigmoide não é a melhor escolha para este tipo de projeto, como pode ser observado na Figura 4.9.

De modo geral, a Figura 4.9 apresenta os grupos muito mais próximos do que nos

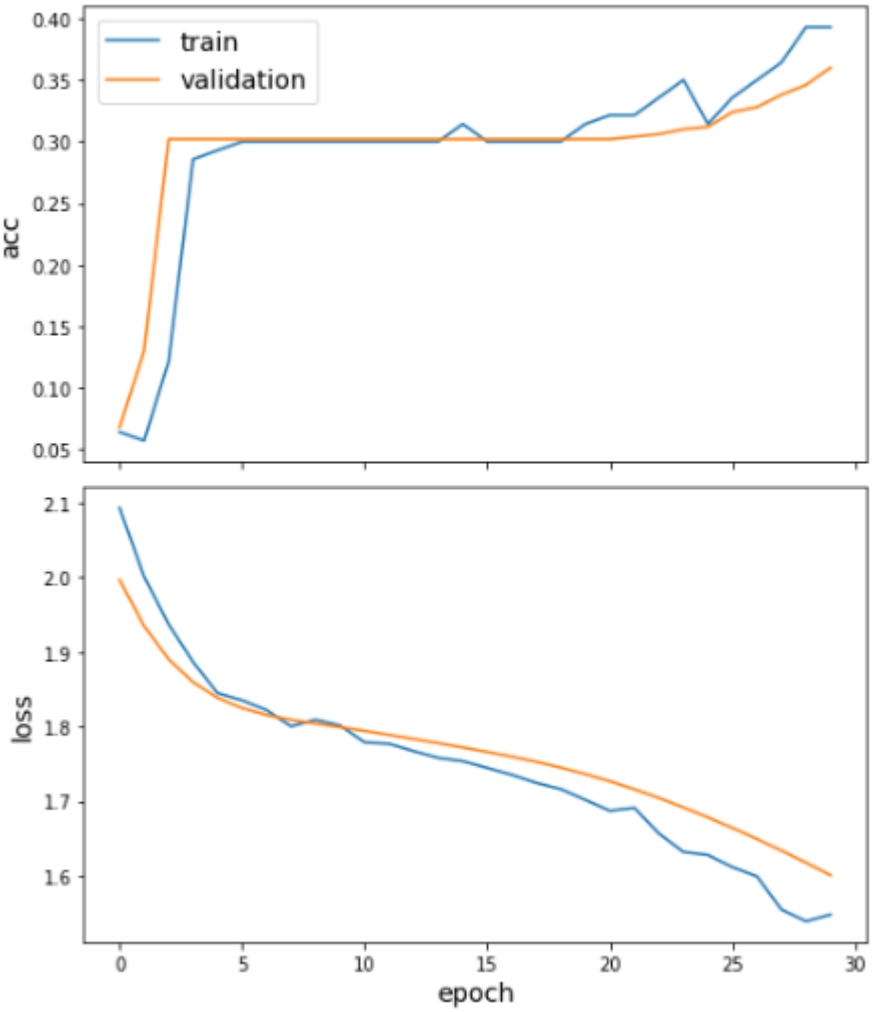

Figura 4.8: Acurácia e perda do modelo de classificação com a função sigmoide.

outros gráficos, com intersecções maiores e algumas regiões do gráfico com um grande número de pontos distintos.

