Instituto de Pesquisas Tecnológicas do Estado de São Paulo

Marcelo de Rezende Martins

Aprendizagem de representação através do uso de redes neurais convolucionais na recuperação de trecho de código-fonte

São Paulo
2019

Sumário

Glossário

one-hot encoding *one-hot encoding* é um vetor esparso que contém:

- Um elemento cujo valor é definido como 1
- O restante dos elementos tem o valor definido como 0

One-hot enconding normalmente é utilizado para representar palavras e ou atributos que contém uma quantidade finita de valores (Google, 2019b). . iv, 6, 12, 13

aprendizagem de máquina sistema ou programa que constrói um modelo preditivo a partir de dados de entrada (Google, 2019b).. iv, v, vi

Colab É uma ferramenta de pesquisa e ensino para aprendizagem de máquina. É um ambiente Jupyter que não necessita configuração ou instalação Google (2019a).. iv, 37

docstring Em programação, um *docstring* é um texto especificado no código-fonte que é usado para documentar um trecho específico do código Wikipedia (2019a).. iv, 15

git git é um versionador de controle distribuído para rastrear alterações no código-fonte durante o desenvolvimento de software. Endereço do site: https://git-scm.com/ (Wikipedia, 2019b). iv, v

GitHub GitHub é uma plataforma de hospedagem de código-fonte com controle de versão usando o git. Endereço do site: https://www.github.com/. iv

Jupyter *Jupyter Notebook* é uma ferramenta interativa que permite desevolver, executar e documentar código em uma aplicação web. O termo *notebook* refere-se a um caderno de anotações, pois é possível desenvolver, salvar as saídas do programa e fazer anotações Jupyter (2019).. iv, v

mecanismo de atenção Mecanismo utilizado comumente na tarefa de tradução por aprendizagem de máquina. Este método lê uma sentença inteira de entrada e gera uma palavra traduzida por vez. E para cada momento que uma palavra é traduzida, o mecanismo de atenção foca em partes diferentes da sentença de entrada. (Goodfellow *et al.*, 2016a).. iv, 13

modelo Representação do que um sistema de aprendizagem de máquina aprendeu a partir de dados de treinamento (Google, 2019b).. iv, 2

neurônio Um neurônio é um nó numa rede neural, que tipicamente recebe múltiplos valores de entrada e gera um valor de resultado. O neurônio aplica uma função de ativação (transformação não-linear) na soma dos valores de entrada com seus respectivos pesos (Google, 2019b).. iv, vi

representação distribuída Representação distribuída significa uma relação de muitos para muitos entre dois tipos de representações (por exemplo, conceitos e neurônios) (Hinton *et al.*, 1986).

- Cada conceito é representado por muitos neurônios
- Cada neurônio participa na representação de muitos conceitos

. iv, 7

Stack Overflow Site de perguntas e respostas de programação. Endereço do site: https://www.stackoverflow.com/. iv, 26, 29, 30

Siglas

CBoW Comsuption Bag of Words. iv, 7

CNN Convolutional Neural Network. iv

GH GitHub. iv, 15

IDE Integrated Development Environment. iv

LSTM Long Short Term Memory. iv, 9, 10, 13

MRR Mean Reciprocal Rank. iv, 27, 29, 33

NLP Natural Language Processing. iv

RNN Recurrent Neural Network. iv, 9

SO Stack Overflow. iv, 14, 15

TFIDF Term Frequency–Inverse Document Frequency. iv, 13

VAE Variational AutoEncoder. iv, 13

VEM Workshop on Software Visualization, Evolution and Maintenance. iv, 18, 29

vGPU Virtual Graphics Processing Unit. iv, 37

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Recuperação de trecho de código-fonte ou *code retrieval* consiste em recuperar um trecho de código-fonte a partir de um repositório, de modo a atender as intenções do desenvolvedor, descritas em linguagem natural. A capacidade de recuperar trechos de código relevantes é uma importante ferramenta de produtividade. Sites como o Stack Overflow tornaram-se muito populares nos últimos anos devido a facilidade em buscar trechos de código-fonte a partir das questões expressas pelos usuários em linguagem natural (Cambronero *et al.*, 2019).

