UNIVERSIDADE FEDERAL DO CEARÁ

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ANTÔNIO MICHAEL FARIAS SOARES

IMPLEMENTAÇÃO E AVALIAÇÃO DE BANCO DE DADOS CHAVE-VALOR COM APRENDIZAGEM DE ÍNDICE

CRATEÚS

2022

ANTÔNIO MICHAEL FARIAS SOARES

IMPLEMENTAÇÃO E AVALIAÇÃO DE BANCO DE DADOS CHAVE-VALOR COM APRENDIZAGEM DE ÍNDICE

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Orientador: Prof. Me. Lívio Antônio Melo Freire

CRATEÚS

2022

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S652i Soares, Antônio Michael Farias.
Implementação E Avaliação de Banco de Dados Chave-Valor Com Aprendizagem de Índice / Antônio Michael Farias Soares. – 2022.
70 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Crateús, Curso de Ciência da Computação, Crateús, 2022.
Orientação: Prof. Me. Lívio Antônio Melo Freire.

1. Tabela Hash. 2. Função Hash. 3. Aprendizagem Profunda. 4. Aprendizagem de Índices. I. Título.
CDD 004

ANTÔNIO MICHAEL FARIAS SOARES

IMPLEMENTAÇÃO E AVALIAÇÃO DE BANCO DE DADOS CHAVE-VALOR COM APRENDIZAGEM DE ÍNDICE

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Aprovada em:

BANCA EXAMINADORA

Prof. Me. Lívio Antônio Melo Freire (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. José Wellington Franco da Silva
Universidade Federal do Ceará (UFC)

Prof. Me. Lisieux Marie Marinho dos Santos Andrade
Universidade Federal do Ceará (UFC)

À minha família, por sua capacidade de acreditar em mim e investir em mim. Mãe, seu cuidado e dedicação foi que deram, em alguns momentos, a esperança para seguir. Pai, suas experiências me fizeram ter outra visão de mundo.

AGRADECIMENTOS

Primeiramente a Deus, por ajudar em momentos difíceis e mim dar forças todos os dias para querer tornar meus sonhos possíveis.

Ao Prof. Me. Lívio Antônio Melo Freire por me orientar no trabalho de conclusão de curso.

Ao Prof. Me. Bruno de Castro Honorato Silva por me orientar no Programa de Iniciação a Docência (PID).

Ao Prof. Dr. Rennan Ferreira Dantas por seus conselhos e conversas em momentos difíceis.

À Prof. Me. Lisieux Marie Marinho dos Santos Andrade por me orientar na disciplina de Projeto de Pesquisa Científica e Tecnológica.

Aos meus pais, que nos momentos de minha ausência dedicados ao estudo tiveram a consciência de sua importância.

Agradeço a todos os professores por me proporcionar o conhecimento não apenas racional, mas a manifestação do caráter e afetividade da educação no processo de formação profissional, por tanto que se dedicaram a mim, não somente por terem me ensinado, mas por terem me feito aprender.

A Universidade Federal do Ceará, pela oportunidade de ingressar e cursar a graduação tão perto de casa.

A todos que contribuíram direto e indiretamente na minha formação acadêmica.

“Se cheguei até aqui foi porque me apoiei no ombro de gigantes.”

(Isaac Newton)

RESUMO

As estruturas de dados estão presentes em várias aplicação definindo os métodos de acesso, a organização e como os conjuntos de dados são processados. A Tabela Hash é um tipo de estrutura especial e associativa que permite a realização de buscas eficientes. O artifício que ela utiliza em seu funcionamento é conhecido por função de *hash* ou de espalhamento. Uma dificuldade da Tabela Hash é que, a depender da distribuição dos dados, a modelagem da função de espalhamento pode ter consequências no desempenho da estrutura. Além disso, para conseguir desempenho aceitável, essas estruturas costumam reservar quantidade de memória algumas vezes maior que o tamanho do conjunto de dados. Atualmente, modelos mais sofisticados de Aprendizagem Profunda (AP), que utilizam Redes Neurais Artificiais, conseguem extrair padrões complexos e aprender a distribuição de dados de acordo com suas características, utilizando quantidade razoável de memória. Dessa forma, o presente trabalho tem como objetivo analisar o desempenho da Tabela Hash tradicional com a abordagem de Aprendizagem de Índices, que utiliza AP para aprender a distribuição dos dados referentes ao índice de um Banco de Dados (BD). Como contribuição, realizou-se experimentos computacionais utilizando conjuntos de dados de Strings, no contexto de BD Chave-Valor em memória, implementados por meio de Tabelas Hash. Assim, com este trabalho, avaliou-se a competitividade do modelo de Aprendizagem de Índice sobre outras implementações de Tabela Hash.

Palavras-chave: Tabela Hash. Função Hash. Aprendizagem Profunda. Aprendizagem de Índices.

ABSTRACT

Data structures are present in many applications defining the access methods, the organization and how datasets are processed. Hash Table is a type of structure special and associative search that allows efficient searches to be carried out. what does she wear in its functioning is known as the hash or spread function. A difficulty of The Hash Table is that, depending on the distribution of the data, the modeling of the distribution function may have consequences on the performance of the structure. Furthermore, to achieve performance tolerance, some structures tend to reserve an amount of memory times greater than the size of the dataset. Currently, more adjusted models of Deep Learning (DL), which use Artificial Neural Networks, can extract complex patterns and learn the distribution of data according to its characteristics, using quantity reasonable memory. Thus, the present work aims to analyze the performance of the traditional Hash Table with the Learning Index approach, which uses DL to learn the distribution of reference data to the index of a Database (DB). How contribution, computational experiments were carried out with string data sets, in the context of Key-Value DB in memory, implementing through Hash Tables. So, With this work, the competitiveness of the Learning model of over other implementations of the Hash Table.

Keywords: Hash Table. Hash Funtion. Deep Learning. Index Learning.

LISTA DE FIGURAS

Figura 1 – Exemplo do uso da função de *hash h*. As chaves k_2 e k_5 colidem. 17
Figura 2 – Indexação padrão por Hashing com Encadeamento Separado 19
Figura 3 – Indexação padrão por Hashing com Sondagem Linear 22
Figura 4 – Exemplo de inserção do Cuckoo Hashing 24
Figura 5 – Modelo de um neurônio real 28
Figura 6 – Modelo de um neurônio artificial 28
Figura 7 – Função de limiar 30
Figura 8 – Função de linear por partes 31
Figura 9 – Função Sigmoide 31
Figura 10 – Arquitetura de um perceptron de múltiplas camadas com duas camadas ocultas 33
Figura 11 – i, f, e o são respectivamente o portão de entrada, esquecimento e saída. c e $\tilde{c}$ denotam a célula de memória e o novo conteúdo da célula de memória. . . . 35
Figura 12 – r e z são respectivamente os portões de reset e atualização, h e $\tilde{h}$ são a ativação e a ativação do candidato. 36
Figura 13 – Ilustração da descida do *gradiente* 39
Figura 14 – Estágios de modelos 41
Figura 15 – Tabela Hash tradicional vs Hash aprendido 42
Figura 16 – Índice aprendido vs B-Tree 44
Figura 17 – Dataset de String: Aprendizagem de índice vs B-Tree 44
Figura 18 – Redução de conflitos 45
Figura 19 – Filtro Bloom aprendido usando RNN é representada por W. 46
Figura 20 – Desempenho medido de acordo com número de palavras desconhecidas . . 48
Figura 21 – Palavras vizinhas mais próximas do bigram(sequência de duas palavras) . . 49
Figura 22 – Representação da entrada codificada para a rede neural 51
Figura 23 – Representação das camadas da RNR e sua saída 52
Figura 24 – Representação da inferência de uma chave e seu armazenamento 53
Figura 25 – Gráfico barras empilhadas considerando o consumo de memória 56
Figura 26 – Gráfico de barras considerando as colisões em cada estrutura 56
Figura 27 – Gráfico de linha considerando o tempo de busca 57
Figura 28 – Gráfico de linha considerando o desvio padrão 58
Figura 29 – Gráfico de linha considerando o desvio padrão para cima 58

Figura 30 – Gráfico de linha considerando o desvio padrão para baixo 59
Figura 31 – Gráfico de barras considerando as colisões em cada estrutura 59
Figura 32 – Gráfico de linha considerando o tempo de busca 60
Figura 33 – Gráfico de linha considerando o desvio padrão 61
Figura 34 – Gráfico de linha considerando o desvio padrão para cima 61
Figura 35 – Gráfico de linha considerando o desvio padrão para baixo 62
Figura 36 – Gráfico de barras considerando as colisões em cada estrutura 63
Figura 37 – Gráfico de linha considerando o tempo de busca 64
Figura 38 – Gráfico de linha considerando o desvio padrão 64
Figura 39 – Gráfico de linha considerando o desvio padrão para cima 65
Figura 40 – Gráfico de linha considerando o desvio padrão para baixo 65

LISTA DE TABELAS

Tabela 1 – Comparativo geral do melhor desempenho 66

LISTA DE ABREVIATURAS E SIGLAS

AP	Aprendizagem Profunda
BD	Banco de Dados
CDF	Cumulative Distribution Function
DB	Database
DL	Deep Learning
DSSM	Deep Semantic Similarity Model ou Deep Structured Semantic Model
GRU	Gated Recurrent Unit
LIF	Learning Index Framework
LSTM	Long Short Term Memory
PNL	*Processing Natural Language*
RMI	Recursive Model Indexes
RNA	Rede Neural Artificial
RNR	Rede Neural Recorrente
SGBDs	Sistemas de Gerenciamento de Banco de Dados

SUMÁRIO

1 INTRODUÇÃO 14
1.1 Objetivo Geral 15
1.2 Objetivos Específicos 15
2 FUNDAMENTAÇÃO TEÓRICA 16
2.1 Banco de Dados Chave-Valor 16
2.1.1 Tabelas Hash 16
2.1.2 Funções de Hash 18
2.1.3 Encadeamento Separado 19
2.1.4 Sondagem Linear 21
2.1.5 Cuckoo Hashing 23
2.2 Redes Neurais Artificiais 27
2.2.1 Neurônio Artificial 27
2.2.2 Funções de Ativação 29
2.2.3 Redes Neurais Multicamadas 32
2.2.4 Redes Neurais Recorrentes 33
2.2.5 Treinamento e Avaliação 37
2.2.6 Aprendizagem de índice 40
3 TRABALHOS RELACIONADOS 43
3.1 The Case for Learned Index Structures 43
3.2 FASTER: An Embedded Concurrent Key-Value Store for State Management 46
3.3 CHARAGRAM: Embedding Words and Sentences via Character n-grams 47
4 METODOLOGIA 50
4.1 Modelo de Aprendizagem e Métricas de Avaliação da Tabela Hash . . . 50
4.2 Cenário 54
5 RESULTADOS 55
5.1 Resultados dos testes com dados em memória 55
5.2 Resultados dos testes com distribuição normal 56
5.3 Resultados dos testes com distribuição uniforme 59
5.4 Resultados dos testes com distribuição de frequência *zipf* 62

5.5 **Resumo dos resultados** 66
6 **CONCLUSÕES E TRABALHOS FUTUROS** 67
REFERÊNCIAS 68

1 INTRODUÇÃO

Um dos temas fundamentais no campo da Ciência da Computação são as Estrutura de Dados, que consistem em algoritmos que determinam a maneira como os dados são organizados, visando otimizar o custo de operações realizadas sobre eles (BLACK, 2004). Há, atualmente, grande variedade de implementações de estruturas de dados, dificultando a escolha do algoritmo mais adequado para a solução de um problema. Na literatura, destacam-se as estruturas: Árvores Rubro-Negras, estrutura eficiente para indexação de dados em memória, que garante a execução eficiente de operações que envolvem dados ordenados; Árvore B, estrutura bastante utilizada para indexar arquivos, permitindo a manipulação de grandes volumes de dados; Tabelas Hash, que mapeiam chaves para certa posição de um registro, possibilitando operações de busca eficientes (SZWARCFITER; MARKENZON, 1994).

Os Índices são Estruturas de Dados utilizadas para referenciar registros, possibilitando consultas otimizadas sobre grandes volumes de dados. Em Sistemas de Gerenciamento de Banco de Dados (SGBDs), os índices são usados para localizar rapidamente os dados, não havendo assim a busca por "força bruta" dos elementos desejados. Essa busca por eficiência se dá pela grande quantidade de dados existentes e que precisam ser gerenciados da melhor forma, o que também implica na evolução de outros ramos da computação (SIQUEIRA, 2019). Além disso, as estrutura tradicionais não tiram proveito dos padrões dos dados e não assumem nada sobre suas distribuições por serem estruturas de uso geral (KRASKA *et al.*, 2018).

Entretanto, outra área que está sendo utilizada nesse contexto é a Aprendizagem Profunda (AP), ramo da Inteligencia Artificial e subárea da Aprendizagem de Máquina que permite a construção de modelos abstratos com o uso de Redes Neurais Artificiais (RNAs).

Nesse sentido, Kraska *et al.* (2018) propõe a abordagem cuja ideia é aplicar AP como substituição às Estruturas de Dados tradicionais, conhecida por "Aprendizagem de Índice". Em outras palavras, RNAs são utilizadas para aprender a distribuição dos dados a serem indexados, tendo como resultado modelos capazes de determinar a posição de chaves de pesquisa dentro de uma sequência, possibilitando a predição eficiente da posição dos registros. Como exemplo, tem-se o SageDB, banco de dados cuja estrutura é radicalmente diferente dos outros sistemas. Na sua implementação, utiliza-se modelos de Aprendizagem de Máquina com propósito de aprender a estrutura de dados adequada para o índice, métodos ótimos de acesso e planos de consulta (KRASKA *et al.*, 2019).

Neste trabalho, avalia-se a Aprendizagem de Índice e a Tabela Hash empregados na

implementação de Banco da Dados Chave-Valor em memória. Assim, busca-se implementações eficientes capazes apreender a distribuição dos dados, de modo a diminuir as colisões, mas que também sejam eficientes em relação ao consumo de memória e ao tempo de pesquisa computacional.

1.1 Objetivo Geral

Avaliar a abordagem de Aprendizagem de Índice para dados do tipo "cadeia de caractere" e comparar com Estruturas de Dados tradicionais de Tabela Hash.

1.2 Objetivos Específicos

- Avaliar o desempenho de diferentes tipos de Tabela Hash para dados do tipo "cadeia de caractere";
- Estabelecer estratégia capaz de codificar "cadeias de caracteres";
- Definir e implementar arquitetura de RNAs capazes de aprender a distribuição de "cadeias de caracteres" codificadas;
- Realizar experimentos computacionais que permitam analisar as Estruturas investigadas sob o ponto de vista de eficiência no uso de memória e tempo computacional para pesquisa sobre grandes volumes de dados do tipo "cadeias de caracteres";

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, apresenta-se a fundamentação teórica necessária sobre Banco de Dados Chave-Valor. Na Seção 2.1 destaca-se o que é uma Tabela Hash, Funções de Hash e resolução de conflitos por encadeamento separado e sondagem linear. Na Seção 2.2 destaca-se os principais conceitos de Redes Neurais Artificiais, definindo neurônio artificial, função de ativação, redes neurais multicamadas, treinamento, avaliação e codificação de "cadeia de caracteres". Ao final do capítulo o leitor será capaz de entender o que é Aprendizagem de Índice.

2.1 Banco de Dados Chave-Valor

Um Banco de Dados Chave-Valor armazena os dados com base no método de acesso formado pelo par chave-valor em que uma chave consiste de um identificador único para se referir a um determinado valor. A chave e o valor pode ser qualquer tipo de objeto. Assim, há diversas estruturas deste tipo que se diferenciam pela maneira como o par chave-valor são tratados e como os dados são armazenados.

2.1.1 Tabelas Hash

Tabela Hash consiste em uma estrutura de dados associativa de chave e valor utilizadas para buscas eficientes. Para tal finalidade, emprega-se geralmente *arrays* e uma *Função de Espalhamento*, também conhecida por *função hash*. A Função de Espalhamento calcula qual o índice do vetor a chave corresponde.

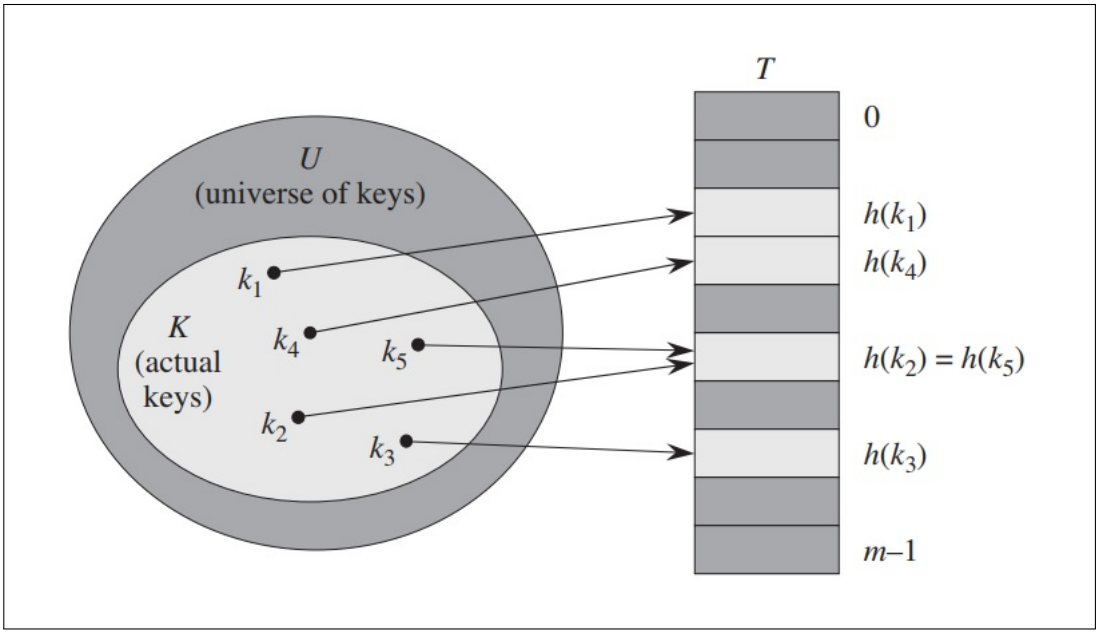
Para tratar previamente desse assunto, suponha que exista uma Tabela T sequencial e que possua dimensão m, assim as posições acessíveis são $[0...m-1]$. Agora, suponha que exista um conjunto de n chaves, com $n = m$. Além disso, os valores das chaves sejam respectivamente $0...m-1$. Logo, pode ser usado o próprio valor da chave como índice da tabela. Por exemplo, uma chave k é armazenada na posição k da tabela. Denomina-se a essa técnica como *endereçamento direto* (SZWARCFITER; MARKENZON, 1994).

Isso é bastante vantajoso em uma aplicação real, pois se existisse um banco de dados com pessoas cadastradas, tal que cada uma tem um número de identificação e que o mesmo corresponde ao índice do banco, então para uma pesquisa a técnica seria perfeita. Porém, segundo Cormen *et al.* (2009), quando se tem um universo U muito grande, o armazenamento de uma tabela T de tamanho $|U|$ pode ser impraticável, ou mesmo impossível, em virtude da limitação

de memória de um computador típico. Ou seja, essa é uma dificuldade do armazenamento direto, pois pode existir um conjunto k de chaves pequeno em relação a U, o que torna o espaço de armazenamento alocado para T desperdiçado.

Assim, pelo exposto, a diferença do endereçamento direto para uma Tabela Hash é que, como descrito acima, um elemento de Chave K é mapeado para a posição K, conforme ilustrado na Figura 1, enquanto com o uso da Tabela Hash existe uma função de hash, que mapeia a chave k para a posição $h(k)$.

Figura 1 – Exemplo do uso da função de *hash h*. As chaves k_2 e k_5 colidem.



Fonte: (CORMEN *et al.*, 2009)

De acordo com Cormen *et al.* (2009), a função de hash ou espalhamento tem a finalidade de reduzir o intervalo de índices de arranjos que precisam ser tratados. Em vez $|U|$ valores, precisará de manipular apenas m valores. Logo, os requisitos de armazenamento são reduzidos de modo correspondente. Vale destacar que o uso dessa técnica possui um detalhe a se observar: duas ou mais chaves podem ter o mesmo valor hash, com endereço na mesma posição. Para essa situação, dá-se o nome de *colisão* .

Embora a função de hash h seja determinística, já que para toda chave k irá gerar sempre a mesma saída, é impossível evitar a ocorrência das colisões para um universo $|U| > m$, já que duas ou mais chaves poderão ter a mesma saída de h. Portanto, para tentar diminuir esse efeito, existem técnicas que também contribuem para deixar a função capaz de gerar espalhamento melhor e com menos colisão. Na seção a seguir, a função de hash é melhor detalhada.

2.1.2 Funções de Hash

Uma função de *hash* ou de dispersão transforma uma chave em um índice da Tabela Hash. Além disso, é considerada de boa qualidade quando diminui as chances de gerar hash na mesma posição. A maioria das funções de hash supõe que o universo de chaves são do conjunto dos números naturais. Assim, se a chave não pertence a esse conjunto, então deve-se encontrar uma maneira de representar como números naturais (CORMEN *et al.*, 2009).

Em Szwarcfiter e Markenzon (1994), considera-se que uma função de dispersão deve satisfazer a seguintes condições:

- produzir um número baixo de colisões;
- ser facilmente computável;
- ser uniforme.

Em implementações reais, pode-se ter situações em que a chaves se adaptam muito bem a padrões conhecido, como em uma empresa em que algumas informações são compostas por mês e o ano do pedido. Ou que o padrão é bastante difícil de se conhecer, como por exemplo a tabela de símbolos de um compilador onde não é possível conhecer quais identificadores serão utilizados. Outra coisa que também pode influenciar na computação do hash é se a tabela é armazenada em disco ou na memória principal. Por isso, uma função de hash h ideal dever ser tal que todos os compartimentos tenham a mesma probabilidade de serem escolhidos. Uma função que satisfaça essa condição é chamada de *uniforme* (SZWARCFITER; MARKENZON, 1994).

Por isso, existem métodos que podem ser utilizados para criar funções de hash. Segundo Cormen *et al.* (2009), no *método de divisão*, mapeia-se uma chave k para uma tabela de M posições, calculando o resto de k dividido por M, assim a função de hash ficaria $h(k) = k \ mod \ M$.