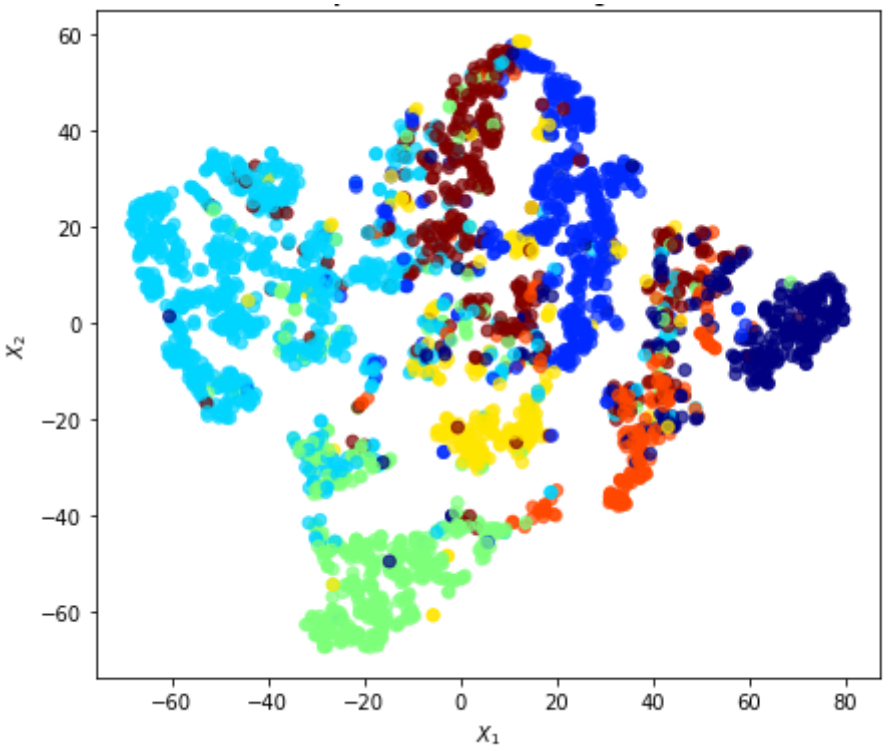

Figura 4.9: Visualização das classificações do banco de dados CORA com a função sigmoide.

Capítulo 5

Considerações Finais

O foco principal desse projeto foi alicerçado em três pilares fundamentais: o primeiro, mais importante, era estudar e se aprender, a fundo, o método de Redes Convolucionais de Grafos, de modo a entender como uma técnica de classificação utilizando redes neurais funcionaria em dados estruturados em grafos, bem como testar e avaliar as consequências de realizar certas mudanças em suas configurações, como a troca de funções de ativação, ilustradas neste projeto.

O segundo pilar que serviu como base para esse Trabalho de Graduação foi a oportunidade de realizar este projeto na linguagem de programação *Python*. Como descrito no Capítulo 4, a sua vasta gama de pacotes, sua facilidade de execução e acessibilidade se mostraram essenciais para que este projeto pudesse ser realizado sem maiores problemas. Dentre as funcionalidades mais úteis para este Trabalho, destaca-se o pacote *Stellargraph* que, com suas funções automaticamente integradas de importação de *datasets* e construção de modelos, facilitou a execução dos treinos e dos gráficos de maneira indescritível.

O terceiro ponto fundamental que motivou a elaboração deste projeto é a construção de um material específico para este tema em português e, em especial, com uma escrita mais voltada para o didatismo, de maneira que até mesmo pessoas completamente leigas no assunto pudessem compreender o que está sendo realizado e os resultados atingidos.

Então, ao ilustrar na prática como o método funciona e coletando seus resultados, identificou-se que as funções ReLU e linear, quando aplicadas nas camadas intermediárias do modelo, obtiveram resultados promissores e, frisa-se, semelhantes com relação à tarefa de classificação de nós do banco de dados CORA. Pode-se sugerir que isso se deve ao fato de ambas as funções apresentarem comportamentos semelhantes a partir de valores positivos de entrada, mas outros estudos comparativos podem ser realizados para verificar

tal hipótese. Além disso, notou-se que a função de ativação sigmoide não obteve resultados satisfatórios e, então, provou-se ineficaz na tarefa de classificação de nós.

Ressalta-se, novamente, que a aplicação exposta neste Trabalho de Graduação é de caráter ilustrativo e, além do método estudado poder ser utilizado (com sucesso) em outras áreas que utilizam redes neurais e classificação de nós, outros testes poderiam ser realizados, como mudança no número de camadas intermediárias ou até mesmo no número de neurônios contidos em cada camada, para avaliar de que forma isso impactaria na performance do modelo. O campo de atuação abordado neste trabalho está em completa expansão e, com o *boom* de popularidade que redes neurais e grafos estão recebendo nos últimos anos, não há dúvidas de que há muitas técnicas a serem descobertas e aperfeiçoados, revelando um futuro extremamente promissor para as área de aprendizado de máquina e estatística.

Referências Bibliográficas

Chollet, F. (2018). Deep learning with python. *Manning Publications*, **1**(1), 1–386.

Hamilton, W. L. (2020). Graph representation learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, **14**(3), 1–159.

Izbicki, R. e dos Santos, T. M. (2020). *Aprendizado de máquina: uma abordagem estatística*. ISBN 978-65-00-02410-4.

Kipf, T. N. e Welling, M. (2017). Semi-supervised classification with graph convolutional networks. Em *International Conference on Learning Representations (ICLR)*.

McCarthy, J. (2021). Universidade de stanford. http://infolab.stanford.edu/pub/voy/museum/samuel.html. Acesso em: 13 out. 2021.