A busca por trecho de código em repositórios públicos tem sido um desafio. Atualmente, o site do Github, que hospeda milhares de projetos de código-aberto, não possui um mecanismo de pesquisa que seja capaz de recuperar um trecho de código-fonte relevante a partir dos termos utilizados no campo de busca (Cambronero *et al.*, 2019). Isto não é somente um desafio para repositórios públicos, mas para repositórios privados também. Os repositórios privados adicionam um desafio a mais, já que os desenvolvedores, muitas vezes, não podem se basear em trechos de código encontrados no Google ou Stack Overflow, pois eles podem não atender aos requisitos específicos da organização de API e uso de bibliotecas (Cambronero *et al.*, 2019).

Conforme Cambronero *et al.* (2019), muitos trabalhos tem sido feitos na academia e indústria em direção a uma busca mais avançada de código-fonte (Allamanis *et al.*, 2015; Cambronero *et al.*, 2019; Chen e Zhou, 2018; Gu *et al.*, 2018; Iyer *et al.*, 2016; Sachdev *et al.*, 2018; Yao *et al.*, 2018). E um ponto em comum nestes trabalhos é o uso de *deep learning*. Tanto que Cambronero *et al.* (2019) cunharam o termo *neural code search*, i.e.,

busca de código-fonte através do uso de redes neurais.

A principal abordagem utilizada nestes trabalhos é encontrar um modelo que seja capaz de correlacionar os trechos de código as intenções do usuário. Uma técnica comumente utilizada é a *joint embedding* ou agrupamento de representações distribuídas. Esta técnica mapeia dados de diferentes distribuições, modalidades para um mesmo espaço vetorial, de tal forma que conceitos similares ocupem regiões próximas neste espaço. Técnica bastante utilizada para associar textos e imagens, traduções e até mesmo problema de perguntas e respostas e recomendações (Lai *et al.*, 2018; Zhang *et al.*, 2019).

A representação dos dados heterogêneos na técnica joint embedding tem um papel importante. Uma boa representação deve ser capaz de auxiliar na aprendizagem de uma tarefa posterior (Goodfellow *et al.*, 2016d). No nosso caso, uma boa representação das intenções e dos trechos de código-fonte deve ser capaz de ajudar a encontrar um modelo que seja capaz de correcioná-los. Além disso, uma boa representação pode ser útil para dados que seguem uma outra distribuição. Uma representação de trechos de código-fonte que foram coletadas de um site de dúvidas de programação podem ser utilizadas para encontrar trechos de código-fonte similares em repositórios de código aberto.

A proposta deste trabalho é verificar o uso das redes convolucionais na aprendizagem de representação. Dado que as redes convolucionais apresentaram um bom desempenho em diversos problema NLP, atingindo resultados competitivos em relação a outras arquiteturas como RNN (Young *et al.*, 2017), queremos avaliá-las na recuperação de trecho de código-fonte. Mais especificamente, pretendemos avaliar se elas auxiliam na obtenção de um modelo capaz de correlacionar os trechos de código-fonte as intenções do desenvolvedor.

Um ponto importante a ressaltar é que ao optarmos por uma arquitetura CNN, estamos levando em consideração as suas características e limitações inerentes. Enquanto o RNN é capaz de fazer associações entre palavras muito distantes em um texto, o CNN prioriza interações locais. Ele tem dificuldades para extrair dependências de palavras muito distantes. E outro ponto é que o CNN é sensível a permutação, ele não é capaz de memorizar a localização exata de uma palavra em um texto (Goodfellow *et al.*, 2016b; Young *et al.*, 2017).

A partir destas considerações, as perguntas que este trabalho pretende responder são:

- Aprendizagem de representação através de uma arquitetura CNN auxilia na recuperação de trecho de código-fonte?
- Será que o CNN é capaz de extrair as características latentes e mais importantes de modo a facilitar o modelo a encontrar uma correlação entre os trechos de código-fonte e as intenções expressas em linguagem natural?

Indiretamente, dado que o CNN prioriza interações locais, estaremos respondendo também a seguinte pergunta:

- As interações locais auxiliam na aproximação das intenções aos trechos de código-fonte?

Para respondê-las, avaliaremos um modelo que busca aproximar as intenções aos trechos de código-fonte. Para isto, o modelo vai ser estimulado a aproximar as suas representações. Estas representações serão obtidas a partir da nossa arquitetura CNN. A avaliação final do modelo será feita em um conjunto de dados composto por pares de questões e trechos de código-fonte coletados do site Stack Overflow.