Em Sedgewick e Wayne (2011), sugere-se outra maneira de como construir uma função de hash. Para os números reais, com chaves entre 0 e 1, multiplica o M que é o tamanho da tabela T, e arrendonda para o próximo valor inteiro que estará entre 0 e $M-1$. Embora seja bastante fácil, essa abordagem é defeituosa, pois dá mais pesos aos bits mais significativos e os bits menos significativos não desempenham nenhum papel.

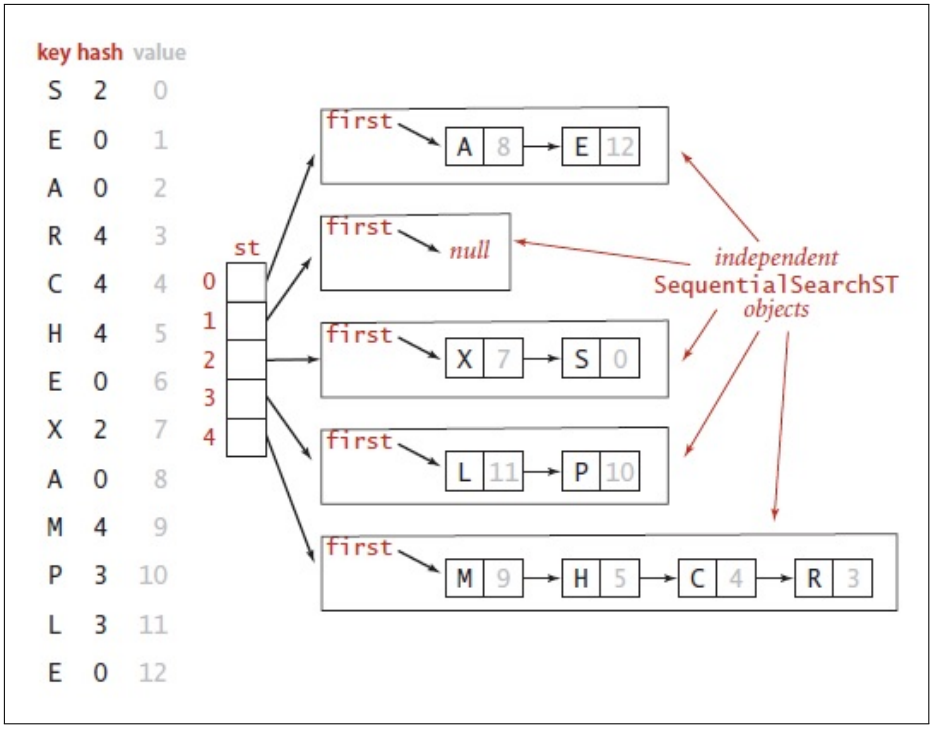
A seguir, será apresentada a técnica mais simples que tenta minimizar o problema de colisão da chaves. A colisão das chaves ocorre quando a função hash gera a mesma saída para diferentes chaves. Então, diferentes implementações de Tabela Hash tratam de como fazer o armazenamento do valor das chaves, que possuem o mesmo valor hash, como também manter

a eficiência da estrutura.

2.1.3 Encadeamento Separado

Nessa técnica de resolução de colisão, cada índice do array vai apontar para uma lista encadeada, que vai vinculando as chaves com mesmo valor hash. Logo, para encontrar determinado elemento x de um array T bastaria fazer a operação $T[h(x)]$, em que h é função de hash, assim o índice é encontrado. Em seguida, basta percorre a lista e retornar o valor buscado, como mostra a Figura 2. Vale observar que o tempo de execução no pior caso para inserção é $O(1)$ na lista, pois a chave é inserida no início da estrutura (CORMEN *et al.*, 2009).

Figura 2 – Indexação padrão por Hashing com Encadeamento Separado



Fonte: (SEDGEWICK; WAYNE, 2011)

Além disso, a técnica de encadeamento separado conta com um artifício que ajuda a diminuir as colisões e garante o tamanho das listas dentro de um intervalo constante, definido por (CORMEN *et al.*, 2009) da seguinte forma: "dada uma tabela hash T com m posições que armazena n elementos, define-se fator de carga α para T como o valor $\alpha = \frac{n}{m}$" . Ou seja, seria basicamente o número médio de elementos da lista que se espera ter em cada posição j do array. Neste trabalho, α será analisado quando seu valor for menor, igual ou maior a 1.

Acrescenta-se também que o desempenho da tabela hash depende de como a função h distribui seus elementos entre as m posições, assim supõe-se uma hipótese chamada de *hash*

uniforme simples em que os elementos têm igual probabilidade de efetuar o hash entre qualquer uma das posições. Por outro lado, a pior configuração que a tabela pode ter é quando todas as n chaves efetuam hash em uma mesma posição, isso criaria uma lista bastante grande com todos os elementos e a pesquisa teria no pior caso o custo $O(n)$ mais o tempo para calcular o hash da chave. Os teoremas seguintes demonstram o tempo de pesquisa quando se considera o número esperado de elementos por posição.

Teorema 2.1.1. *(CORMEN et al., 2009) Em uma Tabela Hash, na qual as colisões são resolvidas por encadeamento, uma pesquisa malsucedida demora tempo esperado* $\theta(1+\alpha)$*, sob hipótese de hash uniforme simples.*

Demonstração. Sob a hipótese de hash uniforme simples, qualquer chave k ainda não armazenada na tabela tem igual probabilidade de efetuar o hash para qualquer das m posições. O tempo médio para pesquisa sem sucesso para uma chave k é, portanto, o tempo esperado para pesquisar até o fim da lista $T[h(k)]$, que tem o comprimento esperado $E[n_{h(k)}] = \alpha$. Assim, o número esperado de elementos examinados em uma pesquisa malsucedida é α, e o tempo necessário (incluindo o tempo para se calcular $h(k)$ é $\theta(1+\alpha)$.

□

Teorema 2.1.2. *(CORMEN et al., 2009) Em uma tabela hash na qual as colisões são resolvidas por encadeamento, uma pesquisa bem-sucedida demora tempo* $\theta(1+\alpha)$*, na média, sob hipótese de hash uniforme simples.*

Demonstração. Supondo que o elemento que está sendo pesquisado tem igual probabilidade de ser qualquer dos elementos n armazenados na tabela. O número de elementos examinados durante uma pesquisa bem-sucedida para um elemento x é uma unidade maior que o número de elementos que aparecem antes de x na lista de x. Os elementos antes de x foram todos inseridos após x ser inserido, porque novos elementos são colocados no início da lista. Para encontrar o número de elementos esperado de elemento examinados, toma-se a média, sobre os n elementos x na tabela, de 1 mais o número esperado de elementos adicionados à lista de x depois que x foi adicionado à lista. Seja x_i o *i-ésimo* elemento inserido na tabela, para $i = 1,2,...,n$, e seja $K_i = chave[x_i]$. Para chaves k_i e k_j, definimos a variável indicadora aleatória $X_{ij} = Ih(k_i) = h(k_j)$. Sob a hipótese de hash uniforme simples, temos $Pr\{h(K_i) = h(K_j)\}$ e então, pelo Lema 5.1, $E[x_{ij}] = \frac{1}{m}$. Desse modo, número esperado de elementos examinados em uma pesquisa bem-sucedida é

$$
\begin{aligned}
&E[\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}X_{ij})] \\
&\qquad = \frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}E[X_{ij}]) \\
&\qquad = \frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}\frac{1}{m}) \\
&\qquad = 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)) \\
&\qquad = 1+\frac{1}{nm}(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i) \\
&\qquad = 1+\frac{1}{nm}(n^2-\tfrac{n(n+1))}{2}) \\
&\qquad = 1+\frac{n-1}{2m} \\
&\qquad = 1+\frac{a}{2}-\frac{a}{2n}.
\end{aligned}
$$

Desse modo, o tempo total para uma pesquisa bem-sucedida(incluído o tempo para calcular a função de hash) é $\theta\left(2+\frac{\alpha}{2}-\frac{\alpha}{2n}\right)=\theta\left(1+\alpha\right)$.

□

Por esse motivo, em uma implementação de encadeamento separado (SEDGEWICK; WAYNE, 2011) diz que o objetivo é escolher o tamanho da tabela T suficientemente pequeno para não desperdiçar uma enorme área de memória contígua vazia, mas suficientemente grande para não perder tempo procurando em longas cadeias.

A seguir, é abordado outra técnica que tenta fazer o melhor uso de memória em relação ao encadeamento separado.

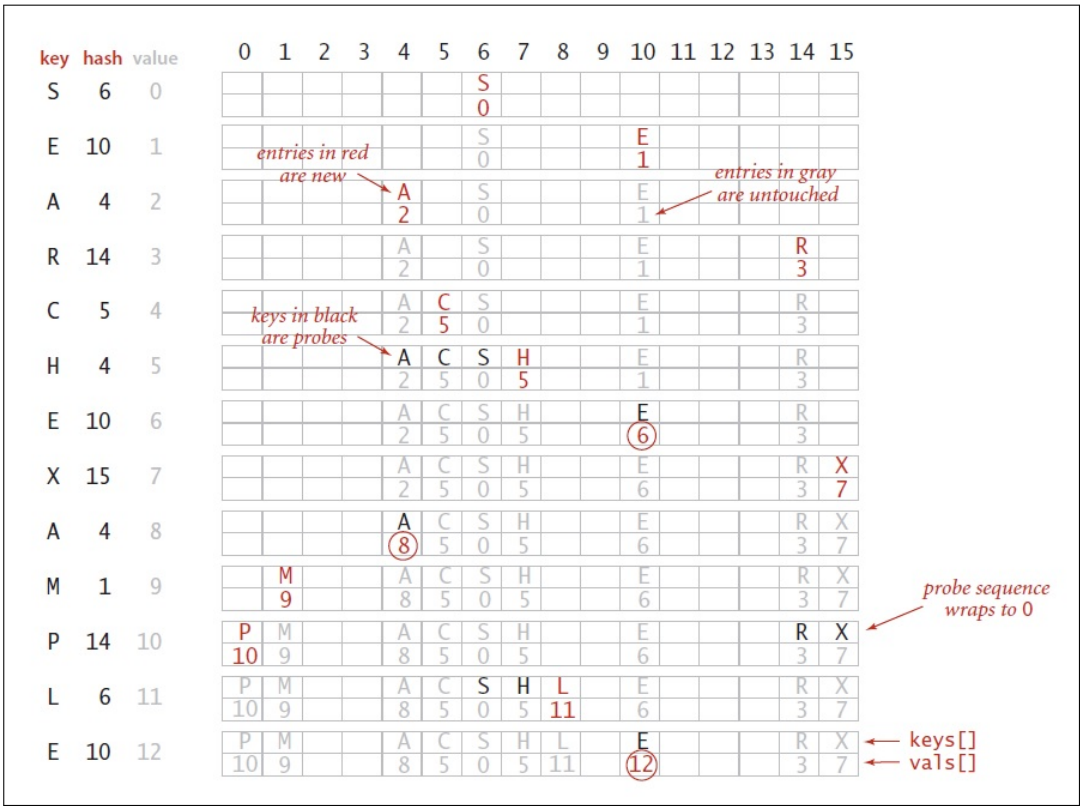
2.1.4 Sondagem Linear

Nessa abordagem não se dispõe de uma lista encadeada em cada posição, o que ocorre é que os elementos são armazenados normalmente na tabela conforme Figura 3. Ou seja, cada índice tem um elemento ou valor Nulo caso a posição seja vazia. Essa é uma técnica simples que faz parte do endereçamento aberto. Pois, ao fazer uma busca por um elemento, aplica-se a função de hash, que dará uma posição. Caso o elemento desejado não esteja nela, então é analisado o próximo índice, até que o elemento seja encontrado ou que, após percorrer todas as

posições possíveis de modo circular, não ache o elemento. Para tal, a função de espalhamento é descrita dessa forma

$$h(x,k) = \left(h'(x) + k\right) mod\ m, \qquad 0 \leq k \leq m-1 \tag{2.1}$$

Figura 3 – Indexação padrão por Hashing com Sondagem Linear



Fonte: (SEDGEWICK; WAYNE, 2011)

A vantagem de utilizar a sondagem linear é que ela não usa referência para memórias, o que pode dar uma quantidade de posição maior para a tabela com a mesma quantidade de memória. Para inserir um novo elemento, examina-se a tabela a partir do índice gerado pela função de hash, conforme 2.8, até encontrar uma posição vazia. Porém, a tabela hash pode ficar totalmente cheia a ponto de não ser possível nenhuma inserção de novo elemento. Além disso, em (SZWARCFITER; MARKENZON, 1994), destaca-se que esse método tende a produzir longos trechos consecutivos de memória ocupados, o que se denomina *agrupamento primário* que pode implicar no aumento do custo de busca por uma chave.

Também, na literatura, há um problema real que faz analogia com a sondagem linear, o Problema do Estacionamento de Knuth. Nele os automóveis procuram vaga de estacionamento em uma rua que já tem carros estacionado. Ou seja, carros chegam em ruas de mão única com M vagas de estacionamento. Cada um deseja estacionar em um espaço aleatório i: se o espaço está

ocupado, tenta-se os espaços $i+1$, $i+2$, etc. Com isso, surge um questionamento, em média, quantas vezes um carro deve tentar até achar uma vaga?

Proposição 2.1.3. *Sob a hipótese de espalhamento uniforme, a média de sondagens em uma tabela de espalhamento com sondagem linear com tamanho M e com* $N = \alpha * M$ *chaves é:* $\sim \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$ *para buscas bem−sucedida e* $\sim \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$ *para buscas ou inserções mal−sucedida.*

Exemplo 1. *Quando* $\alpha = 0.5$*, tem-se aproximadamente 1.5 sondagens por busca bem-sucedida e aproximadamente 2.5 sondagens por busca malsucedida.*

Exemplo 2. *Quando* $\alpha = 0.25$*, tem-se aproximadamente 1.16 sondagens por busca bem-sucedida e aproximadamente 1.39 por busca malsucedida.*

A remoção de um par chave-valor é considerada difícil, porque não se pode deletar diretamente a chave ou simplesmente coloca-la como vazio, ou seja, NULL na posição. Por exemplo, deletando uma chave na posição *i*, então quando fosse fazer uma busca em que várias chaves estão com valor da função de hash na posição i, a busca estaria totalmente comprometida pois a primeira chave tem valor NULL. Logo, uma solução mas não tão boa conhecida como *deleção preguisosa*, é colocar apenas no valor da chave de NULL e que futuramente seria adiada e tratada a eliminação de fato das chaves com valores NULL (SEDGEWICK; WAYNE, 2011).

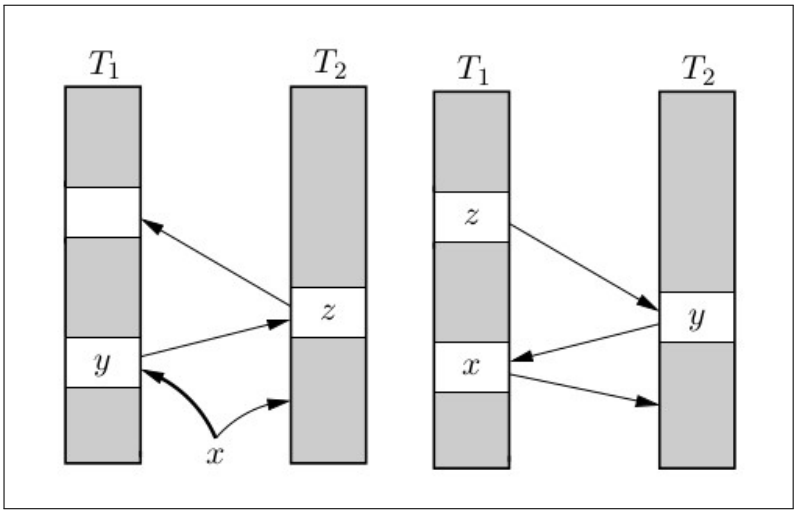
Dessa forma, toda vez que uma chave for removida então é verificado o fator de carga $\alpha = \frac{N}{M}$. Logo, quando o valor de α for maior ou igual a $\frac{1}{2}$ então o tamanho do array T é duplicado. Por outro lado, quando o valor de α for menor ou igual a $\frac{1}{8}$ então o tamanho do array T é reduzido pela metade (SEDGEWICK; WAYNE, 2011). Além disso, a cada operação dessa de redimensionamento as chaves precisarão ser reespalhadas, ou seja, inseridas novamente no novo array. Assim, como alternativa à essa técnica e para prevenir a deleção preguiçosa, é apresentado a seguir uma outra abordagem de Tabela Hash.

2.1.5 Cuckoo Hashing

De acordo com (PAGH; RODLER, 2004), define-se Cuckoo Hashing como uma dinamização de um dicionário que usa duas tabelas de hash T_1 e T_2 tal que cada uma consiste de r elementos, e duas funções hash $h_1, h_2 : U \to \{0,...,r-1\}$, assim, dada uma chave x, então ela será armazenada em $h_1(x)$ de T_1 ou em $h_2(x)$ de T_2, mas nunca nas duas tabelas ao mesmo tempo.

Para inserir um elemento x_1 é simples, basta verificar se ele não está lá, usando a primeira função hash h_1, ou seja, se essa posição estiver vazia então o item pode ser colocado. Agora supondo que deseja-se inserir um segundo elemento x_2, se o valor de $h_1(x_2)$ for igual a uma posição que já esteja ocupada, por exemplo por x_1, então desloca-se x_1 para uma nova posição na segunda tabela usando a função de hash h_2 em seguida colocando x_2 em $h_1(x_2)$. Pode-se observar que, se o valor de $h_1(x_2)$ for igual a $h_1(x_1)$, então poderia ser usado o $h_2(x_2)$ para o elemento ser alocado na segunda tabela, o problema é que nessa posição também poderia acontecer de já ter um elemento lá (WEISS, 2012) conforme ilustração da Figura 4.

Figura 4 – Exemplo de inserção do Cuckoo Hashing



Fonte: (PAGH; RODLER, 2004)

Se a inserção de determinado elemento provocar uma sequência de deslocamento de outros elementos então poderá criar um ciclo e impedir a inserção desse novo elemento. Contudo, segundo Weiss (2012) se o fator de carga for inferior a 0.5, uma analise mais complexa mostra que a chance de um ciclo ocorrer é bem pequena sendo quase que improvável que uma inserção bem sucedida exigira mais do que $O(\log N)$ deslocamentos. Para evitar isso, pode-se simplesmente reconstruir as tabelas com novas funções hash após detectar determinado número de deslocamentos. Mais precisamente, a probabilidade de uma inserção exigir uma nova reconstrução com novas funções hash tem o pior caso o custo de $O(1/N^2)$ (PAGH; RODLER, 2004), conforme a análise a seguir.

Segundo Pagh e Rodler (2004), a análise do procedimento de inserção se baseia em três partes:

- Primeiramente é exibido algumas características do comportamento da inserção. Nesse caso, o tipo de inserção mais simples é quando nenhuma célula da tabela

hash é visitada mais de uma vez, ou seja, ele apenas move as chaves para outra tabela sem repetição de células iguais. Mas, se por acaso todas as chaves voltarem a sua célula de origem, então a chave a ser inserida ficará sem lugar, pois existe um loop fechado. Para isso, o procedimento de rehashing é executado.
Suponha que o procedimento de inserção não entre em um ciclo. Então, para qualquer prefixo $x_1, x_2, ..., x_p$ da sequência de chaves sem ninho, deve haver uma subsequência de pelo menos $p/3$ chaves consecutivas sem repetições, começando com uma ocorrência da chave x_1, isto é, a chave que está sendo inserida.

Demonstração. No caso em que o procedimento de inserção nunca retorna uma célula visitada anteriormente, o prefixo em si é uma sequência de p chaves sem ninho distintas começando com x_1. Se $p < i + j$, o primeiro $j - 1 \geq \frac{i+j-1}{2} \geq p/2$ chaves aninhadas formam a sequência desejada. Para $p \geq i + j$ umas das sequências $x_1, ..., x_{j-1}$ e $x_{j+i-1}, ..., x_p$ deve ter comprimento pelo menos $p/3$.

■

- Em seguida, calcula-se a probabilidade de que o procedimento de inserção usa pelo menos t iterações. Para $t > MaxLoop$ (valor que limita a quantidade de iterações para um rehashing) a probabilidade é 0. Caso contrário, a análise é realizada em duas situações. Na primeira situação, $v \leq l$ denota o número de chaves distintas sem ninho. O número de maneiras pelas quais o loop fechado pode ser formado é menor que $v^2 r^{v-1} n^{v-1}$, onde v^2 é valore possíveis para i e j, r^{v-1} escolhas possíveis de células e n^{v-1} as opções possíveis de chaves diferentes para x_1. Desde $v \leq MaxLoop$ as funções hash são (c, v) universais. Isso significa que cada possibilidade ocorre com probabilidade no máximo $c^2 r^{-2v}$ e somando todos os valores possíveis de v, e usando $r/n > 1 + \varepsilon$, obtemos que a probabilidade da primeira situação é no máximo:

$$\sum_{v=3}^{l} v^2 r^{v-1} n^{v-1} c^2 r^{-2v} \leq \frac{c^2}{rn} \sum_{v=3}^{\infty} v^2 (n/r)^v < \frac{13c^2/\varepsilon}{n^2} = O(1/n^2) \tag{2.2}$$

Na segunda situação há uma sequência de chaves sem ninho distintas $b_1, ..., b_v$, $v \geq (2t-1)/3$, de modo que b_1 é a chave a ser inserida e $\beta_1, \beta_2 = (1,2)$ ou $\beta_1, \beta_2 = (2,1)$:

$$h_{\beta_1}(b_1) = h_{\beta_1}(b_2), h_{\beta_2}(b_2) = h_{\beta_2}(b_3), h_{\beta_1}(b_3) = h_{\beta_1}(b_4), ... \tag{2.3}$$

Dado b_1, existem no máximo n^{v-1} sequências possíveis de v chaves distintas e para qualquer sequência e qualquer uma das duas opções de (β_1, β_2),a probabilidade que as equações $b-1$ em (2.3) são limitadas por $cr^{-(v1)}$, desde de que as funções hash tenha sido escolhidas de uma família universal. Daí a probabilidade de que exista qualquer sequência de comprimento v é, portanto, a probabilidade da situação acima, delimitada por:

$$2c^2(n/r)^{v-1} \leq 2c^2(1+\varepsilon)^{-(2t-1)/3+1} \tag{2.4}$$

- Por fim, argumenta-se que o procedimento utiliza a constante amortizada esperada de Tempo. Agora, restringe-se a atenção ao $MaxLoop = O(n)$. De (2.4) segue-se que o número esperado de iterações no loop de inserção é limitado por:

$$\begin{aligned} &1 + \textstyle\sum_{t=2}^{MaxLoop} 2c^2(1+\varepsilon)^{-(2t-1)/3+1} + O(1/n^2) \\ &\leq 1 + O(\tfrac{MaxLoop}{n^2}) + 2c^2 \textstyle\sum_{t=0}^{\infty}((1+\varepsilon)^{-2/3})^t \\ &= \mathrm{O}(1 + 1\tfrac{}{1-(1+\varepsilon)^{-2/3})} \\ &= \mathrm{O}(1+1/\varepsilon) \end{aligned} \tag{2.5}$$

Por fim, é considerado o custo do rehashing, que ocorre se na inserção há loop que é executado por $t = MaxLoop$ iterações. A equação anterior calcula a probabilidade de que um loop fechado ocorra. Agora, definindo $MaxLoop = \lceil 3\log_{1+\varepsilon} n \rceil$ a probabilidade de repetição sem entrar num loop fechado é, por (2.4), no máximo:

$$2c^2(1+\varepsilon)^{-(2MaxLoop-1)/3+1} = O(1/n^2) \tag{2.6}$$

Assim, no total a probabilidade de qualquer inserção causar um rehashing é $O(1/n^2)$. Em particular, as n inserções durante uma repetição de todas as chaves ter sucesso(ou seja, sem causar nova repetição) tem probabilidade $1-O(1/n)$, e o tempo por inserção é O(1),logo o tempo total esperado para inserir todas as chaves é O (n). Como a probabilidade de ter que recomeçar com novas funções de hash é limitado a 1, o tempo total esperado para um rehash é $O(n)$. Assim, para qualquer inserção, o tempo esperado usado para rehashing é $O(1/n)$.