McCulloch, W.S., P. (1943). A logical calculus of the ideas immanent in nervous activity. https://link.springer.com/article/10.1007%2FBF02478259. Acesso em: 10 out. 2021.

Sen, P., Namata, G., Bilgic, M., Getoor, L., Galligher, B. e Eliassi-Rad, T. (2008). Collective classification in network data. *AI magazine*, **29**(3), 93–93.

Apêndice A

Códigos Utilizados

```
!pip install tensorflow

!pip install stellargraph

!pip install scikit-learn

import pandas as pd
import os

import stellargraph as sg
from stellargraph.mapper import FullBatchNodeGenerator
from stellargraph.layer import GCN

from tensorflow.keras import layers, optimizers, losses, metrics, Model
from sklearn import preprocessing, model_selection
from IPython.display import display, HTML
import matplotlib.pyplot as plt
%matplotlib inline

dataset = sg.datasets.Cora()
display(HTML(dataset.description))
G, node_subjects = dataset.load()
```

```
print(G.info())

node_subjects.value_counts().to_frame()

train_subjects, test_subjects = model_selection.train_test_split(
    node_subjects, train_size=140, test_size=None, stratify=node_subjects
)
val_subjects, test_subjects = model_selection.train_test_split(
    test_subjects, train_size=500, test_size=None, stratify=test_subjects
)

train_subjects.value_counts().to_frame()

target_encoding = preprocessing.LabelBinarizer()

train_targets = target_encoding.fit_transform(train_subjects)
val_targets = target_encoding.transform(val_subjects)
test_targets = target_encoding.transform(test_subjects)

generator = FullBatchNodeGenerator(G, method="gcn")

train_gen = generator.flow(train_subjects.index, train_targets)

######## TESTE PARA RELU ##########

gcn = GCN(
    layer_sizes=[16, 16], activations=["ReLU", "ReLU"],
    generator=generator,
    dropout=0.5
)

x_inp, x_out = gcn.in_out_tensors()
```

```
x_out

predictions = layers.Dense(units=train_targets.shape[1],
activation="softmax")(x_out)

model = Model(inputs=x_inp, outputs=predictions)
model.compile(
    optimizer=optimizers.Adam(lr=0.01),
    loss=losses.categorical_crossentropy,
    metrics=["acc"],
)

val_gen = generator.flow(val_subjects.index, val_targets)

from tensorflow.keras.callbacks import EarlyStopping

es_callback = EarlyStopping(monitor="val_acc",
patience=50,
restore_best_weights=True)

history = model.fit(
    train_gen,
    epochs=30,
    validation_data=val_gen,
    verbose=2,
    shuffle=False,  # Tem que ser falso, pois ''embaralhar''
    # os dados implica em ''embaralhar'' o grafo
    callbacks=[es_callback],
)

sg.utils.plot_history(history)

test_gen = generator.flow(test_subjects.index, test_targets)
```

```
test_metrics = model.evaluate(test_gen)
print("\nTest Set Metrics:")
for name, val in zip(model.metrics_names, test_metrics):
    print("\t{}: {:0.4f}".format(name, val))

all_nodes = node_subjects.index
all_gen = generator.flow(all_nodes)
all_predictions = model.predict(all_gen)

node_predictions = target_encoding.inverse_transform(all_predictions.squeeze())

df = pd.DataFrame({"Predicted": node_predictions, "True": node_subjects})
df.head(20)

embedding_model = Model(inputs=x_inp, outputs=x_out)

emb = embedding_model.predict(all_gen)
emb.shape

from sklearn.decomposition import PCA
from sklearn.manifold import TSNE

transform = TSNE

X = emb.squeeze(0)
X.shape

trans = transform(n_components=2)
X_reduced = trans.fit_transform(X)
X_reduced.shape

fig, ax = plt.subplots(figsize=(7, 7))
```

```
ax.scatter(
    X_reduced[:, 0],
    X_reduced[:, 1],
    c=node_subjects.astype("category").cat.codes,
    cmap="jet",
    alpha=0.7,
)
ax.set(
    aspect="equal",
    xlabel="$X_1$",
    ylabel="$X_2$",
    title=f"{transform.__name__},
)

######## TESTE PARA LINEAR ##########

gcn = GCN(
    layer_sizes=[16, 16], activations=["linear", "linear"],
    generator=generator,
    dropout=0.5
)

x_inp, x_out = gcn.in_out_tensors()

x_out