1.1 Considerações Preliminares

O foco deste trabalho é a recuperação de trecho de código-fonte. Diferentemente dos trabalhos de Iyer *et al.* (2016) e Allamanis *et al.* (2015) que abordaram também o problema de sumarização de código-fonte ou *code summarization*. Para podermos avaliar a nossa arquitetura, iremos compará-la com outras arquiteturas de referência e com a arquitetura proposta por Cambronero *et al.* (2019) que é o estado da arte atualmente. Em relação aos dados, a base de dados disponibilizada por Yao *et al.* (2018) contém pares de questões e trechos de código-fonte em Python e SQL. A princípio, utilizaremos apenas os pares de perguntas e código-fonte em Python.

1.2 Objetivos

O objetivo principal deste trabalho é propor uma nova abordagem para a recuperação de trecho de código-fonte. Queremos avaliar se o uso de redes convolucionais auxiliam na obtenção de um modelo que seja capaz de aproximar os trechos de código-fonte as intenções do desenvolvedor.

1.3 Contribuições

Dentre as contribuições deste trabalho estão:

- Proposta de uma nova abordagem para a recuperação de trecho de código-fonte;
- Avaliação de uma nova arquitetura;
- Avaliação do modelo utilizando uma nova base de dados disponibilizada por Yao *et al.* (2018);
- Disponibilização do código-fonte, desde o pré-processamento até a avaliação final, em um repositório público;
- Publicação dos resultados através de um artigo científico

1.4 Organização do Trabalho

No Capítulo 2, apresentamos os conceitos e trabalhos relacionados a recuperação de trecho de código-fonte. No Capítulo 3, apresentamos a nossa abordagem para o item de pesquisa proposto. No Capítulo 4, exibimos os resultados preliminares da arquitetura proposta utilizando os dados de Yao *et al.* (2018). Já no Capítulo 5 temos os próximos passos e o cronograma até a entrega do trabalho.

Capítulo 2

Fundamentação teórica

De acordo com Chen e Zhou (2018), a tarefa do *code retrieval* consiste em:

Dado uma descrição em linguagem natural, recuperar o trecho de código-fonte mais relevante, tal que os desenvolvedores possam encontrar rapidamente os trechos de código que atendam as suas necessidades.

Já Cambronero *et al.* (2019) utiliza um outro termo para referir-se a *code retrieval*, é o termo *code search*. Para ele, o objetivo do *code search* é recuperar um trecho de código-fonte a partir de um enorme repositório de código-fonte, que mais se aproxima da intenção do desenvolvedor, expressa em linguagem natural. Ambos os termos, a nosso entendimento, referem-se a mesma tarefa. Neste trabalho, utilizaremos o termo *code retrieval* cunhado por Chen e Zhou (2018) para referir-se a recuperação de trecho de código-fonte.

Conforme Cambronero *et al.* (2019), recuperação de trecho de código-fonte consiste em recuperar o trecho de código que mais se aproxima da intenção do usuário expressa em linguagem natural. No nosso caso, as intenções expressas em linguagem natural serão títulos de questões coletadas do site Stack Overflow. Estas questões coletadas tem uma característica importante, elas são do tipo *how-to-do-it*. Estes tipos de questões expressam a intenção do usuário. A partir disso, quando fizermos referência a palavra *questão*, estaremos nos referindo a intenção do desenvolvedor expressa em linguagem natural.

2.1 Definição

Para formalizarmos a tarefa de *code retrieval*, adotaremos a mesma definição proposta por Iyer *et al.* (2016) e Yao *et al.* (2019):

Seja $\mathbb{Q}$ um conjunto formado por questões e/ou intenções descritas em linguagem natural. Seja $\mathbb{C}$ um conjunto de trechos de códigos-fontes. Dado uma questão em linguagem natural $q \in \mathbb{Q}$ e um conjunto com os trechos de código-fonte candidatos $\mathbb{C}_a \subset \mathbb{C}$. O objetivo do *code retrieval* é recuperar o trecho de código $c^+ \in \mathbb{C}_a$, onde c^+ é a resposta anotada como correta a dada questão. Formalmente, para um corpus de treinamento composto por pares de questões e trechos de código-fonte, *code retrieval* é definido como:

Code Retrieval: Dada uma questão em linguagem natural $q \in \mathbb{Q}$, um modelo F_r será treinado a recuperar $c^+ \in \mathbb{C}_a$ com a maior pontuação:

$$c^+ = \underset{c \in \mathbb{C}_a}{argmax}\ F_r(q, c) \tag{2.1}$$

2.2 *Joint Embeddings*

Segundo Goodfellow *et al.* (2016d), um fator importante a ser levado em consideração é a forma como os dados são representados. Uma boa representação consegue identificar as características relevantes e os fatores causadores dos dados observados. Além disso, uma boa representação deve ser capaz de facilitar na aprendizagem de uma tarefa posterior.