Resumidamente, o tempo esperado para inserção é limitado por uma constante. A pequena probabilidade de repetição implica de fato que também a variação do tempo de inserção

é constante. A deleção também é esperada custo no pior caso $O(1)$, sem contar o custo possível de redimensionamento das tabelas caso haja muitos espaços vazios (PAGH; RODLER, 2004) .

Por fim, ao longo das seções é observado que cada estrutura tenta resolver o problema de outra, mas que todas tratam os dados da mesma forma no sentido de fazer somente o pré processamento sobre estes e apontar sua localização. Contudo, é possível ir além e entender a natureza dos dados permitindo, assim, fazer predições sobre eles. Na seção a seguir é abordado mais sobre o assunto.

2.2 Redes Neurais Artificiais

Uma Rede Neural Artificial (RNA) é uma estrutura complexa ou modelo capaz de realizar operações como cálculos em paralelo, para processamento de dados e representação de conhecimento (GRüBLER, 2018). O processamento ocorre por meios de elementos interligados entre si onde o principal elemento de uma RNA é o neurônio artificial, e junto a esse está presente a função de ativação. Além disso, existem várias arquiteturas que definem a finalidade de uma RNA, bem como avaliar se o modelo está apreendendo ou não. Todos esses conceitos são fundamentais para entender o que é Aprendizagem de Índice, objeto de estudo deste trabalho.

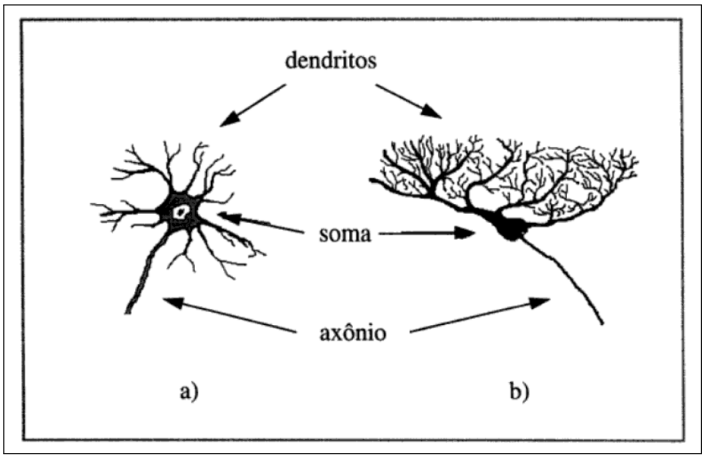
2.2.1 Neurônio Artificial

Segundo Haykin (2007), "um neurônio é uma unidade de processamento de informação que é fundamental para a operação de uma rede neural". O modelo foi proposto originalmente por McCulloch e Pitts em 1943, que de maneira simplificada, implementou o funcionamento biológico de um neurônio como mostra a Figura 5 e também de seus componentes. Nesse caso, os sinais de entrada $x_1, x_2, , x_3, ..., x_n$ que alimenta o modelo com dados, correspondem aos dendritos que recebem impulsos elétricos de outros neurônios. Já a saída y_k corresponde ao axônio. Existe também os estímulos das sinapses, que definem os vários níveis de excitação no neurônio podendo ser muito ou pouco e que é representado pelos pesos sinápticos nesse modelo matemático. A Figura 6 contém a descrição do neurônio artificial.

Neste modelo mostrado na Figura 6, são identificados três elementos básicos:

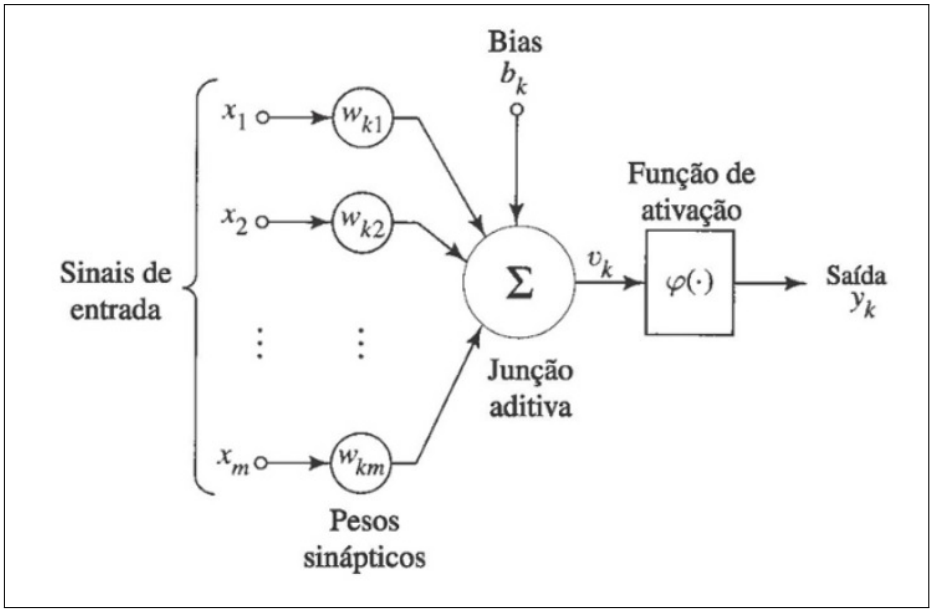
- Pesos sinápticos ou conexões de entrada $w_1, w_2, w_3, ..., w_n$: são valores para ponderar os sinais de cada entrada da rede. Um sinal x_j em uma entrada da sinapse j, conectada ao neurônio k é multiplicado por um peso W_{kj};

Figura 5 – Modelo de um neurônio real



Fonte: (KOVÁCS, 2002)

Figura 6 – Modelo de um neurônio artificial



Fonte: (HAYKIN, 2007)

- Junção aditiva: responsável pela soma de todos os sinais de entrada que foram ponderados pelos pesos sinápticos. Essas somas constitui-se de uma combinação linear.
- Função de ativação: Específica a amplitude de saída de um neurônio e limita o intervalo permissível dessa amplitude a um intervalo unitário finito [0,1] ou [-1, 1].

Além disso, no modelo descrito, tem-se o *bias*, que é incluído no somatório da função de ativação com intuito de aumentar o grau de liberdade da função e ter uma aproximação maior da rede. Acrescenta-se, também, que ele evita que o neurônio apresente uma saída nula mesmo que ainda todas as entradas sejam nulas. Pois, se não houvesse o *bias* e todas as saídas

do neurônio fossem nulas, então a função de ativação seria nula e assim implicaria de o neurônio não conseguir aprender.

Portanto, o neurônio artificial é descrito matematicamente na literatura como o seguinte par de equações:

$$u_k = \sum_{j=1}^{m} w_{kj} x_j \tag{2.7}$$

e

$$y_k = \varphi(v_k + b_k) \tag{2.8}$$

Onde os $x_1, x_2, , x_3, ..., x_n$ são os sinais de entrada; $w_1, w_2, w_3, ..., x_n$ são os pesos sinápticos do neurônio k; v_k é a saída do combinador linear dos sinais de entrada; b_k é o *bias*; y_k é o sinal do neurônio e φ é a função de ativação. Por sua importância, a função de ativação será destacada com mais detalhes na próxima seção.

2.2.2 Funções de Ativação

As funções de ativação são um elemento bastante importante nas redes neurais artificiais. Por exemplo, supondo que em uma classificação de um digito "5" fosse possível alterar os valores dos pesos ou *bias* de um neurônio quando a rede o classificou erroneamente como "6", então com essas alterações poderia fazer com que a rede se comportasse de maneira mais próxima da correta. O problema é que essa alteração pode fazer com que a rede tenha um comportamento totalmente diferente do esperado ao longo da aprendizagem.

Por esse motivo, é introduzido uma função de ativação que permitirá uma pequena mudança nos pesos (ou *bias*) de modo controlado, por exemplo. Ela basicamente decide se o neurônio será ativado ou não, ou seja, verifica se a informação que o neurônio está recebendo é relevante ou se deve ser ignorada.

A função de ativação quando não usada, os pesos e *bias* simplesmente fazem uma transformação linear. Assim, problemas como uma equação linear é fácil de resolver, porém, são limitadas por não conseguir resolverem problemas complexos. Logo, com o uso da função de ativação ocorre a transformação não-linear em que a saída se propaga ao longo das camadas de neurônios tornando uma rede neural capaz de executar e aprender tarefas mais complexas.

De acordo com Haykin (2007), define-se três tipos básicos de funções de ativação:

1. Função de limiar. Descrito na Figura 7, têm-se

$$\varphi(v) = \begin{cases} 1 & se \quad v \geq 0 \\ 0 & se \quad v < 0 \end{cases} \tag{2.9}$$

Correspondentemente, a saída do neurônio k que emprega esta função de limiar é expressa como

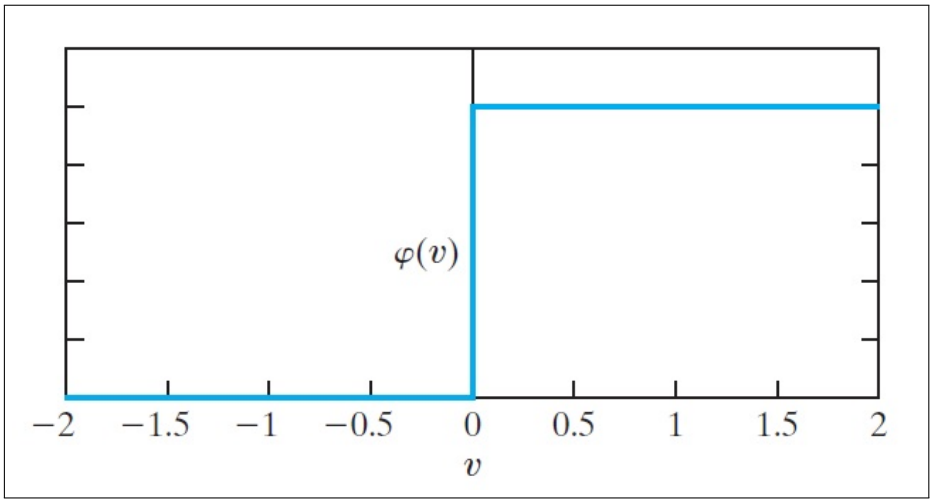
$$\varphi(y_k) = \begin{cases} 1 & se \quad v_k \geq 0 \\ 0 & se \quad v_k < 0 \end{cases} \tag{2.10}$$

onde v_k é o campo induzido do neurônio; isto é

$$v_k = \sum_{j=1}^{m} w_{kj} x_j + b_k \tag{2.11}$$

Essa função é extremamente simples, pois é utilizada num classificador binário, ou seja quando precisa-se dizer sim ou não para uma classe. Está função é contínua e diferenciável, outra vantagem é que seu comportamento está entre uma função linear e não linear, isso faz com que ela empurre os valores de Y para os extremos.

Figura 7 – Função de limiar



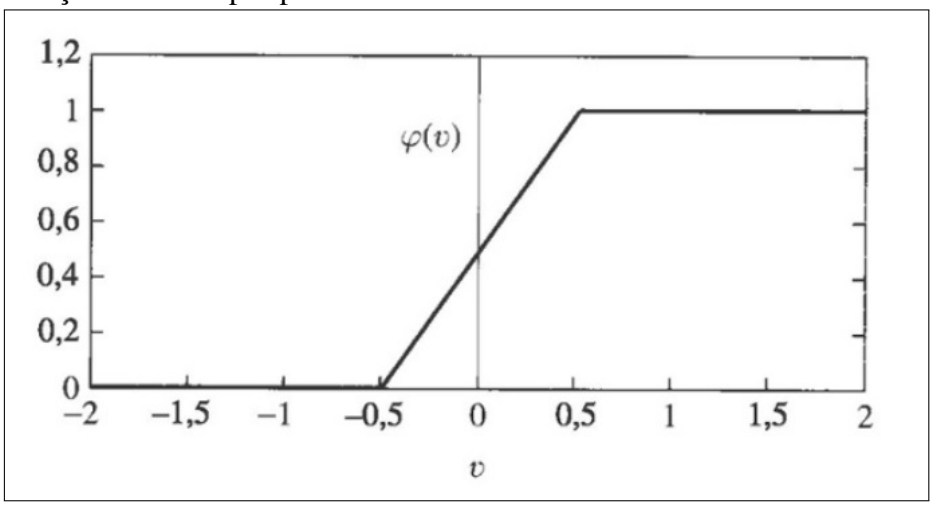
Fonte: (HAYKIN *et al.*, 2009)

2. Função Linear por Partes. A ilustração dessa função está na figura 8.

$$\varphi(v) = \begin{cases} 1, & v \geq +\frac{1}{2} \\ v, & +\frac{1}{2} > v > -\frac{1}{2} \\ 0, & v \leq -\frac{1}{2} \end{cases} \tag{2.12}$$

A função linear por partes é vista como uma aproximação de um amplificador não-linear.

Figura 8 – Função de linear por partes



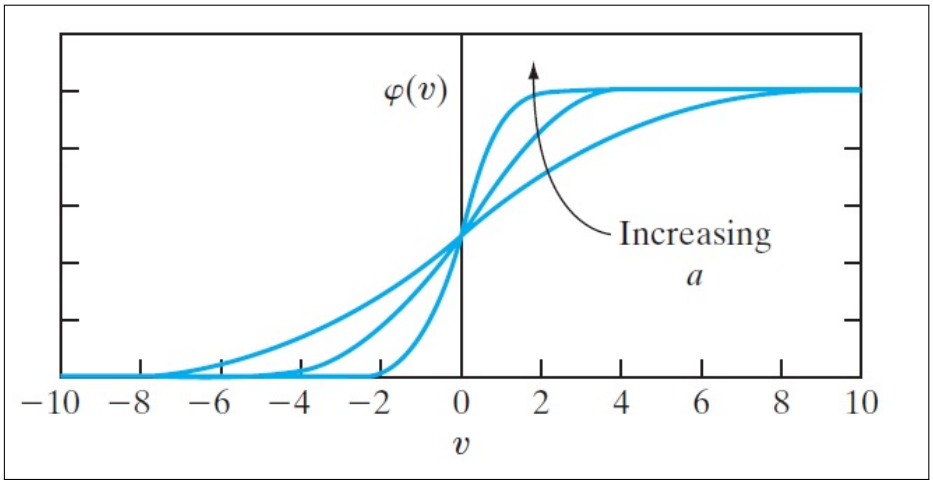
Fonte: (HAYKIN, 2007)

3. *Função Sigmoide*. Essa função, cujo o gráfico tem a forma de S (Figura 9), é de longe a forma mais comum de função de ativação amplamente utilizada na construção de redes neurais artificiais. Um exemplo de função sigmoide é a *função logística*, definida por

$$\varphi(v) = \frac{1}{1+exp(-av)} \tag{2.13}$$

O problema dessa função é que seus valores variam de 0 a 1, além disso, quando os gradientes ficam muito pequenos, ou seja, aproximando-se de zero dificulta a aprendizagem.

Figura 9 – Função Sigmoide



Fonte: (HAYKIN *et al.*, 2009)

Como a função sigmoide lida apenas com duas classes, outra função que é um tipo de sigmoide é a *Softmax*. Essa função transforma também as saídas para cada classe para

valores entre 0 e 1 e divide pela soma das saídas. Com isso é possível ter a probabilidade de uma entrada estar em uma determinada classe, assim, ela é adequada para problemas de classificação multiclasse (ACADEMY, 2021). Sua equação é definida por:

$$\sigma(\mathbf{z})_j = \frac{e^{z}j}{\sum_{k=1}^{K} e^{z}k} \quad for\ j = 1,2,...,K. \tag{2.14}$$

Portanto, a função de ativação é utilizada em cada camada onde a informação terá que se mover e ser processada em outras camadas da rede. Na próxima seção, será discutido mais sobre camadas de rede neural.

2.2.3 Redes Neurais Multicamadas

A maneira pela qual os neurônios estão estruturados em uma rede neural define qual o algoritmo de aprendizagem que será utilizado para o treinamento da rede. Assim, as Redes Neurais Multicamadas, são arquiteturas que possuem duas ou mais camadas de processamento formadas pelos neurônios, esses neurônios, também chamados de *neurônios ocultos* ou *unidades oculta*, têm a função de interpor entre a entrada externa e a saída da rede de maneira útil (HAYKIN, 2007).

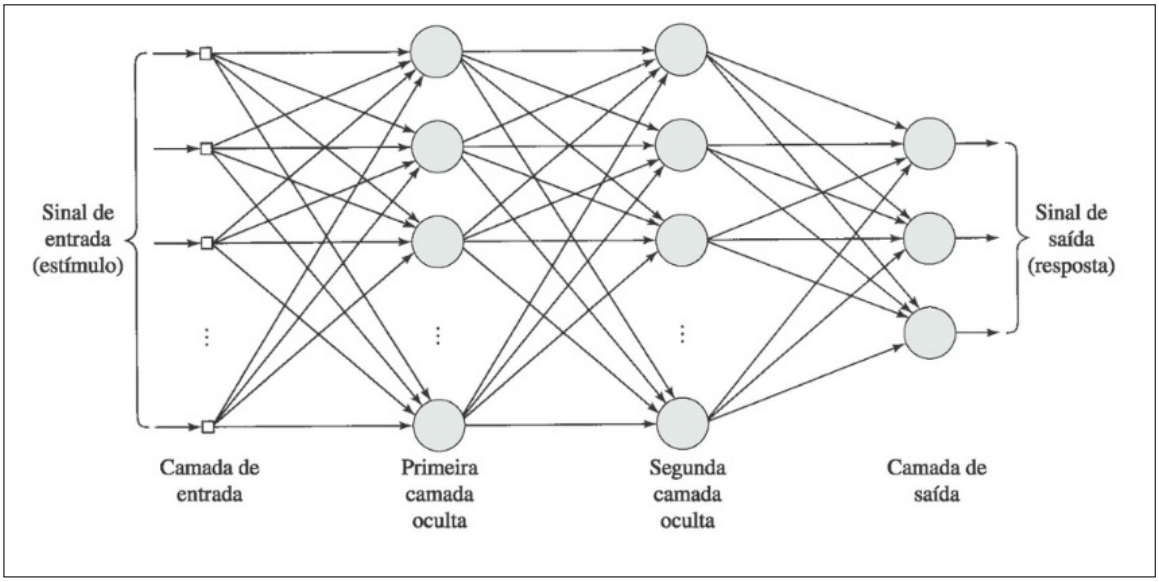
Por exemplo, as redes neurais que possuem só duas camadas são compostas por uma camada de entrada que se conecta a uma de saída. Logo, os neurônios da camada de entrada tem a função de distribuir cada uma das camadas de rede a todos os neurônio da camada seguinte. Vale ressaltar que, a redes de camada única de nós só conseguem resolver problemas linearmente separáveis, já as de múltiplas camadas resolvem problemas não-linearmente separáveis, pois conseguem separar os conjuntos de dados de forma linear usando k retas.

Contudo, antes de começar falar de fato das redes neurais multicamadas é preciso destacar o conceito de *perceptron* que consiste de um único neurônio na camada de saída. Esse é o modelo de arquitetura mais simples de uma rede neural artificial, representado na Figura 6. O Modelo Perceptron foi desenvolvido nas décadas de 1950 e 1960 pelo cientista Frank Rosenblatt inspirado em trabalhos anteriores Warren McCulloch e Walter Pitts. Embora já seja um modelo ultrapassado, ainda permite compreender a introdução de como funciona uma rede neural.

Além disso, de acordo com Haykin (2007), diz-se que uma rede neural é totalmente conectada quando qualquer nó da camada de rede está conectado a todos os nós da camada seguinte. As redes neurais multicamadas são normalmente chamadas de perceptrons de múltiplas camadas, as quais representam uma generalização do perceptron de camada única. Esses

perceptrons têm sido utilizados com sucesso na resolução de problemas difíceis utilizando um algoritmo popular conhecido por *algoritmo de retropropagação de erro (error backpropagation)* de treinamento supervisionado. A Figura 10 descreve uma rede neural multicamadas.

Figura 10 – Arquitetura de um perceptron de múltiplas camadas com duas camadas ocultas



Fonte: (HAYKIN, 2007)

Como cada arquitetura de RNA tem determinada função ou característica para se trabalhar com os dados, a subseção a seguir trata de um de tipo de arquitetura voltada para lidar com dados do tipo texto.

2.2.4 Redes Neurais Recorrentes

Segundo Chung *et al.* (2014), a estrutura de uma Rede Neural Recorrente (RNR) tem por base uma rede *feedforward* com algumas alterações que é capaz de lidar com uma entrada de sequência de comprimento variável, ou seja, é uma rede neural para processamento de dados sequenciais. As realimentações de uma rede recorrente consistem em saídas de neurônios de determinada camada serem reintroduzidas como entradas de neurônios de camadas anteriores ou da própria, no caso o neurônio pode ser realimentado por sua própria saída, essas possibilidades fazem com que a arquitetura dessa rede possa tomar diversas formas. Mais formalmente, dada uma sequência $x = (x_1, x_2, ..., x_t)$, a RNN atualiza seu estado oculto recorrente x_t por:

$$h_t = \begin{cases} 0, & t = 0 \\ \phi(h_{t-1}, x_t), & \textit{caso contrário} \end{cases} \tag{2.15}$$

onde ϕ é uma função não-linear, como a composição de um sigmoide logístico com uma transformação afim.