predictions = layers.Dense(units=train_targets.shape[1],
activation="softmax")(x_out)

model = Model(inputs=x_inp, outputs=predictions)
model.compile(
    optimizer=optimizers.Adam(lr=0.01),
    loss=losses.categorical_crossentropy,
```

```
    metrics=["acc"],
)

val_gen = generator.flow(val_subjects.index, val_targets)

from tensorflow.keras.callbacks import EarlyStopping

es_callback = EarlyStopping(monitor="val_acc",
patience=50,
restore_best_weights=True)

history = model.fit(
    train_gen,
    epochs=30,
    validation_data=val_gen,
    verbose=2,
    shuffle=False,  # Tem que ser falso, pois ''embaralhar''
    # os dados implica em ''embaralhar'' o grafo
    callbacks=[es_callback],
)

sg.utils.plot_history(history)

test_gen = generator.flow(test_subjects.index, test_targets)

test_metrics = model.evaluate(test_gen)
print("\nTest Set Metrics:")
for name, val in zip(model.metrics_names, test_metrics):
    print("\t{}: {:0.4f}".format(name, val))

all_nodes = node_subjects.index
all_gen = generator.flow(all_nodes)
all_predictions = model.predict(all_gen)
```

```
node_predictions = target_encoding.inverse_transform(all_predictions.squeeze())

df = pd.DataFrame({"Predicted": node_predictions, "True": node_subjects})
df.head(20)

embedding_model = Model(inputs=x_inp, outputs=x_out)

emb = embedding_model.predict(all_gen)
emb.shape

from sklearn.decomposition import PCA
from sklearn.manifold import TSNE

transform = TSNE

X = emb.squeeze(0)
X.shape

trans = transform(n_components=2)
X_reduced = trans.fit_transform(X)
X_reduced.shape

fig, ax = plt.subplots(figsize=(7, 7))
ax.scatter(
    X_reduced[:, 0],
    X_reduced[:, 1],
    c=node_subjects.astype("category").cat.codes,
    cmap="jet",
    alpha=0.7,
)
ax.set(
    aspect="equal",
```

```
    xlabel="$X_1$",
    ylabel="$X_2$",
    title=f"{transform.__name__}",
)

######## TESTE PARA SIGMOIDE ##########

gcn = GCN(
    layer_sizes=[16, 16], activations=["sigmoid", "sigmoid"],
    generator=generator,
    dropout=0.5
)

x_inp, x_out = gcn.in_out_tensors()

x_out

predictions = layers.Dense(units=train_targets.shape[1],
activation="softmax")(x_out)

model = Model(inputs=x_inp, outputs=predictions)
model.compile(
    optimizer=optimizers.Adam(lr=0.01),
    loss=losses.categorical_crossentropy,
    metrics=["acc"],
)

val_gen = generator.flow(val_subjects.index, val_targets)

from tensorflow.keras.callbacks import EarlyStopping

es_callback = EarlyStopping(monitor="val_acc",
patience=50,
```

```
restore_best_weights=True)

history = model.fit(
    train_gen,
    epochs=30,
    validation_data=val_gen,
    verbose=2,
    shuffle=False,  # Tem que ser falso, pois ''embaralhar'' os
    # dados implica em ''embaralhar'' o grafo
    callbacks=[es_callback],
)

sg.utils.plot_history(history)

test_gen = generator.flow(test_subjects.index, test_targets)

test_metrics = model.evaluate(test_gen)
print("\nTest Set Metrics:")
for name, val in zip(model.metrics_names, test_metrics):
    print("\t{}: {:0.4f}".format(name, val))

all_nodes = node_subjects.index
all_gen = generator.flow(all_nodes)
all_predictions = model.predict(all_gen)

node_predictions = target_encoding.inverse_transform(all_predictions.squeeze())

df = pd.DataFrame({"Predicted": node_predictions, "True": node_subjects})
df.head(20)

embedding_model = Model(inputs=x_inp, outputs=x_out)

emb = embedding_model.predict(all_gen)
```

```
emb.shape

from sklearn.decomposition import PCA
from sklearn.manifold import TSNE

transform = TSNE

X = emb.squeeze(0)
X.shape

trans = transform(n_components=2)
X_reduced = trans.fit_transform(X)
X_reduced.shape

fig, ax = plt.subplots(figsize=(7, 7))
ax.scatter(
    X_reduced[:, 0],
    X_reduced[:, 1],
    c=node_subjects.astype("category").cat.codes,
    cmap="jet",
    alpha=0.7,
)
ax.set(
    aspect="equal",
    xlabel="$X_1$",
    ylabel="$X_2$",
    title=f"{transform.__name__}",
)
```