Dado que o nosso conjunto de dados é formado por pares de questões e trechos de código-fonte coletados do Stack Overflow. E cada questão e trecho de código-fonte é composto por uma sequência de palavras e símbolos. Uma boa representação para as palavras, que compõem eles, é um vetor de representação distribuída. Segundo Goodfellow *et al.* (2016d), as redes neurais generalizam bem quando as palavras são representadas através de vetores de representação distribuída em comparação a um *one-hot encoding*. Representação distribuída induz a um rico espaço de similaridade, no qual con-

ceitos semanticamente similares estão próximos, uma propriedade que não está presente em representações puramente simbólicas.

A representação é um fator importante para a tarefa de *code retrieval*. Pois com uma boa representação, um modelo de redes neurais será capaz de aproximar os trechos de código-fonte as intenções do desenvolvedor mais facilmente. Três aspectos devem ser levados em consideração ao fazer esta aproximação. O primeiro aspecto é a representação dos *tokens* ou palavras que compõem as intenções e os trechos de códigos-fontes. O segundo é a obtenção de uma representação para as intenções e trechos de códigos. E por último, como fazer a associação entre eles.

2.2.1 *Representação dos tokens ou palavras*

As questões e os trechos de códigos-fontes são compostas por um conjunto de palavras ou *tokens*. E conforme citado anteriormente, uma boa representação para as palavras é um vetor de representação distribuída. No caso, o vetor de representação distribuída ou *embedding* permite representar uma palavra através de um vetor com valores reais ($\mathbb{R}$). Podemos definir **embedding** como uma função $\mathbb{E}$ (Cambronero *et al.*, 2019):

$$\mathbb{E} : \mathbb{X} \rightarrow \mathbb{R}^d \tag{2.2}$$

A função $\mathbb{E}$ mapeia um dado de entrada x, $x \in \mathbb{X}$, para um vetor de representação distribuída correspondente em um espaço vetorial de dimensão d. O vetor de representação distribuída utiliza a hipótese distribucional, no qual palavras que estão próximas uma das outras em vários contextos tem significados similares (Goodfellow *et al.*, 2016d). Uma opção para a função $\mathbb{E}$ é utilizar o algoritmo não-supervisionado *word2vec*. O *word2vec* utiliza o algoritmo *skip-gram* ou CBoW. A diferença basicamente entre eles é que o CBoW prediz uma palavra de acordo com as palavras do contexto e o *skip-gram* faz o inverso, prediz as palavras do contexto dado uma palavra alvo (Mikolov *et al.*, 2013).

Por exemplo, dado o trecho de código-fonte abaixo:

Python: Exemplo de código para escrever em um arquivo

```
with open(filename, mode) as file:
    file.write(text)
```

Removendo os acentos, tabulações, quebra de linha e as pontuações, o código-fonte pode ser representado por um vetor formado pela seguinte sequência de palavras:

Trecho de código-fonte representado através de vetor de *tokens*.

['with', 'open', 'filename', 'mode', 'as', 'file', 'file', 'write', 'text']

Neste caso, ao utilizar o *skip-gram* do *wordvec* para a palavra *file*. Ele representará a palavra *file* próxima das palavras *mode*, *as*, *file*, *write*, caso o parâmetro *window* seja 2. O parâmetro *window* define basicamente quantas palavras a direita e a esquerda fazem parte do contexto.

Palavra *file*, em amarelo, e as palavras similares, em verde.