Também, opcionalmente a RNN pode ter a saída $y = (y_1, y_2, ..., y_t)$ que pode ser novamente variável de comprimento. A atualização do estado oculto recorrente é representado pela equação a seguir:

$$h_t = g(Wx_t + Uh_{t-1}) \tag{2.16}$$

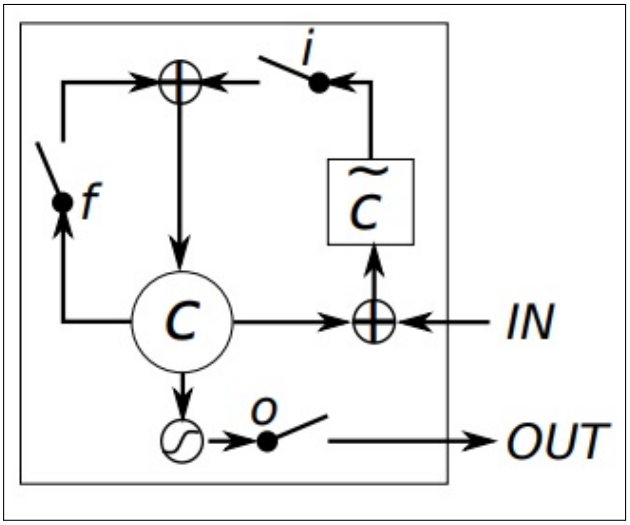
onde g é uma função suave e limitada, como uma função sigmóide logística ou uma função tangente hiperbólica.

Além disso, costuma-se dizer que as redes neurais recorrentes possuem "memória"na qual o propósito é preservar as informações no estado oculto da rede e que consegue passar muitos passos de tempo à medida que ela avança. Assim, isso se torna vantajoso para as redes recorrentes que utilizam para realizar tarefas que as redes de *feedforward* não consegue.

Vale ressaltar também que, existem dois tipos de abordagens atuais uma chamada de Long Short Term Memory (LSTM) e a outra chamada de Gated Recurrent Unit (GRU). A LSTM inicialmente proposta por Hochreiter e Schmidhuber (1997), desde então sofreu pequenas modificações, contém unidades especiais chamadas blocos de memória na camada oculta recorrente. Os blocos de memória contêm células de memória com auto-conexões armazenando o estado temporal da rede, além de unidades multiplicativas especiais chamadas portas para controlar o fluxo de informação (SAK *et al.*, 2014). Anteriormente, cada bloco de memória na arquitetura original continha um portão de entrada e um portão de saída. O portão de entrada controla o fluxo de ativações de entrada na célula de memória. O portão de saída controla o fluxo de saída da célula ativações no resto da rede. Só mais tarde é que foi adicionado um portão de esquecimento no bloco de memória, esse portão de esquecimento segundo (SAK *et al.*, 2014), dimensiona o estado interno da célula antes de adiciona-la como entrada através da conexão auto-recorrente da célula, portanto, esquecendo-se ou redefinindo de forma adaptativa a memória. A Figura 11 descreve um pouco da arquitetura da rede recorrente LSTM.

Formalmente uma rede recorrente LSTM calcula um mapeamento de uma sequência de entrada $x = (x_1, ..., x_T)$ para uma sequência de saída $y = (y_1, ..., y_T)$ calculando as ativações

Figura 11 – i, f, e o são respectivamente o portão de entrada, esquecimento e saída. c e $\tilde{c}$ denotam a célula de memória e o novo conteúdo da célula de memória.



Fonte: (CHUNG *et al.*, 2014)

da unidade de rede das seguintes equações iterativamente de $t = 1$ a T:

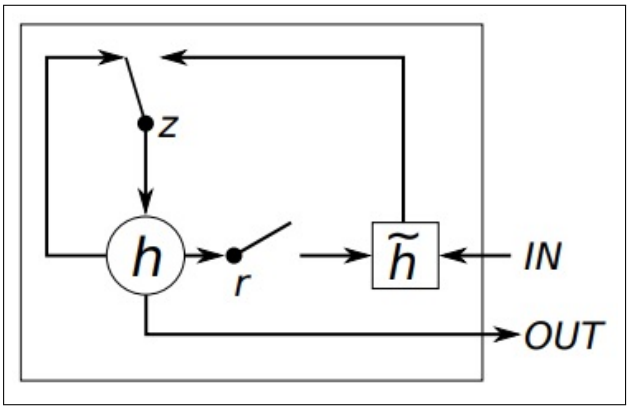
$$\begin{aligned}
\mathrm{i}_t &= \sigma(W_{ix}x_t + W_{im}m_{t-1} + W_{ic}c_{t-1} + b_i) \\
\mathrm{f}_t &= \sigma(W_{fx}x_t + W_{fm}m_{t-1} + W_{fc}c_{t-1} + b_f) \\
\mathrm{c}_t &= f_i \odot c_{t-1} + i_t \odot g(W_{cx}x_t + W_{cm}m_{t-1} + b_c) \\
\mathrm{o}_t &= \sigma(W_{ox}x_t + W_{om}m_{t-1} + W_{oc}c_t + b_o) \\
\mathrm{m}_t &= o_t \odot h(c_t) \\
\mathrm{y}_t &= \phi(W_{ym}m_t + b_y)
\end{aligned} \tag{2.17}$$

Tal que,

- Os termos W denotam matrizes de peso (por exemplo, W_{ix} é a matriz de pesos da porta de entrada para a entrada);
- W_{ic}, W_{fc}, W_{oc} são matrizes de peso diagonal para conexões peephole;
- O b termos denotam vetores de polarização (b_i é o vetor de polarização da porta de entrada);
- σ é a função sigmoide logística;
- i, f, o e c são respectivamente o portão de entrada, portão de esquecimento, portão de saída e veto de ativação de célula, todos os quais são do mesmo tamanho que a ativação da saída da célula do vetor m;
- $\odot$ é o produto elementar dos vetores;
- g e h são as funções de ativação de entrada e saída da célula;
- ϕ é a função de ativação da saída de rede;

Por fim, a GRU introduzida por Chung *et al.* (2014) objetivou resolver o problema do **gradiente de desaparecimento** que vem da RNN padrão. A GRU, também pode ser considerada uma variação do LSTM, pois são projetados de maneira semelhante. Cada GRU, tem um portão de reset e um portão de atualização que lembram o portão de esquecimento e portão de entrada do LSTM conforme Figura 12. No entanto, ao contrário do LSTM, a GRU expõe totalmente seu conteúdo de memória a cada timestep e saldos entre o conteúdo da memória anterior e o novo conteúdo de memória estritamente usando a integração com vazamento(CHUNG *et al.*, 2015).

Figura 12 – r e z são respectivamente os portões de reset e atualização, h e $\tilde{h}$ são a ativação e a ativação do candidato.



Fonte: (CHUNG *et al.*, 2014)

Segundo (CHUNG *et al.*, 2015), na timestep t, o estado h_j t da j-ésima GRU é calculado por

$$h_t^j = (1 - z_t^j)h_{t-1}^j + z_t^j \tilde{h}_t^j \tag{2.18}$$

- Onde h_{t-1}^j e $\tilde{h}_t j$ correspondem respectivamente ao conteúdo anterior de memória e o novo conteúdo de memória candidato.
- O portão de atualização z_t^j controla quando o conteúdo anterior de memória é para ser esquecido e quando o novo conteúdo de memória deve ser adicionado.

O portão de atualização é calculado com base nos estados ocultos anteriores h_{t-1} e entrada atual x_t:

$$z_t = \sigma(W_z x_t + U_z h_{t-1}), \tag{2.19}$$

O novo conteúdo de memória $\tilde{h}_t^j$ é calculado de forma semelhante a função de transição convencional:

$$\tilde{h}_t = \tanh(W x_t + r_t \odot U h_{t-1}), \tag{2.20}$$

onde $\odot$ é uma multiplicação de elementos.

Ainda, uma diferença importante da função de transição tradicional (Eq. (2.14)) é que os estados do passo anterior h_{t-1} é modulado pelas portas de reset r_t. Esse comportamento permite uma GRU ignorar os estados ocultos anteriores sempre que for considerado necessário, considerando os estados ocultos anteriores e a entrada atual:

$$r_t = \sigma(W_r x_t + U_r h_{t-1}), \tag{2.21}$$

Portanto, conforme (CHUNG *et al.*, 2015) o mecanismo de atualização ajuda a GRU a capturar dependências de longo prazo. Sempre que um recurso detectado anteriormente, ou o conteúdo da memória é considerado importante para uso posterior, o portão de atualização será fechado para transportar o conteúdo corrente de memória em vários timesteps. O mecanismo de reset ajuda a GRU a usar a capacidade do modelo de forma eficiente permitindo que ele redefina sempre que o recurso detectado não é mais necessário.

Além disso, é preciso que um modelo de RNA seja avaliado se está aprendendo ou não, e para isso é empregado funções de custo. A subseção a seguir detalha mais sobre o assunto.

2.2.5 Treinamento e Avaliação

O objetivo principal do treinamento de uma rede neural é fazer com que o algoritmo encontre um conjunto de pesos e *bias* que se adapte minimamente bem ao conjunto de entradas de treinamento, para produzir saídas desejadas (NIELSEN, 2015). Com isso, define-se *função de custo* ou função de perda para avaliar e quantificar o quão bem esse objetivo está próximo de ser alcançado.

Existem várias funções de custo, cada uma com suas características particulares, e que sua escolha deve ser feita de com base no objetivo a ser alcançado. Uma função de custo bastante conhecida é a do *erro quadrático médio*(mean squared erro - MSE), denotado por:

$$C(w,b) = \frac{1}{2n} \sum_x \|y(x) - a\|^2 \tag{2.22}$$

Onde,

- w é a coleção de todos os pesos da rede;
- b todos os *bias*;
- n é o número total de entradas de treinamento;
- a é o vetor de saídas quanto x é a entrada;

- $\sum$ é a soma sobre todas as entradas de treinamento x;

Pode-se observar que $C(w,b)$ é não negativo, pois cada termo na soma é não negativo. Além disso, $C(w,b) \approx 0$ quando $y(x)$ é próximo da saída a para todas as entrada de treinamento x.

Contudo, uma função de custo não consegue por si só fazer o treinamento da rede neural. É preciso que ao final do treinamento a função de custo seja a menor possível, assim, o problema de treinamento é equivalente ao problema de minimizar essa função. Para isso, existem técnicas e algoritmos que otimizam funções, nesse caso, o método que será abordado é o do *Gradiente Descendente* (NIELSEN, 2015).

Primeiramente, para encontrar o mínimo de uma função, basta calcular sua derivada parcial. Isso é simples quando a função tem poucas variáveis para manipular. Mas, se torna um pesadelo quando uma função tem muitas outras variáveis, no caso, uma rede neural por exemplo, pode ter muitas varáveis e que a função de custo vai depender de milhões ou até bilhões de pesos e *bias*, deixando a rede bastante complicada e que o uso do cálculo de derivadas não vai funcionar para a minimização (NIELSEN, 2015).

Felizmente, uma analogia simples com o cálculo da derivada pode ser utilizada. Se for imaginado que uma função seja uma espécie de tigela ou um vale, em seguida, colocando uma pequena bola no topo e deixando-a rolar é notável que ela irá parar no fundo da tigela ou do vale. Também, é possível notar que a bola pode ser colocada em qualquer lugar do topo e sempre irá parar no fundo, ou seja, escolhendo-se um lugar aleatório qualquer do topo, ela sempre irá para o mesmo lugar (NIELSEN, 2015).

Com base nessa imaginação, o método do Gradiente tem um funcionamento semelhante. Ele é basicamente um vetor de derivadas parciais. Segundo a função de custo citada anteriormente, pode-se definir o vetor de derivadas de C da forma $\left(\frac{C}{\partial v_1}, \quad \frac{\partial C}{\partial v_2}\right)^T$. Em que T é uma operação de transposição. Logo, o vetor de *gradiente* é denotado por ∇C, ou seja, $\nabla C \equiv \left(\frac{C}{\partial v_1}, \quad \frac{\partial C}{\partial v_2}\right)^T$ (NIELSEN, 2015).

A equação a seguir explica por que ∇C é chamado de vetor de gradiente:

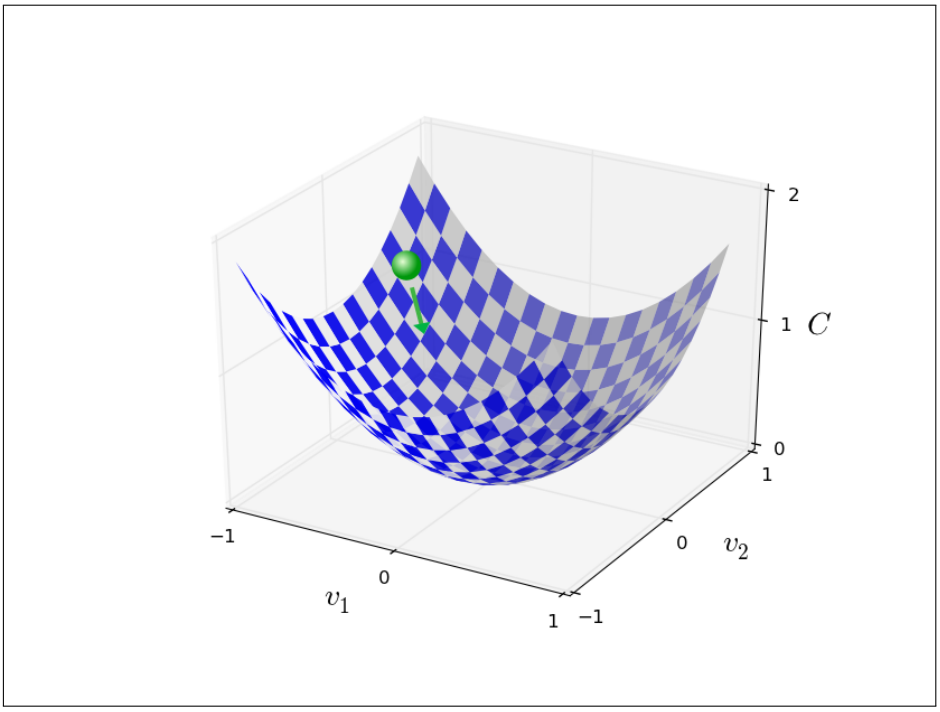
$$\nabla C \approx \nabla C \cdot \nabla v \tag{2.23}$$

Essa equação permite escolher ∇v de modo a tornar ∇C negativo. Também, pode-se supor que $\nabla v = -\eta \nabla C$, onde η é um parâmetro pequeno e positivo (conhecido como *taxa de aprendizagem*). Então, a equação acima diz que $\nabla C \approx -\eta \nabla C \cdot \nabla C = -\eta \left\|\nabla C\right\|^2$. Pois, $\left\|\nabla C\right\|^2 \geq 0$, isso garante que $\nabla C \leq 0$, ou seja, C sempre diminuirá, nunca irá aumentar.

A regra de atualização a seguir também pode ser utilizada para fazer o mesmo cálculo para encontrar o mínimo global, e que também representa a descida do gradiente. Resumidamente, a maneira como o algoritmo funciona, é calcular iterativamente o gradiente de ∇C, a Figura 13 faz essa ilustração.

$$v \to v' = v - \eta \nabla C \tag{2.24}$$

Figura 13 – Ilustração da descida do *gradiente*



Fonte: (NIELSEN, 2015)

Vale ressaltar que, existem algumas variações do gradiente descendente que visam a aceleração do aprendizado. Pois, se o conjunto de dados for muito grande, então o cálculo do gradiente irá demorar bastante. Para resolver isso, o *Gradiente Descendente Estocástico* embaralha os dados e observa se uma parte é parecida com outra parte, então pode-se conseguir uma aproximação do gradiente olhando alguns exemplo dos dados (NIELSEN, 2015).

Já o Gradiente Descendente *Mini-Batch*, utiliza uma quantidade pré-fixa de dados escolhidos aleatoriamente chamados de *mini-lote*. Formalmente, a partir de um conjunto $X_1, X_2, X_3, ..., X_m$ tem-se:

$$w_k \to w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X\,j}}{\partial w_k} \tag{2.25}$$

$$b_l \to b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X\,j}}{\partial b_l} \tag{2.26}$$

Onde, w_k e b_l são respectivamente os pesos e *bias*. As equações acima são resolvidas iterativamente até que as entradas do treinamento sejam esgotadas.

Portanto, se no treinamento de uma rede neural conseguiu encontrar um conjunto de pesos que se adaptou ao conjunto de dados de entrada do treinamento, têm-se então um modelo adequado para um problema. Baseado nisso, uma rede neural é empregada em uma abordagem chamada Aprendizagem de índice.

2.2.6 Aprendizagem de índice

As estruturas de dados tradicionais ainda possuem limites de desempenho quanto a distribuição de dados e não tiram proveito dos padrões que os dados podem minimamente fornecer. Além disso, sabe-se que os dados não obedecem um padrão que seja conhecido e perfeito, caso contrário, os custos de desempenho seria menos dispendiosos. Porém, a tentativa de aprender um modelo que correlacione e distingue os padrões dos dados pode resultar automaticamente em uma estrutura denominada *índice aprendido*, que aproveita esses padrões para obtenção de desempenho (KRASKA *et al.*, 2018).

Assim, a Tabela Hash por exemplo, utiliza uma função hash que recebe uma chave e retorna um valor correspondendo a uma posição da tabela a qual a chave será armazenada. O desafio maior dessa abordagem é impedir que diferentes chaves sejam direcionadas para a mesma posição na tabela e que dentre as variações de implementação estão o encadeamento separado, sondagem linear e o cuckoo hashing. Independente da implementação as colisões ainda afetam o desempenho e o armazenamento, e o modelo de índice aprendido pode ser uma alternativa para tentar diminuir as colisões.

Conforme dito anteriormente, pode-se concluir que uma estrutura de dados de índices é um modelo que recebe uma chave como entrada e prevê a posição do registro e isso chega a um ponto importante a se observar, pois um modelo que prediz a posição de uma chave dentro de uma tabela então ele se aproxima de uma Cumulative Distribution Function (CDF), logo ela pode ser usada para prever a posição de um dado com a seguinte equação:

$$p = F(Key) * N \tag{2.27}$$

Onde o p é a posição estimada, $F(Key)$ é a CDF estimada para os dados para estimar a probabilidade de observar uma chave menor ou igual à chave de consulta $P(X \leq Key)$ e N o total de chaves (KRASKA *et al.*, 2018).

Para substituir uma função de hash, a CDF pode ser usada para aprender melhor a distribuição das chaves. E o modelo CDF seria $h(K) = F(k) * M$ onde K é a função hash, assim, se o modelo aprendesse perfeitamente a distribuição das chaves então cada uma ocuparia uma única posição, logo, eliminaria os conflitos entre chaves. Acrescenta-se também, que a função acumulativa sozinha não resolve o problema por inteiro, é preciso de uma arquitetura de modelo recursivo.

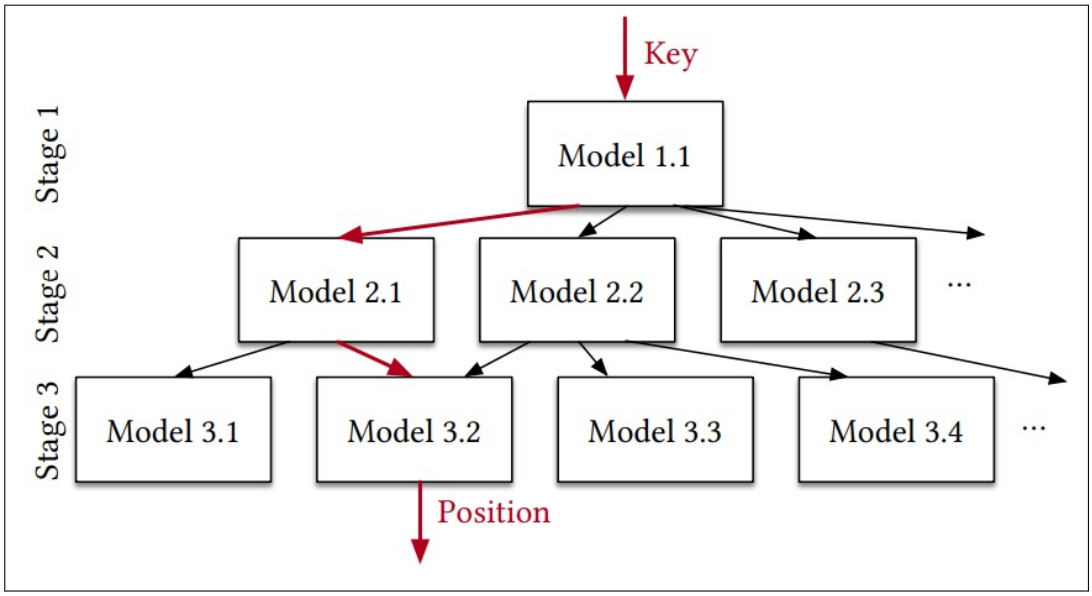
Basicamente, esse modelo recursivo é uma hierarquia de modelos onde em cada estágio o modelo leva a chave como entrada e, com base nela, escolhe outro modelo, até o estágio final predizer a posição. Formalmente, $f(x)$ onde x é a chave, y ε $[0,N)$ a posição, l o estágio, M_l modelos e modelo k no estágio l denotado por $f_l^{(}k)$. A função de custo utilizada para treinar é dada por:

$$L_l = \sum_{(x,y)} (f_l^{(\lfloor M_l f_{(l-1)}(x)/N \rfloor)}(x) - y)^2 \tag{2.28}$$

$$L_0 = \sum_{(x,y)} (f_0(x) - y)^2 \tag{2.29}$$

Esse modelo treina iterativamente cada estágio com perda L_l para construir o modelo completo. A ideia por trás disso é fazer com que cada modelo faça uma previsão para a posição da chave, assim essa previsão se propaga para o próximo modelo onde ele fará uma previsão melhor ainda com erro mínimo como mostra a Figura 14 e Figura 15.