['with', 'open', 'filename', 'mode', 'as', 'file', 'file', 'write', 'text']

O algoritmo *skip-gram* mapeia as palavras que estão perto uma das outras em diferentes contextos para regiões próximas no espaço vetorial $\mathbb{R}^d$. Lembrando que estas palavras são representadas por vetores de valores reais ($\mathbb{R}$). Os vetores que estão próximos um dos outros são considerados similares. No caso da aplicação do *word2vec* em trechos de código-fonte em Python, por exemplo, é possível verificar que a palavra *if* tem como palavras similares *elif* e *else*. Ou no caso do exemplo anterior, a palavra *file* terá como palavras similares *mode*, *as*, *write*. Isto ocorre pois este conjunto de palavras costumam aparecer próximas umas das outras em diferentes trechos de código-fonte. Devido

a isto, o *skip-gram* vai mapear os vetores que representam estas palavras para regiões próximas no espaço vetorial $\mathbb{R}^d$.

2.2.2 *Representação das sentenças e código-fonte*

O vetor de representação distribuída de cada palavra pode ser combinada para gerar um vetor de representação distribuída para a sentença inteira, e.g., podemos ter um vetor de representação distribuída para a questão inteira, combinando os vetores de cada palavra que fazem parte da questão. De acordo com Cambronero *et al.* (2019), duas possíveis abordagens podem ser utilizadas para combinar as palavras. Na primeira abordagem, podemos tratar o vetor de palavras como um *bag*. Neste caso, a ordem das palavras não importa. Por exemplo, o vetor abaixo:

Vetor de *tokens*.

['with', 'open', 'filename', 'mode', 'as', 'file', 'file', 'write', 'text']

Seria equivalente a seguinte permutação:

Vetor de *tokens* embaralhado.

['file', 'as', 'with', 'write', 'file', 'text', 'mode', 'open', 'filename']

Nesta abordagem, podemos utilizar uma rede neural que agrupa os tokens através de uma função de agrupamento que obtém o valor máximo, *maxpool*, por exemplo.

Uma outra abordagem é levar em consideração a ordem. Para isto, podemos utilizar o Long Short Term Memory (LSTM), uma variação do Recurrent Neural Network (RNN), para obter uma representação dos *tokens*. Os vetores de representação de cada palavra serão fornecidos um a um sequencialmente a rede neural. O termo *recorrente* do nome Recurrent Neural Network (RNN) refere-se a forma como é aplicada os mesmos passos a cada item da sequência de tal forma que a saída é dependente dos resultados e cálculos

prévios (Young *et al.*, 2017).

No caso do LSTM, por exemplo, dado uma sequência de vetores de representação de cada palavra, $\boldsymbol{x} = \{\boldsymbol{x}(1), \boldsymbol{x}(2), ..., \boldsymbol{x}(n)\}$, onde $\boldsymbol{x}(t)$ é um vetor de representação distribuída de dimensão d. $\boldsymbol{x}$ pode ser uma questão ou trecho de código-fonte, por exemplo. Podemos combinar cada palavra da questão ou trecho de código-fonte fornecendo-os sequencialmente para obter a combinação ou representação final através de um vetor $\boldsymbol{h}$. A cada passo uma palavra é fornecida como entrada a rede LSTM. No passo t, o valor do vetor de resultado $\boldsymbol{h}(t)$ (tamanho H) é:

$$i_t = \sigma(\boldsymbol{W}_i \boldsymbol{x}(t) + \boldsymbol{U}_i \boldsymbol{h}(t-1) + \boldsymbol{b}_i) \tag{2.3}$$

$$f_t = \sigma(\boldsymbol{W}_f \boldsymbol{x}(t) + \boldsymbol{U}_f \boldsymbol{h}(t-1) + \boldsymbol{b}_f) \tag{2.4}$$

$$o_t = \sigma(\boldsymbol{W}_o \boldsymbol{x}(t) + \boldsymbol{U}_o \boldsymbol{h}(t-1) + \boldsymbol{b}_o) \tag{2.5}$$

$$\bar{C}_t = tanh(\boldsymbol{W}_c \boldsymbol{x}(t) + \boldsymbol{U}_c \boldsymbol{h}(t-1) + \boldsymbol{b}_c) \tag{2.6}$$

$$C_t = i_t * \bar{C}_t + f_t * C_{t-1} \tag{2.7}$$

$$\boldsymbol{h}_t = o_t * tanh(C_t) \tag{2.8}$$

A arquitetura do LSTM tem três portas (porta de entrada i, porta *forget* f e porta *output* o), um vetor de célula de memória C, σ é a função sigmóide. A porta de entrada determina como os vetores x_t alteram o estado da célula de memória. A porta *output* permite a célula de memória influenciar os resultados finais. Por fim, a porta *forget* é um mecanismo que permite a célula de memória esquecer ou lembrar os estados anteriores. $\boldsymbol{W} \in \mathbb{R}^{HXd}$, $\boldsymbol{U} \in \mathbb{R}^{HXH}$ e $\boldsymbol{b} \in \mathbb{R}^{HX1}$ são as matrizes de pesos a serem aprendidos (Tan *et al.*, 2015).

O vetor *hidden* $\boldsymbol{h}$ é apontado comumemente como a parte mais importante do LSTM.

Pois ele engloba o resultado dos cálculos e, implicitamente, através da célula de mémoria C, os resultados prévios. Normalmente, ele é utilizado para representar os dados (Young *et al.*, 2017). No nosso caso, para representar uma questão ou trecho de código-fonte, podemos calcular a média dos vetores de resultado $\boldsymbol{h} = \{\boldsymbol{h}_1, \boldsymbol{h}_2, ..., \boldsymbol{h}_t\}$ após t iterações, por exemplo.

2.2.3 Agrupamento de representações distribuídas

Com os vetores de representação das questões e trechos de código-fonte, o objetivo é encontrar um modelo que seja capaz de recuperar o trecho de código-fonte mais relevante para uma determinada questão. Uma abordagem comumente utilizada é mapear os vetores das questões e trechos de código-fonte para um mesmo espaço vetorial. E apróximá-los através de uma função objetivo, de tal maneira que as questões fiquem próximas dos trechos de código-fonte relevantes.

Nesta abordagem, o objetivo é encontrar um modelo que consiga correlacionar dados heterogêneos. No livro Goodfellow *et al.* (2016d), a associação entre as representações é referida como agrupamento de distribuições ou *joint distribution*. Já Cambronero *et al.* (2019) e Allamanis *et al.* (2015) utilizam o termo *bi-modal embedding*. Zhang *et al.* (2019) utilizam *jointly model*. E Gu *et al.* (2018) utilizam o temo agrupamento de representação distribuída ou *joint embedding*. Neste trabalho, utilizaremos o termo *joint embedding* ou agrupamento de representação distribuída.

Joint embedding, ou agrupamento de representação distribuída, é uma técnica para mapear dados heterogêneos para um mesmo espaço vetorial, de tal forma que conceitos similares ocupem regiões próximas neste espaço (Gu *et al.*, 2018). Esta técnica é comumente utilizada para encontrar modelos que correlacionem imagem e texto, tradução de textos e também é utilizada em problemas de perguntas e respostas e recomendações de conteúdo (Lai *et al.*, 2018; Zhang *et al.*, 2019).

Por exemplo, sejam $\mathbb{Q}$ e $\mathbb{C}$ conjuntos de dados heterogêneos. *Joint embedding* pode

ser formulado como:

$$f : q \rightarrow t_q \rightarrow h_\theta(t_q, t_c) \leftarrow t_c \leftarrow c : g \tag{2.9}$$

A função f, $f : q \rightarrow \mathbb{R}^d$, mapeia o vetor $q \in \mathbb{Q}$ para um espaço vetorial $\mathbb{R}$ de dimensão *d*. A função g, $g : c \rightarrow \mathbb{R}^d$, mapeia o vetor $c \in \mathbb{C}$ para o mesmo espaço vetorial $\mathbb{R}$. A função h_θ calcula a similaridade entre dois vetores do mesmo espaço vetorial.

O objetivo é aprender as funções f e g tal que para uma função de similaridade h_θ, e.g., *cosine*, e vetores $t_q \in \mathbb{R}^d$ e $t_c \in \mathbb{R}^d$. A função $h_\theta(t_q, t_c)$ seja maximizada para t_q e seu correspondente vetor t_c. Através do *joint embedding*, os vetores t_q e t_c podem ser correlacionados (Cambronero *et al.*, 2019; Gu *et al.*, 2018).