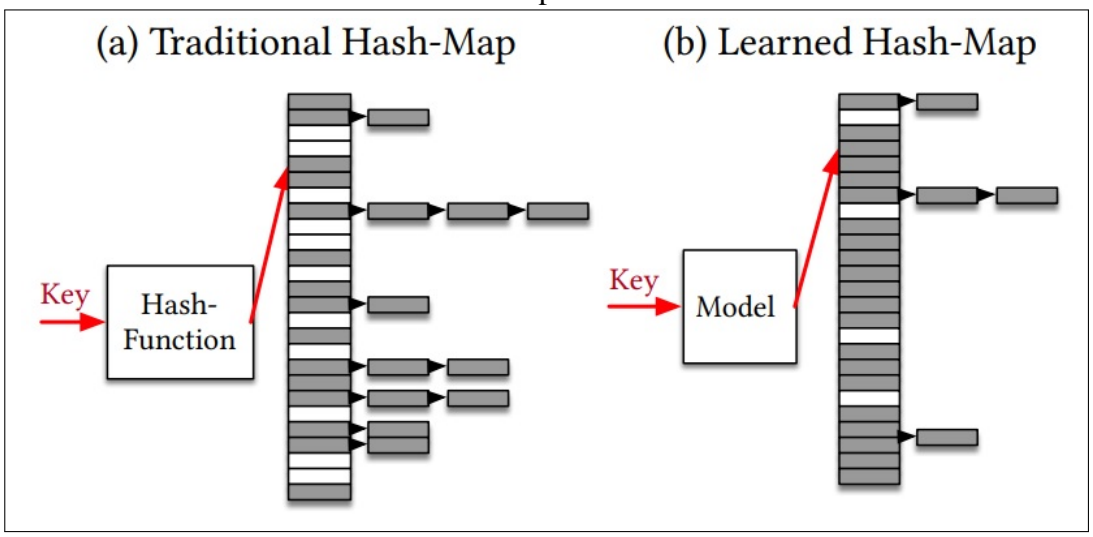
Figura 14 – Estágios de modelos



Fonte: (KRASKA *et al.*, 2018)

Segundo a avaliação de (KRASKA *et al.*, 2018) sobre os índices aprendidos, foi utilizado a implementação da Tabela Hash por encadeamento separado usando a função hash

Figura 15 – Tabela Hash tradicional vs Hash aprendido



Fonte: (KRASKA *et al.*, 2018)

MurmurHash3 comparando o desempenho com o modelo recursivo aprendido com modelo de 2 estágios de tamanho 100k no segundo estágio e nenhuma camada oculta. Para isso, foram utilizados três conjunto de dados onde dois são do mundo real, weblogs e maps e um conjunto de dados sintético, lognormal.

3 TRABALHOS RELACIONADOS

Neste capítulo, apresenta-se os trabalhos que motivaram o desenvolvimento do presente trabalho. Na seção 3.1, é descrita a abordagem Aprendizagem de Índice, que é o foco principal deste trabalho. Na seção 3.2, apresenta os desafios de indexar dados em memória. Por último, a seção 3.2 descreve a metologia para trabalhar com cadeias de caracteres em uma RNA, que também é o foco principal deste trabalho.

3.1 The Case for Learned Index Structures

Segundo (KRASKA *et al.*, 2018), propôs ser possível a substituição de estrutura de dados de índices como as B-Trees, Hash Table e Filtros Blooms por outros modelos de aprendizagem profunda denominando assim de Índices Aprendidos. Ele afirma que a ideia chave é aprender como essas estruturas tradicionais classificam ou realizam uma busca de um registro, assim, o modelo pode prever com eficácia a posição ou se o registro existe.

Para tornar isso possível, o autor destaca primeiramente o funcionamento de uma árvore *B* salientando que essa estrutura consegue predizer a localização de um valor que se encontra em um conjunto classificado por chaves, com um erro mínimo de 0 e máximo do tamanho da página. Assim, afirma que essa árvore é um modelo tal que em *machine learning* seria uma árvore de regressão o que torna possível substitui-la por uma rede neural. Acrescentasse também que, a árvore *B* precisa ser reequilibrada e na terminologia de aprendizagem seria retreinada.

Em seguida, desenvolveu o modelo Learning Index Framework (LIF) e Recursive Model Indexes (RMI). O LIF é considerado um sistema de índice em que dada uma especificação de índice ele gera diferentes configurações para que possa ser possível otimiza-las e depois testar automaticamente e assim aprender. Já o RMI, consiste numa estrutura hierárquica de modelos, onde em cada estágio o modelo recebe a chave como entrada que com base nela e sua estimativa de erro possa escolher outro modelo, isso até o estagio final que terá a posição prevista. Ainda, no RMI é possível torna-lo hibrido por exemplo, nas camadas superiores uma rede neural usando ReLU e nas camadas inferiores uma árvore *B*. Vale ressaltar que, se nesse modelo híbrido for muito difícil de aprender então todas as camadas superiores seriam substituída pela árvore *B*, e portanto o modelo seria virtualmente uma árvore *B* inteira.

Para treinar os dados, foi utilizados dois conjuntos de dados do mundo real e um

conjunto de dados sintéticos lognormal, estes tipos de dados foram escolhidos com padrões bastante complexo, só para dificultar a aprendizagem de índice. No primeiro experimento foi feito a comparação do modelo RMI com uma árvore *B* otimizada. Os resultados estão na Figura 16.

Figura 16 – Índice aprendido vs B-Tree

		Map Data			Web Data			Log-Normal Data		
Type	**Config**	Size (MB)	Lookup (ns)	Model (ns)	Size (MB)	Lookup (ns)	Model (ns)	Size (MB)	Lookup (ns)	Model (ns)
Btree	page size: 32	52.45 (4.00x)	274 (0.97x)	198 (72.3%)	51.93 (4.00x)	276 (0.94x)	201 (72.7%)	49.83 (4.00x)	274 (0.96x)	198 (72.1%)
	page size: 64	26.23 (2.00x)	277 (0.96x)	172 (62.0%)	25.97 (2.00x)	274 (0.95x)	171 (62.4%)	24.92 (2.00x)	274 (0.96x)	169 (61.7%)
	page size: 128	13.11 (1.00x)	265 (1.00x)	134 (50.8%)	12.98 (1.00x)	260 (1.00x)	132 (50.8%)	12.46 (1.00x)	263 (1.00x)	131 (50.0%)
	page size: 256	6.56 (0.50x)	267 (0.99x)	114 (42.7%)	6.49 (0.50x)	266 (0.98x)	114 (42.9%)	6.23 (0.50x)	271 (0.97x)	117 (43.2%)
	page size: 512	3.28 (0.25x)	286 (0.93x)	101 (35.3%)	3.25 (0.25x)	291 (0.89x)	100 (34.3%)	3.11 (0.25x)	293 (0.90x)	101 (34.5%)
Learned Index	2nd stage models: 10k	0.15 (0.01x)	98 (2.70x)	31 (31.6%)	0.15 (0.01x)	222 (1.17x)	29 (13.1%)	0.15 (0.01x)	178 (1.47x)	26 (14.6%)
	2nd stage models: 50k	0.76 (0.06x)	85 (3.11x)	39 (45.9%)	0.76 (0.06x)	162 (1.60x)	36 (22.2%)	0.76 (0.06x)	162 (1.62x)	35 (21.6%)
	2nd stage models: 100k	1.53 (0.12x)	82 (3.21x)	41 (50.2%)	1.53 (0.12x)	144 (1.81x)	39 (26.9%)	1.53 (0.12x)	152 (1.73x)	36 (23.7%)
	2nd stage models: 200k	3.05 (0.23x)	86 (3.08x)	50 (58.1%)	3.05 (0.24x)	126 (2.07x)	41 (32.5%)	3.05 (0.24x)	146 (1.79x)	40 (27.6%)

Fonte: (KRASKA *et al.*, 2018)

Conjunto de dados do tipo *String* também foram usados. Nesse caso, para testar o desempenho dos índices aprendidos foram usados 10 milhões de documentos contínuos de um grande índice da web utilizados pela Google. O desempenho foi comparado entre um modelo RMI não hibrido que faz uma busca quartenária denominando-o de Learned QS, um modelo RMI hibrido, o LIF e finalmente a árvore *B*. O resultado está na Figura 17.

Figura 17 – Dataset de String: Aprendizagem de índice vs B-Tree

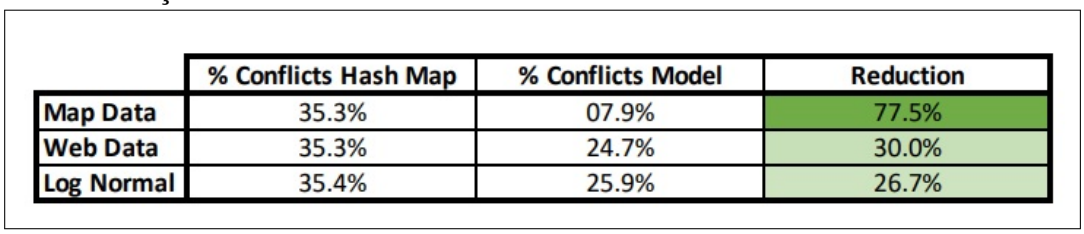
	Config	Size(MB)	Lookup (ns)	Model (ns)
Btree	page size: 32	13.11 (4.00x)	1247 (1.03x)	643 (52%)
	page size: 64	6.56 (2.00x)	1280 (1.01x)	500 (39%)
	page size: 128	3.28 (1.00x)	1288 (1.00x)	377 (29%)
	page size: 256	1.64 (0.50x)	1398 (0.92x)	330 (24%)
Learned Index	1 hidden layer	1.22 (0.37x)	1605 (0.80x)	503 (31%)
	2 hidden layers	2.26 (0.69x)	1660 (0.78x)	598 (36%)
Hybrid Index	t=128, 1 hidden layer	1.67 (0.51x)	1397 (0.92x)	472 (34%)
	t=128, 2 hidden layers	2.33 (0.71x)	1620 (0.80x)	591 (36%)
	t= 64, 1 hidden layer	2.50 (0.76x)	1220 (1.06x)	440 (36%)
	t= 64, 2 hidden layers	2.79 (0.85x)	1447 (0.89x)	556 (38%)
Learned QS	1 hidden layer	1.22 (0.37x)	1155 (1.12x)	496 (43%)

Fonte: (KRASKA *et al.*, 2018)

Até agora toda a comparação e aprendizado foi feito em cima da B-Tree. Outra estrutura de dados que o autor abordou para construir um índice aprendido foi a tabela hash que é objeto de estudo deste trabalho. Aprender a distribuição das chaves é uma maneira de aprender a função de hash tornando-a melhor, assim se o modelo aprender perfeitamente então os conflitos

poderiam deixar de existir. A avaliação foi feita com três conjuntos de dados medindo a taxa de conflito da função de hash, é notório o quanto um índice aprendido ganha em desempenho de uma tabela hash de acordo com a figura 18.

Figura 18 – Redução de conflitos



	% Conflicts Hash Map	% Conflicts Model	Reduction
Map Data	35.3%	07.9%	77.5%
Web Data	35.3%	24.7%	30.0%
Log Normal	35.4%	25.9%	26.7%

.

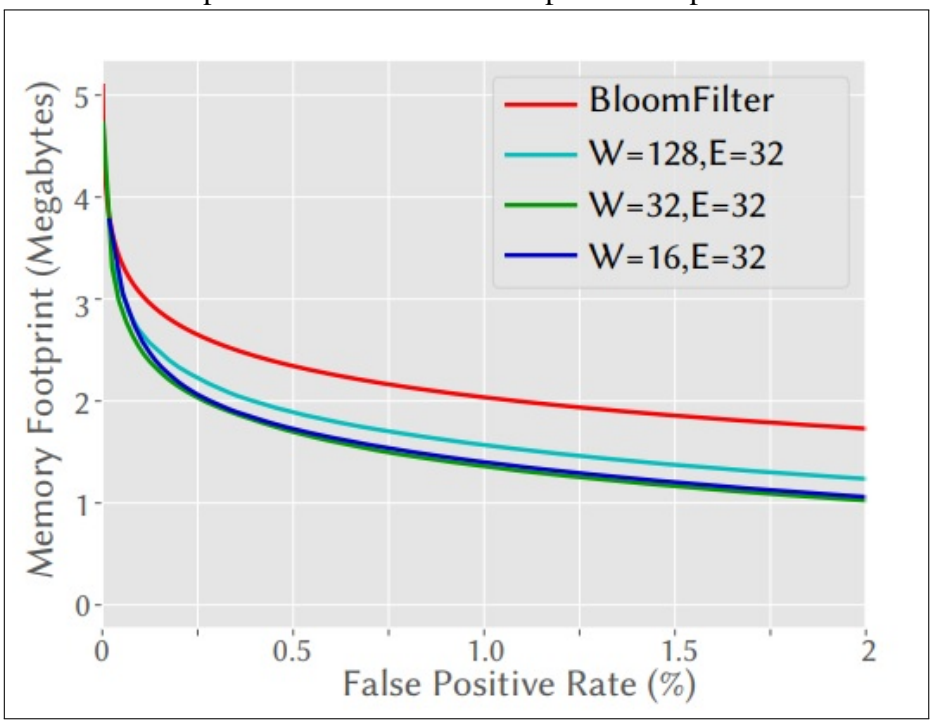
Fonte: (KRASKA *et al.*, 2018)

Por fim, a última estrutura que foi usada para aprender foi o Filtro Bloom. Basicamente é uma estrutura de dados probabilística para testar se um elemento pertence a um conjunto, filtro bloom são uteis quando geralmente estão relacionado a ganhos de velocidade e espaço. A diferença básica dele na aprendizagem para as estruturas aprendidas citadas anteriormente é que ele precisa aprender uma função que separe todas as chaves de todo o resto, enquanto as estruturas aprendem como distribuir as chaves.

Vale ressaltar que, na tabela hash uma função de hash é boa quando ela consegue evitar o máximo de colisões, já no filtro bloom quando é usado o modelo hash então uma boa função de hash seria uma que tivesse muitas colisões entre as chaves e muitas colisões entre não-chaves, mas poucas colisões de chaves e não-chaves, outro modelo de filtro bloom que pode ser usado é se enquadrando em um problemas de classificação probabilística binária, ou seja, quando pretende-se aprender um modelo em que dada uma consulta consiga dizer se x é uma chave ou não-chave.

Portanto, o treinamento para o filtro bloom se deu a partir de um conjunto de dados de um relatório de transparência do Google que consistia de 1.7 milhões de URLs. O objetivo era poder determinar URLs de phishing para uma ista negra, assim foi formado um conjunto negativo de URLs válidas e URLs que poderiam ser confundidas com sites de phishing, esse conjunto foi dividido aleatoriamente em conjunto de treinamento, validação e teste e por fim uma rede neural recorrente foi treinada com esses dados. O filtro Bloom Aprendido melhora o consumo de memória de acordo com a Figura 19.

Figura 19 – Filtro Bloom aprendido usando RNN é representada por W.



Fonte: (KRASKA *et al.*, 2018)

3.2 FASTER: An Embedded Concurrent Key-Value Store for State Management

Segundo (CHANDRAMOULI *et al.*, 2018), diversos dados são criados em fontes de borda, como dispositivos da Internet, aplicativos móveis, navegadores e servidores atualmente. Esses dados são processados por aplicativos e serviços na nuvem para que se obtenha percepções. Esse processamento pode incluir análises ad-hoc de dados em lotes (por exemplo, no Hadoop e Spark) ou monitoramento em tempo real. Assim, a gestão de estado se torna um componente crítico no que tange a necessidade de processamento.

Para (CHANDRAMOULI *et al.*, 2018), um dos desafios enfrentados por tais aplicações ocorrem quando se tem muitos estados sendo acessados e acaba por exceder a memória principal. O autor também destaca uma solução existente adotada por muitos sistemas em que o particionamento do estado é feito em várias máquinas com uso de estrutura de dados em memória pura, porém, a solução é dispendiosa deixando recursos das máquinas subutilizados tornando recuperações de falhas bastante complexa. Outra solução bastante popular é o armazenamento chave-valor projetado para manipular dados maiores que a memória principal armazenando dados na memória secundária e com suporte a recuperação de falhas.

Assim, o autor criou um novo sistema de armazenamento chave-valor chamado FASTER que foi projetado para auxiliar as aplicações que envolve a gestão de estado com

atualização intensiva. Ele suporta dados maiores que a memória e quando encaixados na memória pode processar centenas de operações por segundo.

Para isso o FASTER usa um índice de hash que é simultâneo, redimensionável e amigável ao cache contendo ponteiros para registros chave-valor e um alocador de registros que aloca e gerencia registros individuais. Além disso, o autor propõe o uso do HybridLog, uma nova estrutura de dados que combinada atualizações no local(na memória) e organização estruturada por log(em disco), enquanto fornece acesso simultâneo a registros sem interrupção. Portanto, o HybridLog abrange a memória principal e o armazenamento secundário em que uma parte dessa memória atua como cache para os registros mais atualizados.

Portanto, de acordo com o autor FASTER consegue melhor taxa de transferência de até 160 milhões de operações por segundo em uma máquina do que os sistemas implantados amplamente hoje e supera as estruturas de dados de memória quando a carga de trabalho cabe na memória.

3.3 CHARAGRAM: Embedding Words and Sentences via Character n-grams

Representar sequencias textuais tem sido um componente fundamental para a *Processing Natural Language* (PNL) onde arquiteturas funcionais tem sido propostas para permitir a modelagem para a composição das sequencias de palavras. Com isso, o autor propõe o CHARAGRAM com uma arquitetura funcional mais simples representando uma sequência de caractere por um vetor contendo contagens de caractere.

O autor faz uma revisão de trabalhos relacionados iniciando com uma abordagem mais simples falando sobre o uso de informações de sub-palavras em modelos de incorporação de palavras, isso permite que o modelo aprenda e possa executar tarefas específicas. Outra abordagem recente é o uso de arquiteturas funcionais mais ricas para converter sequências de caracteres em palavras embeddings, também há o trabalho que faz o uso de análise morfológica não supervisionada. Outros treinaram modelos de linguagem RNN em nível de fornecer recursos para tarefas de PNL, incluindo tokenização, segmentação e normalização de texto. Mas, o trabalho que mais se aproxima da abordagem do autor é o Deep Semantic Similarity Model ou Deep Structured Semantic Model (DSSM) onde as palavras são representadas por vetores contendo contagens de caracteres n-grams utilizado para recuperação da informação.

Os experimentos realizados pelo autor se deu em três conjuntos. Nos dois primeiros experimentos é comparado a capacidade que o modelo tem de capturar a similaridade semântica

para palavras e sentenças. Para a semelhança de palavras, o experimento foi baseado principalmente em dois conjuntos de dados que são usados com mais frequência para avaliação de semelhança semântica o conjunto WordSim-353 e o conjunto SimLex-999. Já para semelhança de sentença, foram avaliados 22 conjuntos de dados de similaridade textual. Para os experimentos de similaridade de palavras treinou-se pares de palavras e para o experimento de sentenças também foi treinado pares de frases. Por fim, no terceiro experimento é uma tarefa de classificação, o autor a partir de um modelo existente, faz modificações nesse modelo fazendo a inclusão da arquitetura que desenvolveu, charCNN, charLSTM e CHARAGRAM.

Também foram realizado uma análise quantitativa e análise qualitativa. Para a análise quantitativa foi abordado a questão das palavras fora do vocabulário, ou seja, as palavras desconhecidas. O modelo do autor também chamado de CHARAGRAM-PHRASE apresenta melhor desempenho em relação ao modelo existente PARAGRAM-PHRASE, pois quanto maior o número de palavras desconhecidas mais esse modelo degrada de acordo com os resultados da Figura 20. Já na análise qualitativa, o modelo PARAGRAM-PHRASE não tem a capacidade de modelar a ordem das palavras, resumidamente o modelo não consegue lidar com palavras que são negação o que não ocorre com o modelo CHARAGRAM-PHRASE conforme Figura 21. O autor consegue então demostrar que em seu modelo CHARAGRAM, os embeddings superam as arquiteturas mais complexas baseadas em recorrência e redes neurais convolucionais.

Figura 20 – Desempenho medido de acordo com número de palavras desconhecidas

Number of Unknown Words	N	PARAGRAM-PHRASE	CHARAGRAM-PHRASE
0	11,292	71.4	**73.8**
1	534	68.8	**78.8**
2	194	66.4	**72.8**
≥ 1	816	68.6	**77.9**
≥ 0	12,108	71.0	**74.0**

.

Fonte: (WIETING *et al.*, 2016)

Figura 21 – Palavras vizinhas mais próximas do bigram(sequência de duas palavras)

Bigram	CHARAGRAM-PHRASE	PARAGRAM-PHRASE
not capable	incapable, unable, incapacity	not, capable, stalled
not able	unable, incapable, incapacity	not, able, stalled
not possible	impossible impracticable unable	not, stalled, possible
not sufficient	insufficient, sufficient, inadequate	not, sufficient, stalled
not easy	easy, difficult, tough	not, stalled, easy

.

Fonte: (WIETING *et al.*, 2016)

Pelo exposto, (KRASKA *et al.*, 2018) apresentou ser possível substituir estrutura de dados tradicionais por uma nova abordagem denominada Aprendizagem de Índice, que utiliza modelos de aprendizagem profunda. Em (CHANDRAMOULI *et al.*, 2018), abordou uma estrutura híbrida que utiliza a Tabela Hash para resolução do problema de indexar dados em memória. Já em (WIETING *et al.*, 2016), o autor expôs uma maneira simples de como converter sequências de caracteres em números (ou "vetorizar"o texto) antes de alimentá-lo no modelo, pois modelos de aprendizado de máquina recebem vetores (matrizes de números) como entrada. Deste modo, todas essas principais características é explorado no presente trabalho, conforme descreve o capítulo a seguir.

4 METODOLOGIA

Devido aos bons resultados obtidos por Kraska *et al.* (2018) na aplicação de Aprendizagem de Índice para mostrar que é possível substituir uma estrutura de dados por um modelo de aprendizagem profunda, motiva-se realizar experimentos computacionais utilizando essa abordagem, mas focados apenas em dados do tipo “cadeia de caracteres”. Desta forma, neste trabalho será implementado uma RNA para o tipo de dado especificado. Em seguida, será comparado o seu desempenho com o da estrutura Tabela Hash.

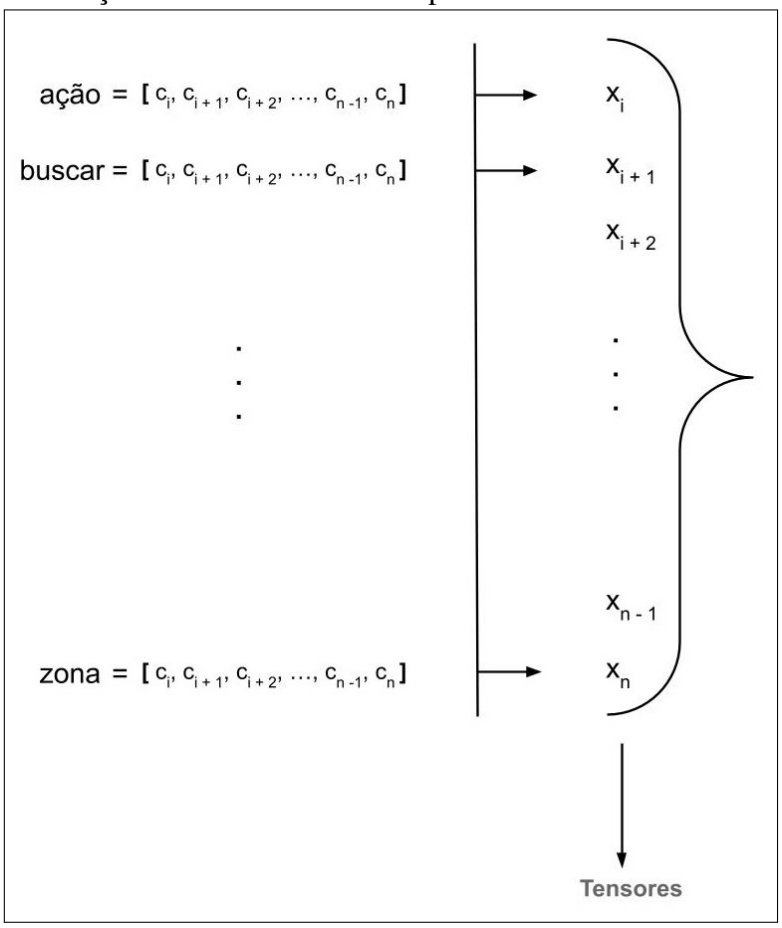
Neste capítulo, é proposto um modelo de RNA para treinamento dos dados e as métricas para avaliação, conforme a seção 4.1.

4.1 Modelo de Aprendizagem e Métricas de Avaliação da Tabela Hash

De maneira simplificada, o modelo foi construído com a utilização de um tipo de Rede Neural Artificial, mais especificamente Rede Neural Recorrente (RNR). Os parâmetros como o número de camadas, a quantidade de neurônios por camada, taxa de aprendizado e função de custo foram definidos de maneira empírica sendo definido aquele que apresentou favorável para realizar o experimento. Ao trabalhar com texto, deve-se primeiro criar uma estratégia para converter as sequencias de caracteres em números (ou "vetorizar"o texto) antes de alimentar o modelo, visto que modelos de aprendizado de máquina recebem vetores (tensores) como entrada.

Assim, a RNA recebe como entrada uma String S, em que cada caractere c_i da sequência S é submetido ao processo de codificação, cujo objetivo é convertê-los para vetores numéricos conforme representação da Figura 22. Observa-se que o vetor resultante da codificação são padronizados para o tamanho da maior String do vocabulário. Além disso, antes do processo de codificação todas as Strings precisam estar ordenadas.

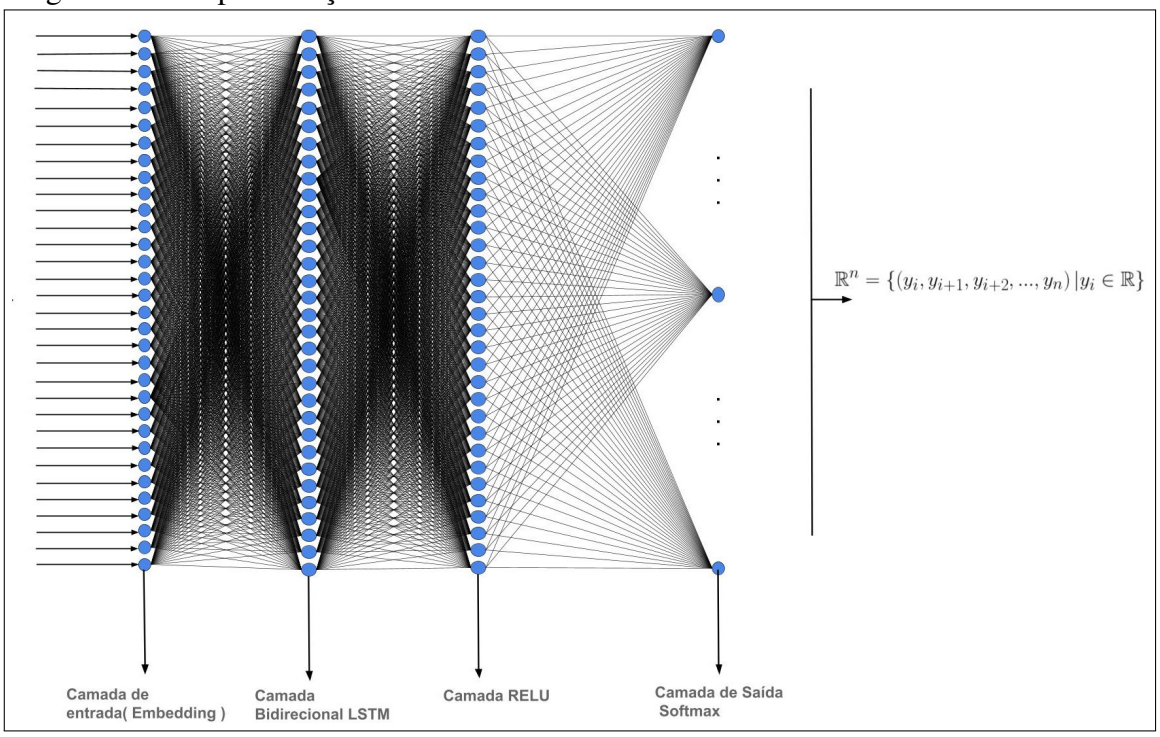
Figura 22 – Representação da entrada codificada para a rede neural



Fonte: Próprio autor.

As camadas são empilhadas sequencialmente para construir o modelo classificador. A primeira camada é uma camada Embedding. Essa camada pega o vocabulário em inteiros e olha o vetor embedding capturando parte da semântica da entrada e colocando entradas semanticamente semelhantes próximas umas das outras. Esses vetores são aprendidos pelo modelo, ao longo do treinamento. A Figura 23 tem a representação completa da arquitetura.

Figura 23 – Representação das camadas da RNR e sua saída



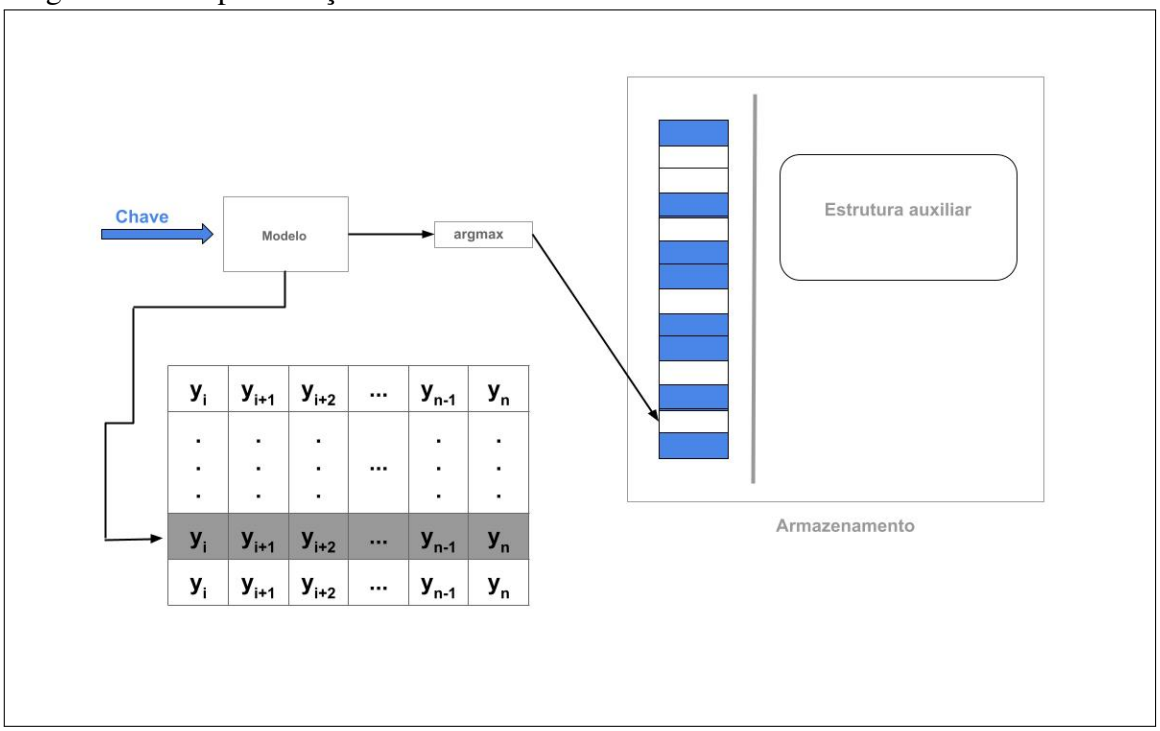
Fonte: Próprio autor.

A segunda camada, oculta, é composta por uma camada RNR Bidirecional do tipo LSTM de 32 neurônios que processa a saída da camada anterior iterando através dos elementos, ou seja, propaga a entrada para a frente e para trás através da camada e então concatena a saída final com a próxima camada.

A terceira camada, também oculta, possui uma RNA de arquitetura mais simples, a Rede Neural Multicamadas. Ela consiste de 32 neurônios com a função de ativação RELU que fazem algum processamento e envia para camada final. A quarta e ultima camada possui uma RNA de arquitetura simples e densa, com a softmax como função de ativação e com a quantidade de neurônios equivalente ao tamanho do vocabulário.

Essa camada tem como saída os tensores com distribuição de probabilidade. Assim, após fazer a inferência de um elemento, utiliza-se uma função auxiliar para retornar o índice do valor máximo ao longo de um eixo especificado. No caso de múltiplas ocorrências dos valores máximos, é devolvido o índice correspondente à primeira ocorrência. Dessa forma, têm-se uma posição estimada da chave correspondente. Caso a posição já esteja ocupada, será utilizada uma outra estrutura para armazenar essa chave. Como o Cuckoo Hashing apresenta bom desempenho ele será utilizado. A Figura 24 representa o processo de predição de uma chave e o armazenamento na estrutura.

Figura 24 – Representação da inferência de uma chave e seu armazenamento



Fonte: Próprio autor.

Já com relação as estruturas tradicionais, elas serão implementadas conforme propriedades expostas na fundamentação deste trabalho. Em atenção ao Cuckoo Hashing, para evitar seu ciclo infinito será estabelecido o valor de 5 deslocamento para alterar o tamanho da tabela, isso implica na sua eficiência, pois quanto menor o número de deslocamento maior será o aumento de memória devido o redimensionamento. E se o valor de deslocamento para redimensionar for muito grande, haverá o aumento do tempo de busca de uma posição vazia.

Como medida de desempenho para avaliação, foi analisado o consumo de memória da Tabela Hash comparando-o com o consumo de memória do modelo de índice. Mediu-se também o tempo médio de busca para algumas distribuições de dados como normal, uniforme e distribuição de frequência $zipf$, assim como o desvio padrão para cada técnica, seja ela Encadeamento Separado, Sondagem Linear e Cuckoo Hashing. Conforme o tipo da distribuição, é gerado índices que corresponderão às chaves selecionadas para realização do experimento. Por fim, salienta-se que o foco principal deste trabalho se concentra em medir as colisões das chaves, já que isso é um problema da estrutura da Tabela Hash.

4.2 Cenário

O algoritmo de aprendizagem deste trabalho foi implementado em Python, usando o ambiente Google Colab, assim como as estruturas de Hash tradicionais. Os modelos de RNAs foram implementados em Keras, com o *Backend* Tensorflow. E Para o ambiente de testes utilizou-se o computador com seguintes configurações:

1. Memória RAM de 13GB.
2. Placa de vídeo NVIDIA Tesla T4 16GB.
3. 68.35GB de HD

O Dataset, conjunto de dados para treinamento, de palavras utilizado neste trabalho foi o Wikipedia Summary Dataset (TSCHEEPERS, 2017). Mas qualquer vocabulário de palavras pode ser empregado para o treinamento da RNA, pois o modelo é próprio para esse tipo de dado.

5 RESULTADOS

Após a exposição do modelo apresentado neste trabalho, é necessário validar sua eficiência em relação as demais estruturas já existentes. Após efetuar a simulação de um banco de dados do tipo chave-valor usando as estruturas tradicionais e o modelo proposto neste trabalho, este capitulo irá apresentar os resultados através de uma comparação baseada nas métricas e distribuições de dados citados anteriormente. Salienta-se que os experimentos foram realizados várias vezes e não foi identificado melhores casos, pois a divergência entre os resultados eram insignificantes.

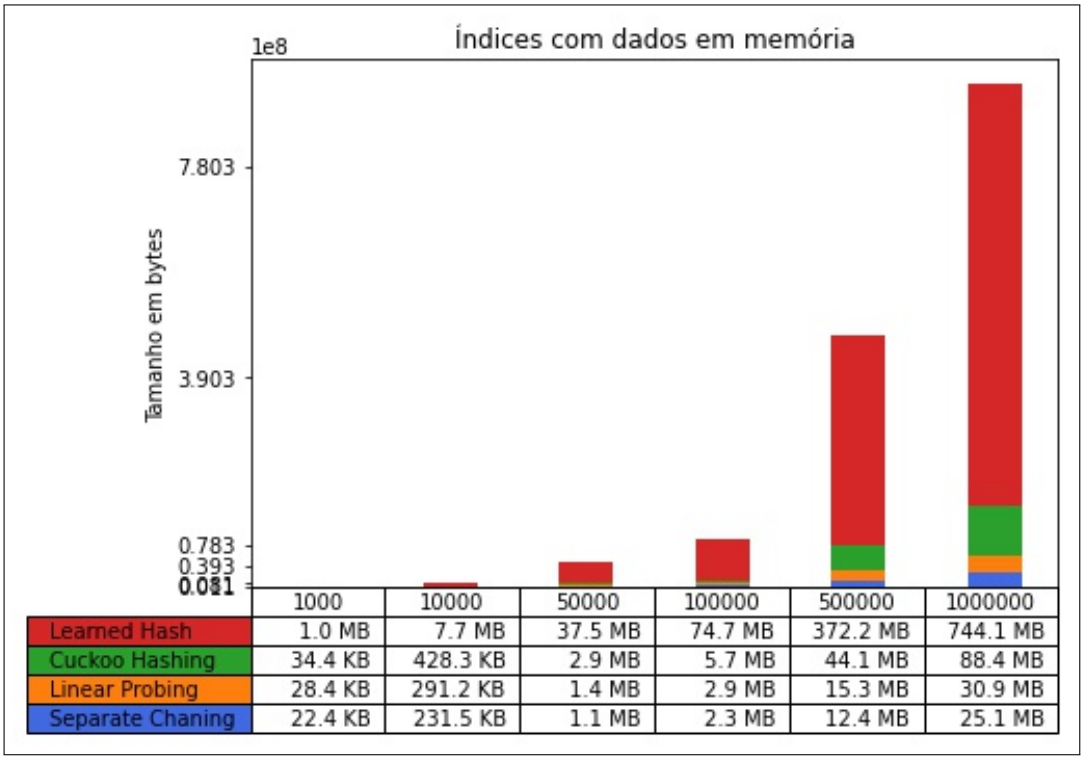
Para calcular o consumo de memória das estruturas foi utilizado um módulo da linguagem python que salva os dados em memória, que posteriormente facilitou avaliar o tamanho ocupado por aquela estrutura. Já a quantidade de colisões foram calculados durante as buscas das chaves assim como o tempo médio de buscas.

5.1 Resultados dos testes com dados em memória

No quesito consumo de memória, percebe-se, através do gráfico apresentado na Figura 25, que as estruturas tradicionais são muito melhores do que o modelo de índice aprendido. Para os diferentes tamanhos de vocabulário, a ordem da melhor para a pior estrutura de consumo de memória foi Encadeamento Separado, Sondagem Linear, Cuckoo Hashing e Hash Aprendido.

Isso se deve ao fato de que, estruturas tradicionais usam arrays simples enquanto que um modelo de rede neural utiliza-se de grandes matrizes de números. Tais resultados comprovam a ineficácia do modelo proposto, contrariando assim os bons resultados apresentados por (KRASKA *et al.*, 2018).

Figura 25 – Gráfico barras empilhadas considerando o consumo de memória



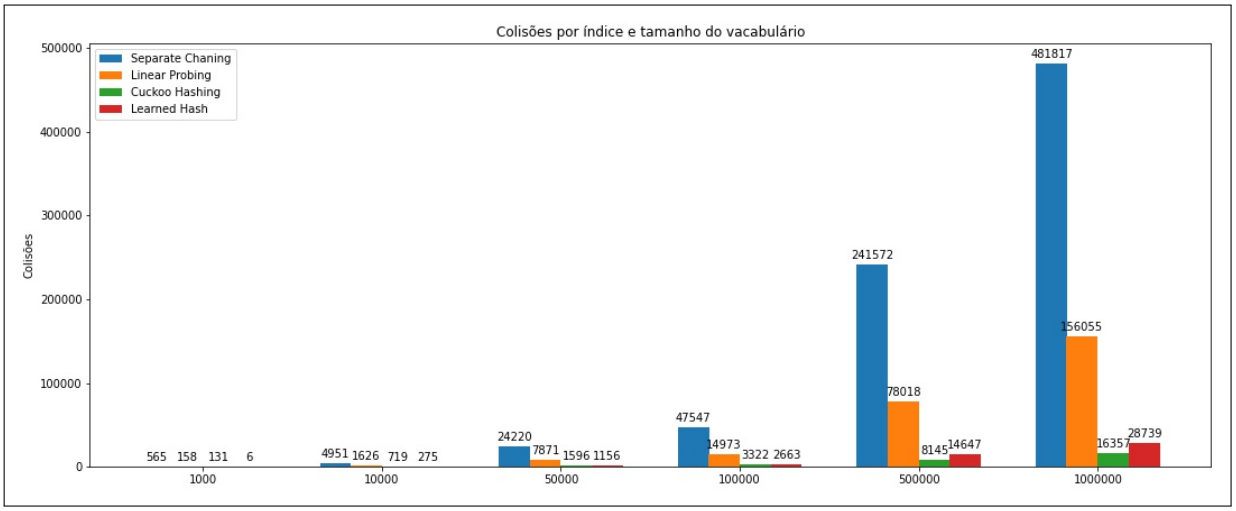
	1000	10000	50000	100000	500000	1000000
Learned Hash	1.0 MB	7.7 MB	37.5 MB	74.7 MB	372.2 MB	744.1 MB
Cuckoo Hashing	34.4 KB	428.3 KB	2.9 MB	5.7 MB	44.1 MB	88.4 MB
Linear Probing	28.4 KB	291.2 KB	1.4 MB	2.9 MB	15.3 MB	30.9 MB
Separate Chaning	22.4 KB	231.5 KB	1.1 MB	2.3 MB	12.4 MB	25.1 MB

Fonte: Próprio autor.

5.2 Resultados dos testes com distribuição normal

Em relação a quantidade de colisões ocorridas durante a simulação, o modelo de índice proposto neste trabalho apresentou resultados satisfatório. Apesar disso, vale destacar através do gráfico apresentado na Figura 26, que a medida que o tamanho do vocabulário aumenta o seu desempenho vai sendo superado pelo Cuckoo Hashing.

Figura 26 – Gráfico de barras considerando as colisões em cada estrutura

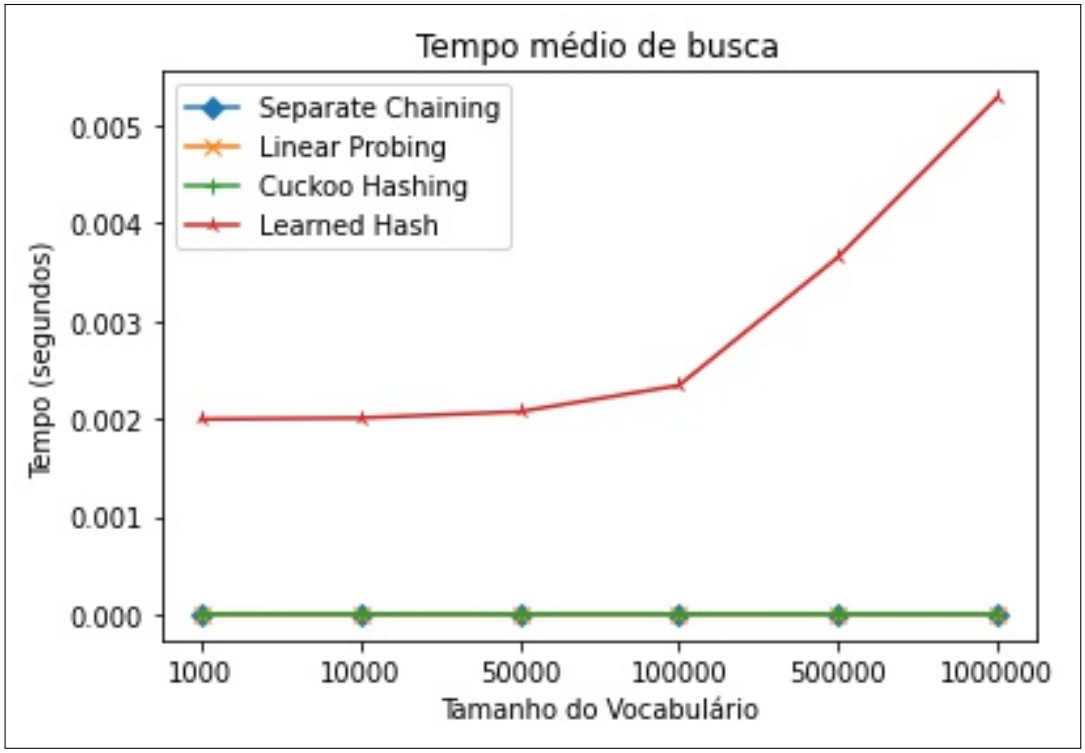


Fonte: Próprio autor.