2.3 Trabalhos relacionados

Conforme Cambronero *et al.* (2019), uma boa parte dos modelos propostos para o *code retrieval* utilizam a abordagem *joint embedding*. Compilamos os principais trabalhos que utilizam esta abordagem na Tabela 2.1. Para compilar estes artigos, inicialmente, fizemos uso das referências apontadas nos trabalhos de Allamanis *et al.* e Yao *et al.*. Posteriormente, realizamos buscas nas bases Scopus, Google Scholar e IEEE com as palavras-chaves *neural network*, *code retrieval* e *code search*. Após as leituras dos títulos, resumos e palavras-chaves, mantivemos os artigos que vão ao encontro com a nossa definição de *code retrieval*.

Conforme a Tabela 2.1, a maioria dos trabalhos representaram as palavras através de vetores de representação distribuída. Com exceção do Iyer *et al.* (2016) que optou por representar com *one-hot encoding*. Já em relação a combinação das palavras, i.e., representação da sentença os trabalhos de Gu *et al.* (2018) e Iyer *et al.* (2016) optaram pelo uso de redes neurais recorrentes, levando em consideração a ordem das palavras. Já Allamanis *et al.* (2015) optou por representar as questões como um *bag* de palavras, enquanto para os trechos de código-fonte optou-se por representar através de uma árvore sintática obtida através do compilador. Neste caso, a relação hierárquiva entre as palavras

Artigo	Sumário
Allamanis *et al.* (2015)	Representação distribuída para as questões / Árvore sintática para o código-fonte Combinou os vetores de *tokens* de questão através da média / Combinou os *tokens* de código-fonte usando operações multiplicativas
Chen e Zhou (2018)	Representação distribuída para questão e código-fonte Combinou os vetores de representação distribuída usando VAE
Iyer *et al.* (2016)	*one-hot encoding* para questão e código-fonte Combinou os vetores usando LSTM com mecanismo de atenção
Gu *et al.* (2018)	*word2vec* para questão e código-fonte Combinou os vetores usando uma rede bi-LSTM
(Sachdev *et al.*, 2018)	*word2vec* para código-fonte e questão Combinou os vetores de *tokens* da questão usando a média / Combinou os vetores do código-fonte através do TFIDF
Cambronero *et al.* (2019)	*word2vec* para questões e código-fonte Combinou os vetores de cada palavra da questão calculando a média / Combinou os vetores do código-fonte usando o mecanismo de atenção

Tabela 2.1: *Sumário das diferentes abordagens adotadas pelos pesquisadores para o problema do code retrieval. Tabela adaptada de Cambronero et al. (2019)*

foi levada em consideração. Os trabalhos de Chen e Zhou (2018), Sachdev *et al.* (2018) e Cambronero *et al.* (2019) optaram por uma abordagem mais simples e trataram o vetor de tokens como um *bag*.

É interessante observar que nenhum trabalho fez uso das redes convolucionais na recuperação de trecho de código-fonte. Sendo que as redes convolucionais apresentaram um bom desempenho em problemas similares em NLP (Feng *et al.*, 2015; Lai *et al.*, 2018; Tan *et al.*, 2015). Yao *et al.* fez uma menção sobre a possibilidade do uso de CNN na tarefa de *code retrieval*.

Para avaliação do modelo, há uma diferença, basicamente, no corpus de busca. Corpus de busca entende-se como um conjunto formado por trechos de código-fonte no qual o modelo ou o sistema irá utilizar para recuperar o trecho de código mais relevante para uma questão. Iyer *et al.* (2016) e Chen e Zhou (2018) utilizaram um corpus de busca composto pelo trecho de código-fonte apontado como correto e outros 49 distratores para

avaliação. Já Gu *et al.* (2018), Sachdev *et al.* (2018) e Cambronero *et al.* (2019) utilizaram repositórios do Github.

		Treinamento	Avaliação	
Artigo	**Linguagem**	**# de dados**	**# questões anotadas**	**Corpus de busca**
Allamanis *et al.* (2015)	C#	24.812	N/D	50
Chen e Zhou (2018)	C# / SQL	66.015 / 25671	100	50
Iyer *et al.* (2016)	Java	16 milhões	50	4 milhões
Gu *et al.* (2018)				
Sachdev *et al.* (2018)	Java (Android)	787 mil	100	5,5 milhões
Cambronero *et al.* (2019)	Java / Java (Android)	16 milhões / 787 mil	50 / 287	4 milhões / 5,5 milhões

Tabela 2.2: *Relação da quantidade de dados utilizada para treinamento e avaliação dos modelos. A coluna* ***# questões anotadas*** *refere-se a quantidade de questões anotadas manualmente para avaliação final do modelo.*