Já em relação a métrica de tempo médio de busca, o modelo mostrou-se desfavorável no ambiente de simulação e as estrutura tradicionais mostraram-se ser mais eficiente. Apesar disso, é um resultado compreensível pois o calculo da função hash é uma operação menos custosa do que a multiplicação de matrizes usada no modelo.

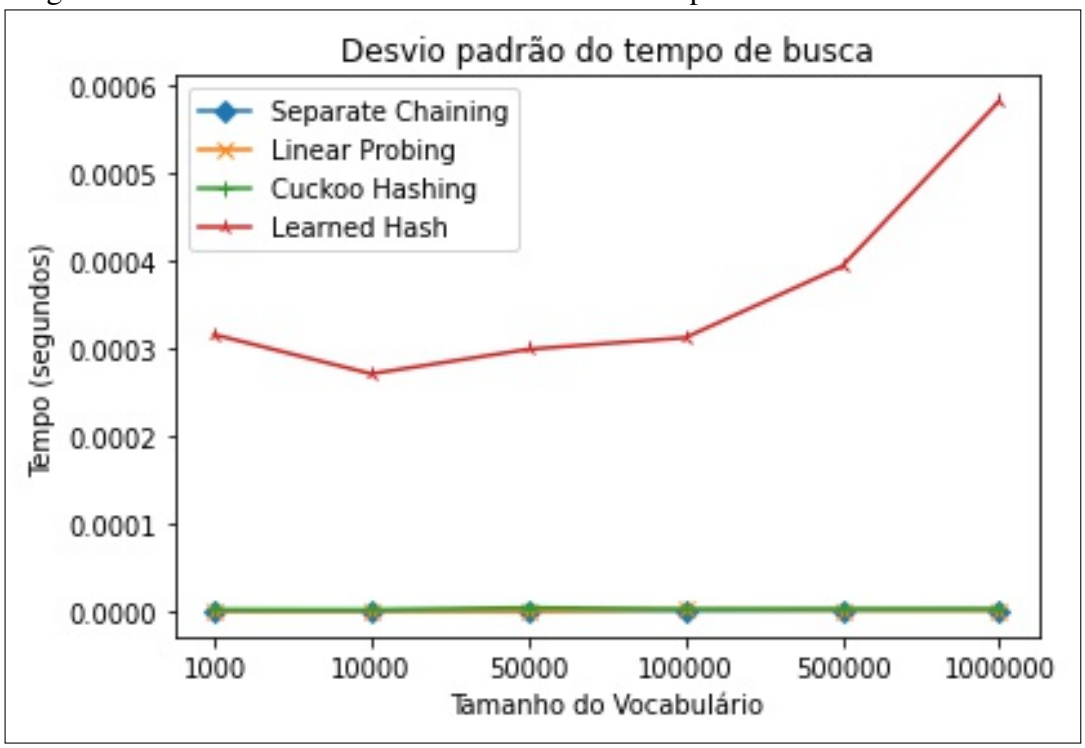
Conforme mostra o gráfico apresentado na Figura 27, o tempo médio de busca do modelo aumenta a medida que o tamanho do vocabulário aumenta, principalmente quando ultrapassa cem mil palavras. E na métrica desvio padrão, percebe-se, através da Figura 28 que ele tem leve divergência da média, principalmente o vocabulário de tamanho dez mil que apresenta maior divergência. O desvio para cima e para baixo são apresentados respectivamente pelas Figuras 29 e 30.

Figura 27 – Gráfico de linha considerando o tempo de busca



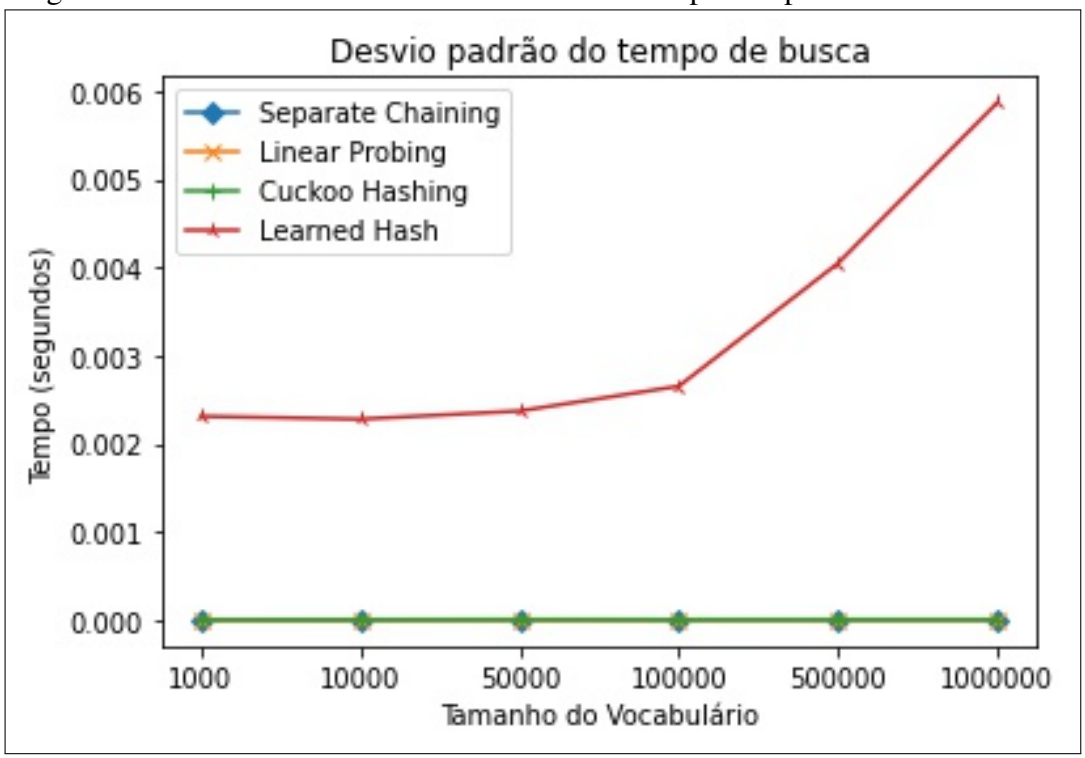
Fonte: Próprio autor.

Figura 28 – Gráfico de linha considerando o desvio padrão



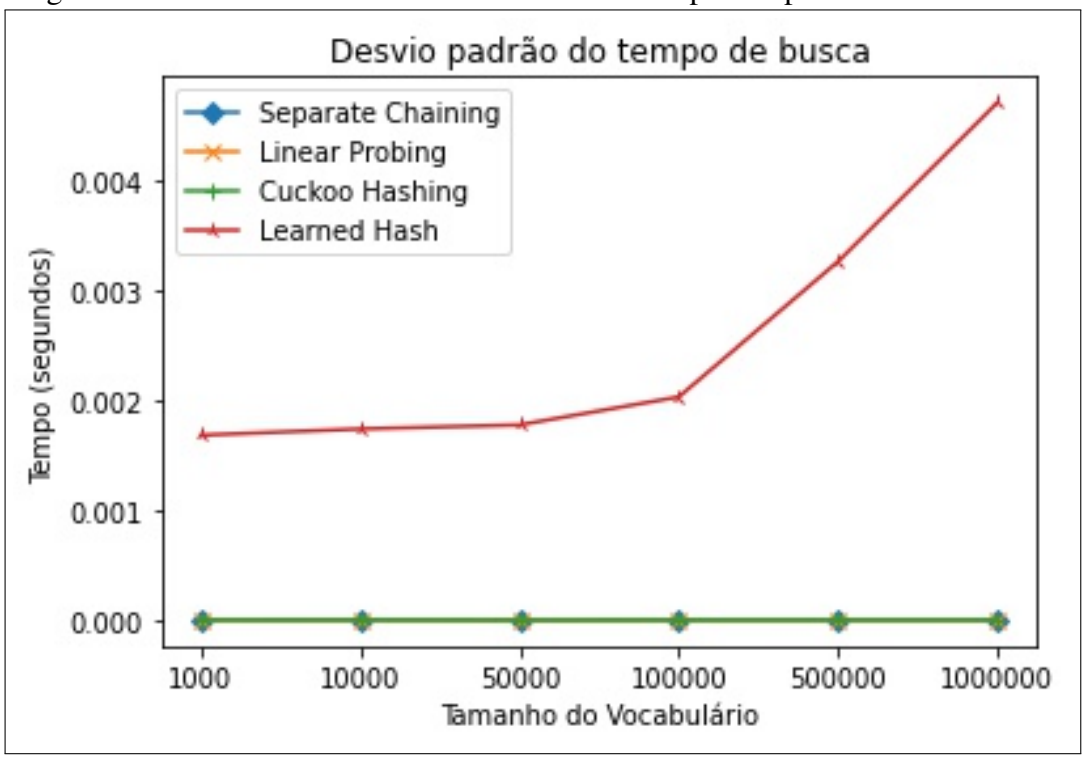
Fonte: Próprio autor.

Figura 29 – Gráfico de linha considerando o desvio padrão para cima



Fonte: Próprio autor.

Figura 30 – Gráfico de linha considerando o desvio padrão para baixo

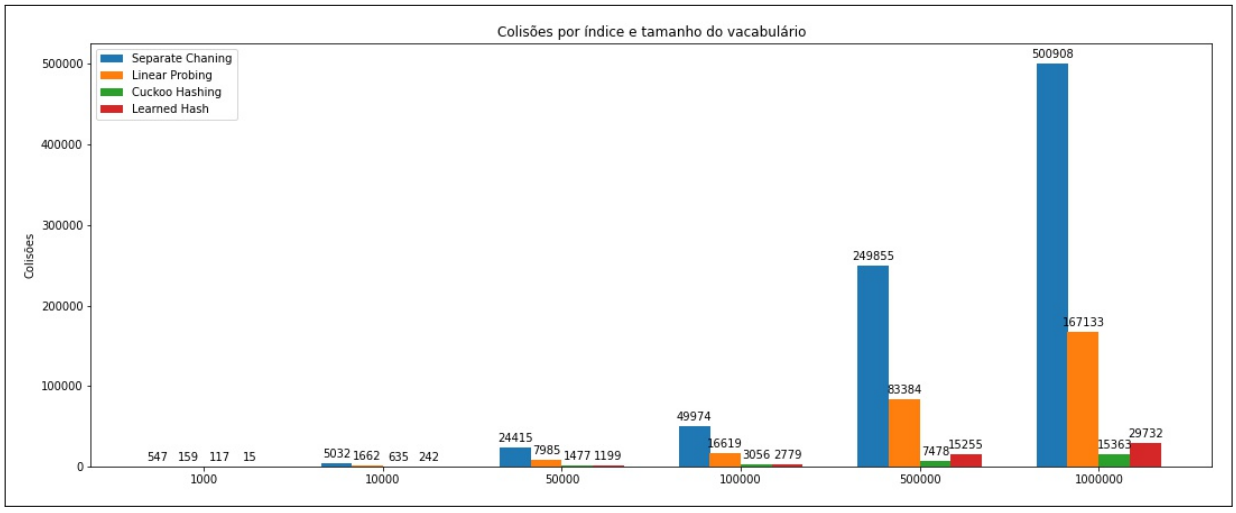


Fonte: Próprio autor.

5.3 Resultados dos testes com distribuição uniforme

Analisando as colisões com distribuição de dados do tipo uniforme, percebe-se, através do gráfico apresentado na Figura 31 que há semelhança na quantidade de colisões quando comparada com a distribuição de dados do tipo normal apresentada na seção anterior. Assim, o modelo apresenta resultados satisfatório até o vocabulário de tamanho cem mil, depois disso ele começa ser superado novamente pelo Cuckoo Hashing.

Figura 31 – Gráfico de barras considerando as colisões em cada estrutura

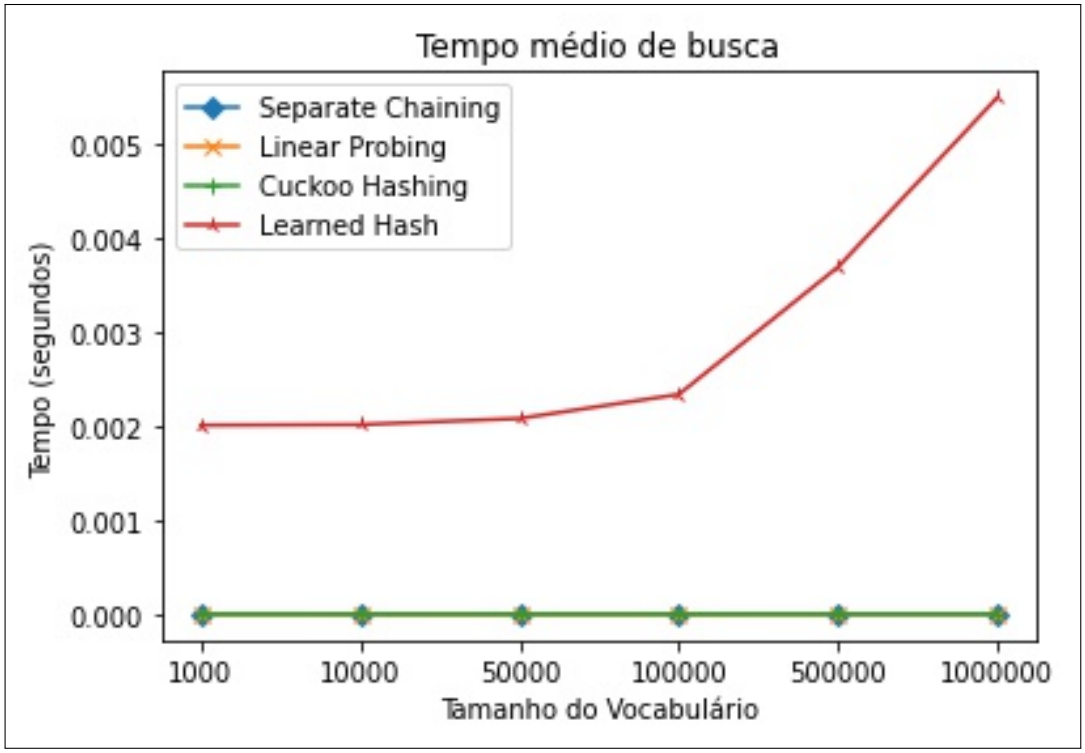


Fonte: Próprio autor.

Analisando também o tempo médio de busca, o modelo de índice aprendido têm o pior resultado quando comparado com as estruturas tradicionais. Vale observar que, quando o tamanho do vocabulário aumenta o tempo de busca aumenta. Isso é mais notável quando ultrapassa-se os cem mil conforme mostra a Figura 32.

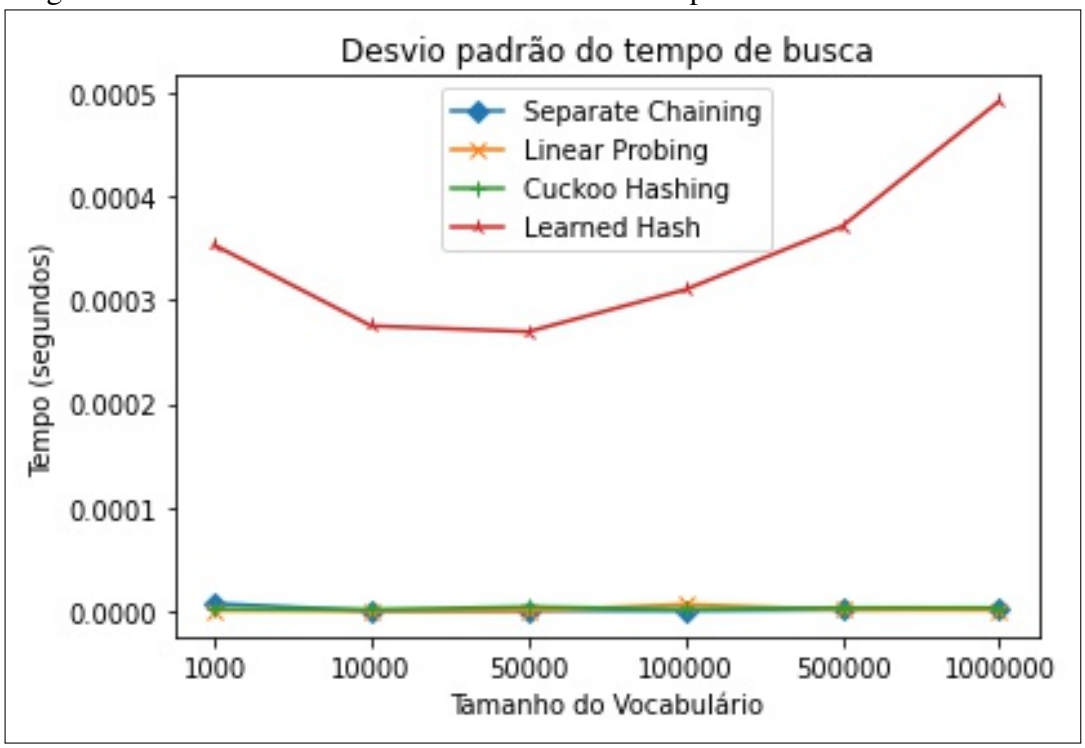
O desvio padrão também se assemelha ao desvio padrão da distribuição de dados do tipo normal conforme o gráfico apresentado na Figura 33. O que há de comum nos dois é que um vocabulário de tamanho mil a divergência da media é considerável, isso tudo tratando-se apenas do modelo aprendido enquanto que os índices tradicionais se mantém com resultados melhores. Percebe-se ainda que, o desvio padrão para cima e para baixo mostrados na Figura 34 e 35, respectivamente, são mais próximos do tempo médio de busca.

Figura 32 – Gráfico de linha considerando o tempo de busca



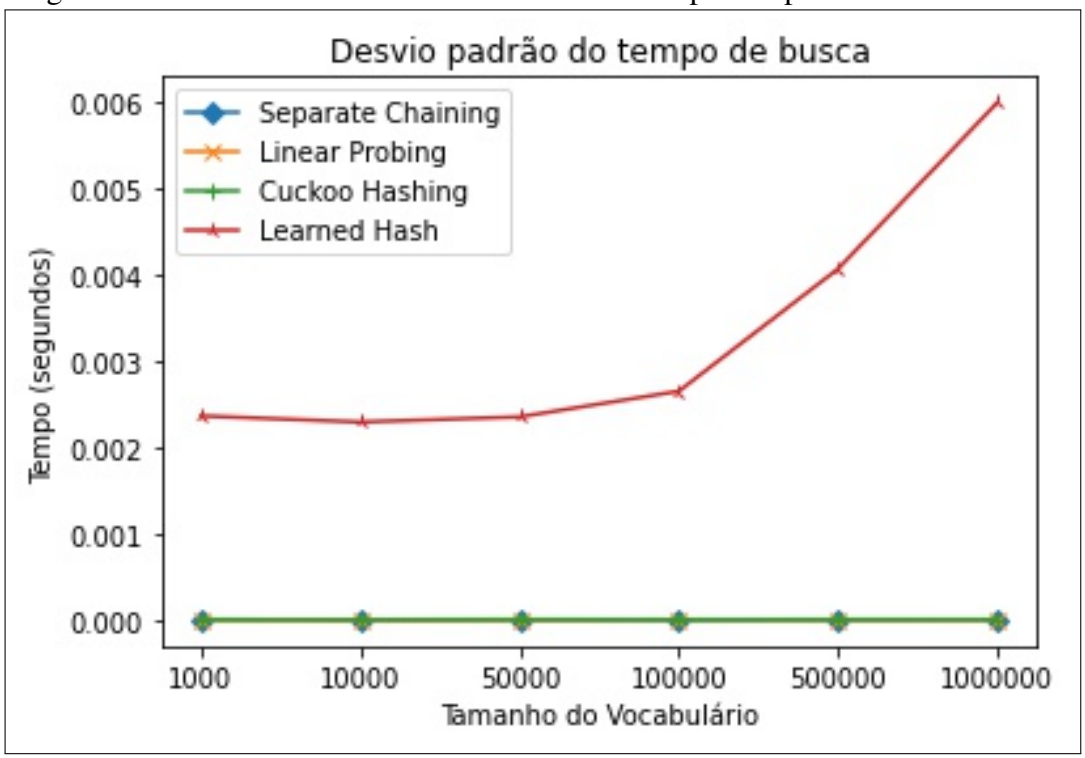
Fonte: Próprio autor.

Figura 33 – Gráfico de linha considerando o desvio padrão



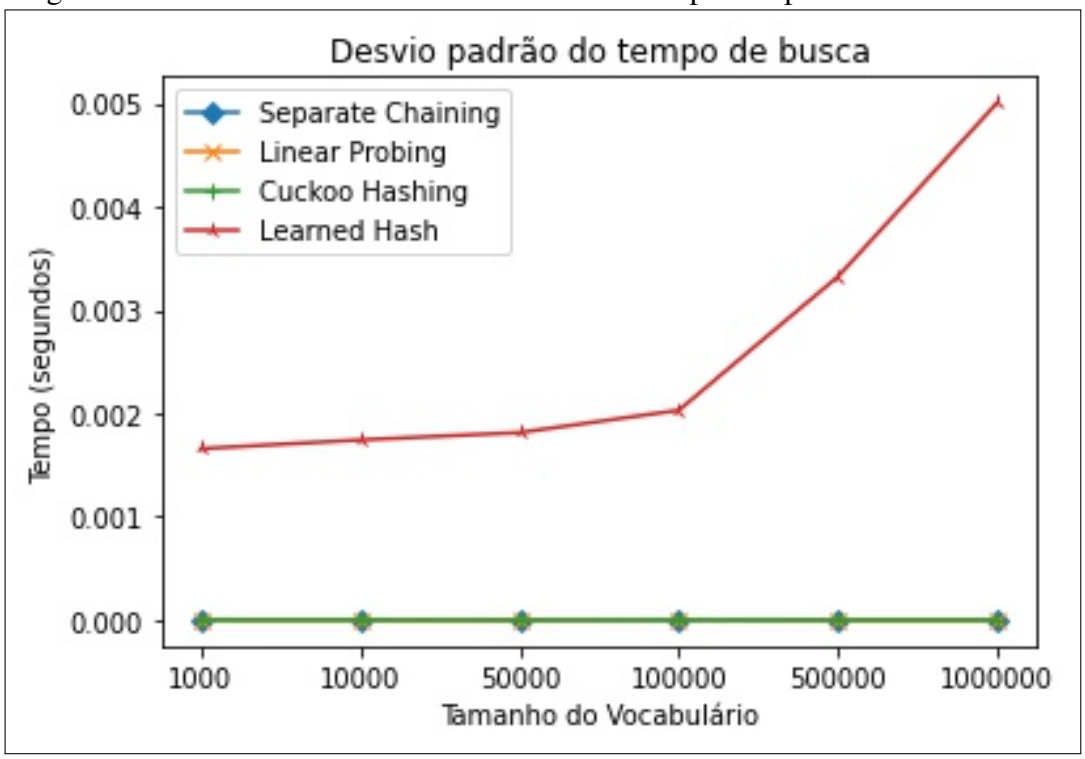
Fonte: Próprio autor.

Figura 34 – Gráfico de linha considerando o desvio padrão para cima



Fonte: Próprio autor.

Figura 35 – Gráfico de linha considerando o desvio padrão para baixo

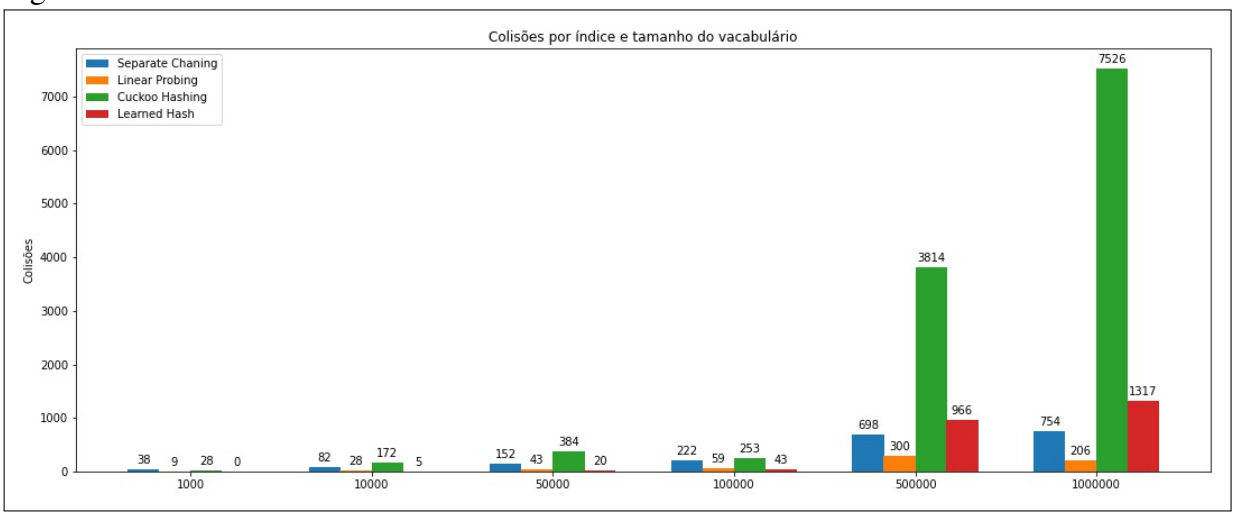


Fonte: Próprio autor.

5.4 Resultados dos testes com distribuição de frequência *zipf*

Esta métrica avaliadora de colisões foi única que mostrou resultados diferentes dos que vinha sendo apresentados nas seções anteriores. Percebe-se então que pelo gráfico apresentado na Figura 36, o Cuckoo Hashing que superava todas estruturas com grandes vocabulários acabou apresentando o pior resultado. Já o modelo de índice aprendido, apresentou bons resultados até ser superado e ficar em terceiro lugar a partir do vocabulário de tamanho quinhentos mil. A Sondagem Linear foi a que apresentou melhores resultados, seguido do Encadeamento Separado quando o vocabulário cresce.

Figura 36 – Gráfico de barras considerando as colisões em cada estrutura

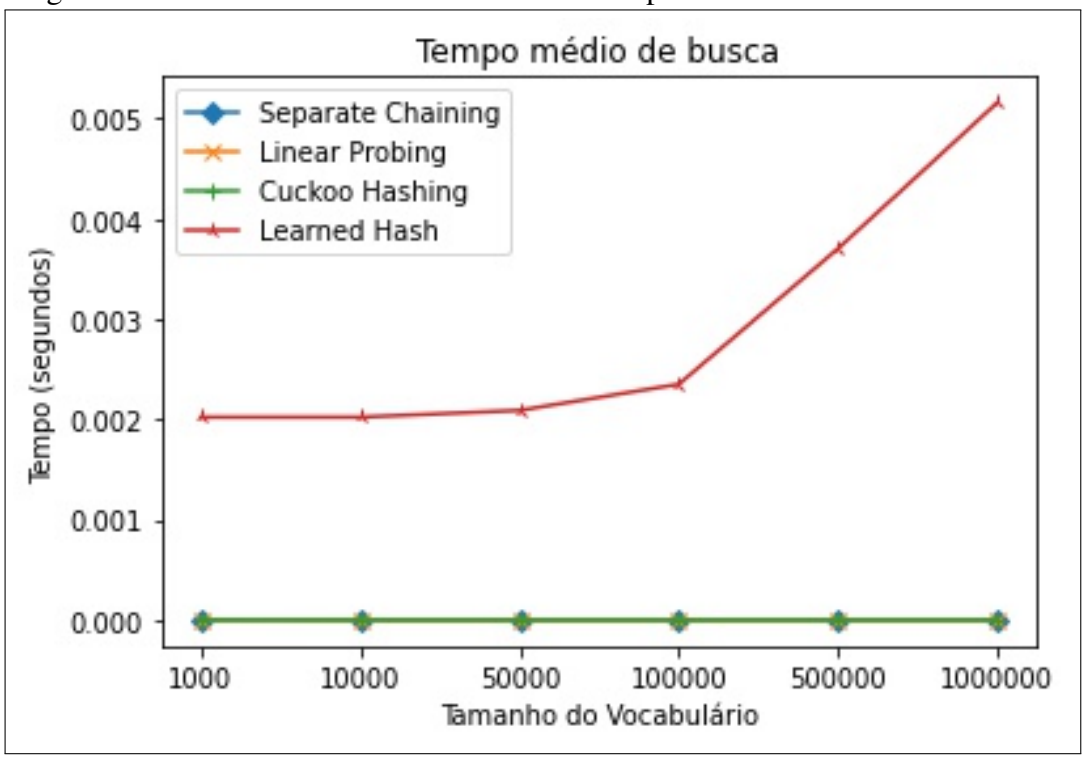


Fonte: Próprio autor.

Analisando a métrica de tempo de busca, percebe-se, através do gráfico apresentado na Figura 37 que o modelo de índice aprendido o tempo de busca aumenta a medida que o vocabulário cresce mostrando-se totalmente desfavorável e mantendo assim sua semelhança de tempo, inclusive com o desvio padrão, para todos os tipos de distribuições de dados apresentadas aqui.

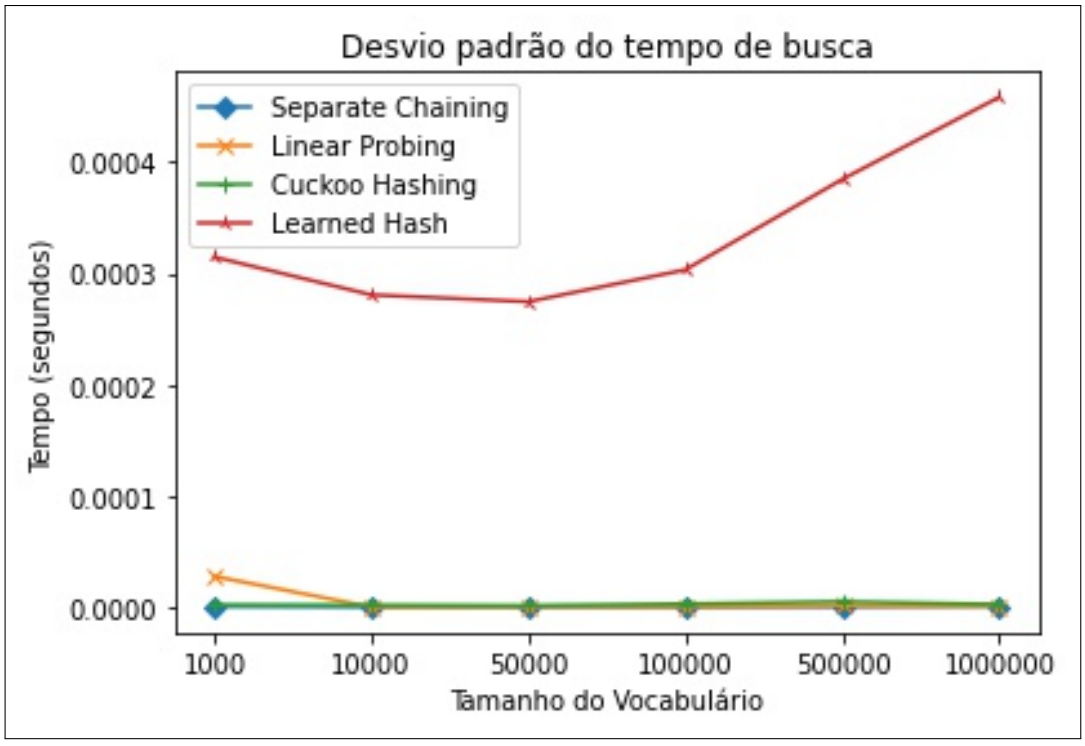
Ressalta-se ainda que, embora a estrutura de Sondagem Linear tenha obtido o melhor resultado final é possível observar que seu desvio padrão diverge consideravelmente em relação as outras duas estruturas tradicionais Cuckoo Hashing e Encadeamento Separado no menor tamanho de vocabulário. Isto é apresentado através do gráfico da Figura 38. Já os desvios padrões para cima e para baixo, Figuras 39 e 40, equiparam-se aos já apresentados.

Figura 37 – Gráfico de linha considerando o tempo de busca



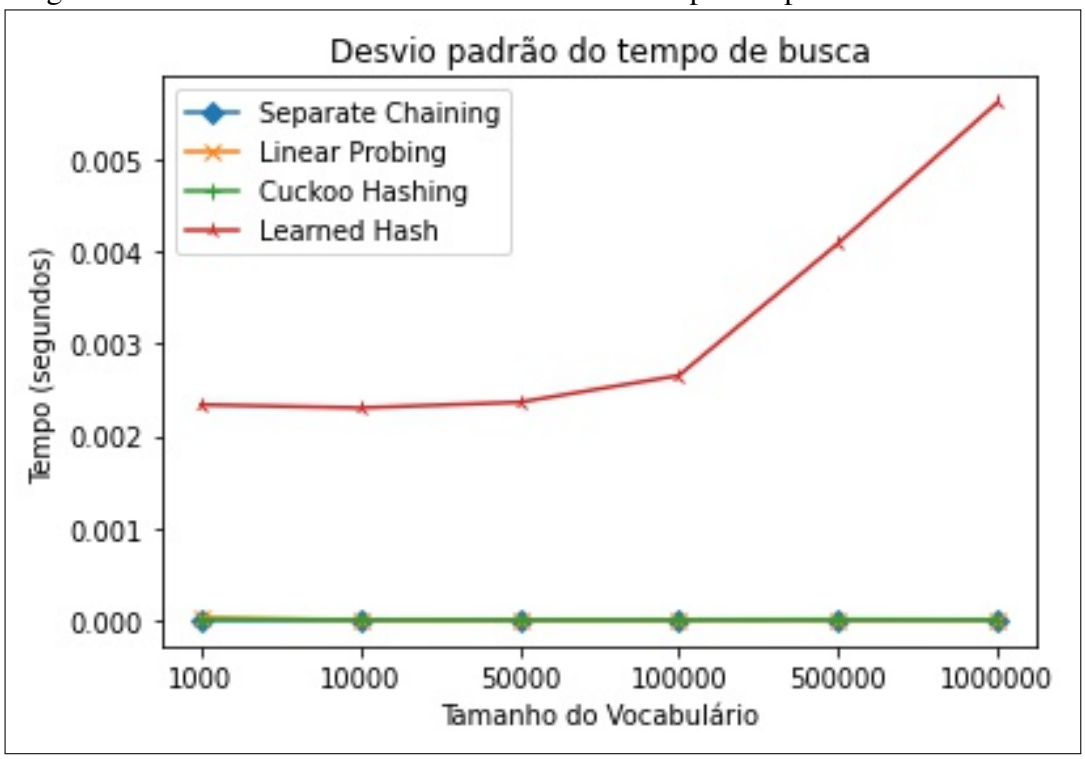
Fonte: Próprio autor.

Figura 38 – Gráfico de linha considerando o desvio padrão



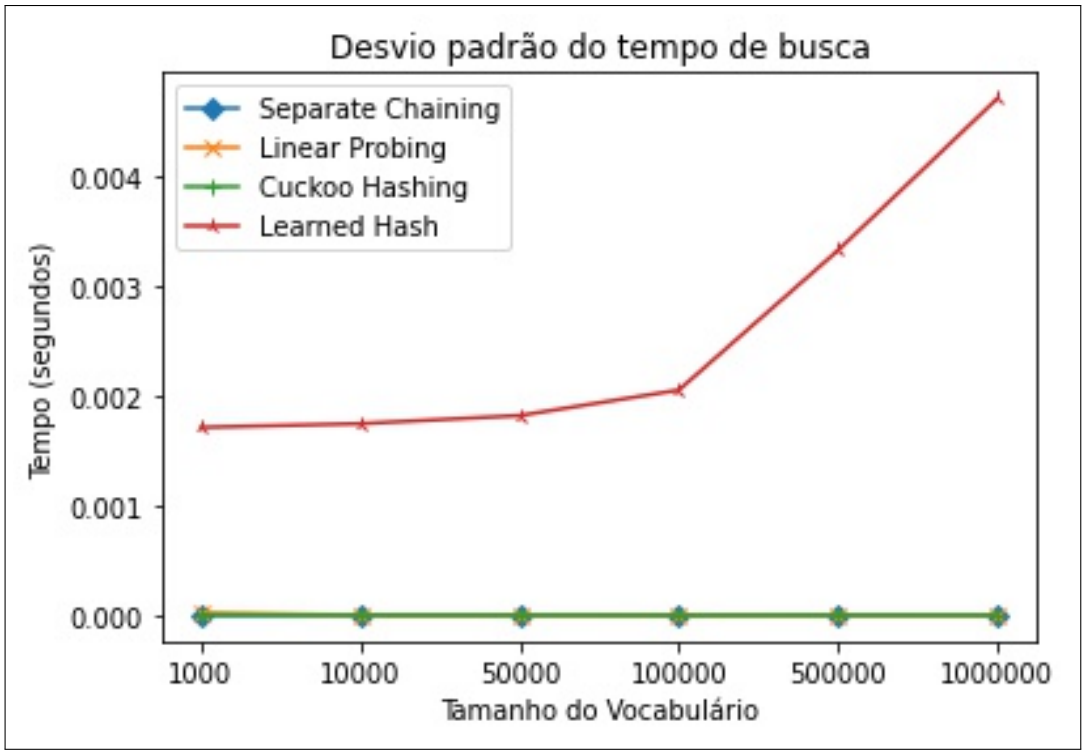
Fonte: Próprio autor.

Figura 39 – Gráfico de linha considerando o desvio padrão para cima



Fonte: Próprio autor.

Figura 40 – Gráfico de linha considerando o desvio padrão para baixo



Fonte: Próprio autor.

5.5 Resumo dos resultados

Tendo em vista a análise dos resultados obtidos, pode-se comparar qual a estrutura, que de modo geral, predominou os melhores resultados. O modelo de índice, apesar de ser superado nas colisões pelo Cuckoo Hashing em alguns momentos, prevalece dominante para vocabulários pequenos independentemente do tipo de implementação da Tabela Hash. A tabela a seguir mostra qual estrutura foi melhor por métricas de avaliação.

Tabela 1 – Comparativo geral do melhor desempenho

	Tempo	Memória	Colisões
Tabela Hash	X	X	
Modelo de Índice Aprendido			X

Fonte: Próprio autor.

6 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho definiu um modelo de aprendizagem profunda, denominado aprendizagem de índice, com o objetivo de minimizar o problema da estrutura Tabela Hash. Diferente dos outros modelos apresentados neste trabalho, a abordagem de aprendizagem de índice utilizou um modelo de Rede Neural Recorrente para determinar as posições de chaves do tipo String. A contribuição deste trabalho foi, além de desenvolver um modelo de rede neural, compará-la com outras estruturas tradicionais de índices existentes na fundamentação teórica simulando um banco de dados chave-valor utilizando diferentes distribuições de dados aleatórios.

A utilização de distribuições de dados aleatórios e os diferentes tamanhos de vocabulários utilizados dão a garantia e segurança dos resultados obtidos. A analise desses resultados demostra que o presente trabalho alcançou parcialmente seu objetivo principal ao minimizar a quantidade de colisões. O modelo proposto minimizou as colisões mais que todas as outras estruturas para vocabulários pequenos. Para as distribuições de dados normal e uniforme o modelo foi superado somente pelo Cuckoo Hashing. Porém, com a distribuição de dados $zipf$ o modelo ficou em terceiro lugar e em último o Cuckoo Hashing apresentou pior desempenho ao apresentar muitas colisões. Contudo, vale ressaltar, que embora a analise das colisões do modelo proposto tenham tido resultados satisfatórios o tempo de pesquisa deixou a desejar assim como o consumo de memória, tal resultado se deve pelo fato de o modelo de rede neural usar camadas densas e muitas matrizes. Ainda com relação ao tempo, além da limitação de hardware tida no ambiente de execução a API tensorflow apresenta limitação com carga baixa, por exemplo, quando executado com muito dados sua velocidade de inferência é maior. Mas como no experimento a inferência foi calculada com somente uma chave de cada vez, então sua velocidade foi mais lenta. Apesar disso, o modelo teve uma acurácia de 96% o que torna compreensível ele ter sido superado com as colisões em alguns momentos.

Como trabalhos futuros, podem ser realizados experimentos modificando os parâmetros do modelo assim como o uso de outras bibliotecas de aprendizado de máquina para constatar a limitação da API utilizada neste trabalho. Além disso, podem ser utilizada memória RAM e GPUs melhores para realizar o treinamento mais rápido da rede neural. Uma vez que o treinamento do modelo proposto tende a ficar mais lento quando ultrapassa a acurácia citada e a consumir muita memória com grandes quantidades de dados. Tudo isso para que se possa alcançar um menor tempo de predição além de reduzir drasticamente as colisões.

REFERÊNCIAS

ACADEMY, D. S. **Capítulo 8 - Função de Ativação - Deep Learning Book**. 2021. <https://www.deeplearningbook.com.br/funcao-de-ativacao/>. (Accessed on 06/07/2021).

BLACK, P. **Bresenham's Algorithm. Dictionary of Algorithms and Data Structures. US National Institute of Standards and Technology**. 2004.

CHANDRAMOULI, B.; PRASAAD, G.; KOSSMANN, D.; LEVANDOSKI, J.; HUNTER, J.; BARNETT, M. Faster: an embedded concurrent key-value store for state management. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 11, n. 12, p. 1930–1933, 2018.

CHUNG, J.; GULCEHRE, C.; CHO, K.; BENGIO, Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. **arXiv preprint arXiv:1412.3555**, 2014.

CHUNG, J.; GULCEHRE, C.; CHO, K.; BENGIO, Y. Gated feedback recurrent neural networks. In: **International Conference on Machine Learning**. [S.l.: s.n.], 2015. p. 2067–2075.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to algorithms**. [S.l.]: MIT press, 2009.

GRüBLER, M. **Entendendo o funcionamento de uma Rede Neural Artificial | by Murillo Grübler | aibrasil | Medium**. 2018. <https://medium.com/brasil-ai/entendendo-o-funcionamento-de-uma-rede-neural-artificial-4463fcf44dd0>. (Accessed on 11/06/2020).

HAYKIN, S. **Redes neurais: princípios e prática**. [S.l.]: Bookman Editora, 2007.

HAYKIN, S. S.; HAYKIN, S. S.; HAYKIN, S. S.; ELEKTROINGENIEUR, K.; HAYKIN, S. S. **Neural networks and learning machines**. [S.l.]: Pearson education Upper Saddle River, 2009. v. 3.

KOVÁCS, Z. L. **Redes neurais artificiais**. [S.l.]: Editora Livraria da Fisica, 2002.

KRASKA, T.; ALIZADEH, M.; BEUTEL, A.; CHI, E. H.; DING, J.; KRISTO, A.; LECLERC, G.; MADDEN, S.; MAO, H.; NATHAN, V. Sagedb: A learned database system. 2019.

KRASKA, T.; BEUTEL, A.; CHI, E. H.; DEAN, J.; POLYZOTIS, N. The case for learned index structures. In: ACM. **Proceedings of the 2018 International Conference on Management of Data**. [S.l.], 2018. p. 489–504.

NIELSEN, M. A. **Neural networks and deep learning**. [S.l.]: Determination press San Francisco, CA, USA:, 2015. v. 25.

PAGH, R.; RODLER, F. F. Cuckoo hashing. **Journal of Algorithms**, Elsevier, v. 51, n. 2, p. 122–144, 2004.

SAK, H.; SENIOR, A.; BEAUFAYS, F. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In: **Fifteenth annual conference of the international speech communication association**. [S.l.: s.n.], 2014.

SEDGEWICK, R.; WAYNE, K. **Algorithms (4a. ediçao)**. [S.l.]: Addison-Wesley, 2011.

SIQUEIRA, F. d. **2 - Banco de Dados - Professor Fernando De Siqueira - Banco de Dados I**. 2019. Disponível em <https://sites.google.com/site/uniplibancodedados1/aulas/aula-2---banco-de-dados>. Acesso em: 04 de agosto de 2019.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de Dados e seus Algoritmos**. [S.l.]: Livros Técnicos e Científicos, 1994. v. 2.

TSCHEEPERS, T. **Tscheepers/wikipedia-summary-dataset: This dataset contains all titles and summaries (or introductions) of English Wikipedia articles, extracted in September of 2017. it could be useful if one wants to use the smaller, more concise, and more definitional summaries in their research. or if one just wants to use a smaller but still diverse dataset for efficient training with resource constraints.** 2017. Disponível em <https://github.com/tscheepers/Wikipedia-Summary-Dataset> Acesso em: 02 de julho de 2021.

WEISS, M. A. **Data structures and algorithm analysis in Java**. [S.l.]: Addison-Wesley Reading, MA, USA, 2012. v. 3.

WIETING, J.; BANSAL, M.; GIMPEL, K.; LIVESCU, K. Charagram: Embedding words and sentences via character n-grams. **arXiv preprint arXiv:1607.02789**, 2016.