De acordo com a Tabela 2.2, com exceção de Allamanis *et al.* (2015), os outros trabalhos avaliaram o modelo utilizando questões coletadas e anotadas manualmente. Uma análise e verificação manual das respostas feitas pelos modelos foi relizada na maioria dos trabalhos, exceto nos trabalhos de Allamanis *et al.* (2015) e Cambronero *et al.* (2019). No caso do Cambronero *et al.* (2019) foi criado um sistema de avaliação automática, onde as respostas apontadas pelo modelo são consideradas corretas quando a diferença do resultado da função de similaridade h_θ entre elas e a resposta anotada como correta estão dentro de um certo intervalo.

Observamos que todos os trabalhos coletaram questões do Stack Overflow (SO) para avaliação final do modelo. Isto quer dizer que para avaliar se o modelo está recuperando o trecho de código-fonte que atenda a intenção do usuário, as intenções estão sendo expressas através de questões coletadas do Stack Overflow. Já em relação ao corpus de busca, i.e., onde o modelo vai realizar a busca do trecho de código-fonte, não há um consenso. Alguns trabalhos (Cambronero *et al.*, 2019; Gu *et al.*, 2018; Sachdev *et al.*,

2018) utilizam o GitHub (GH), enquanto outros trabalhos (Allamanis *et al.*, 2015; Chen e Zhou, 2018; Iyer *et al.*, 2016) utilizam trechos de código do Stack Overflow.

	Treinamento			Avaliação	
Artigo	**Fonte**	**Questões**	**Código-Fonte**	**Fonte das questões**	**Fonte do corpus de busca**
Allamanis *et al.* (2015)	SO	Título da postagem	Trecho de código-fonte	SO	SO
Chen e Zhou (2018)	SO	Título da postagem	Trecho de código-fonte	SO	SO
Iyer *et al.* (2016)	SO	Título da postagem	Trecho de código-fonte	SO	SO
Gu *et al.* (2018)	GH	Docstring	Método	SO	GH
Sachdev *et al.* (2018)	GH	N/D	Método	SO	GH
Cambronero *et al.* (2019)	GH / SO	Docstring / Título da postagem	Método / Trecho de código-fonte	SO	GH

Tabela 2.3: *Repositórios utilizados para coleta dos dados de treinamento e avaliação dos modelos no problema de code retrieval.*

A composição dos pares de questões e trechos de código-fonte para obtenção do modelo diferiu de acordo com a fonte utilizada para coleta dos dados de treinamento. Os trabalhos que utilizaram o GitHub (GH), utilizaram os métodos como trecho de código-fonte e o docstring deste método como descrição. No caso do Stack Overflow (SO), os pares foram formados pelo título da questão e pelo trecho de código-fonte da resposta aceita.

De acordo com Cambronero *et al.* (2019), os dados utilizados para compor os pares utilizados no treinamento do modelo, influenciaram no desempenho final. Em seu trabalho, foram observados um ganho no desempenho das redes neurais quando treinados com os títulos das questões e trechos de código-fonte do Stack Overflow (SO), em comparação aos pares formados por docstring e métodos extraídos dos códigos-fontes dos projetos do GitHub (GH). Além da composição dos pares, a variação das questões e tre-

chos de código-fonte é um fator importante também. Em um trabalho recente, Yao *et al.* (2018) coletou milhares de pares de perguntas e trechos de código-fonte do *StackOverFlow*. Eles treinaram o modelo proposto por Iyer *et al.* (2016) neste conjunto de dados e obtiveram um ganho de mais de 10% no desempenho final quando comparado com os dados originais.

Para Cambronero *et al.* (2019), o modelo consegue aproximar mais o trecho de código-fonte as intenções do usuário quando treinados com questões do *StackOverFlow*. E Yao *et al.* (2018) apontam para uma característica importante a ser observada nos dados, a variabilidade. A variabilidade auxilia na obtenção de um modelo mais robusto e que generaliza bem.

Conforme citado anteriormente e até onde o nosso conhecimento alcança, não encontramos trabalhos e artigos que verifiquem o desempenho das redes convolucionais na tarefa de *code retrieval*. A proposta deste trabalho é verificar o desempenho das redes convolucionais quando adicionada a uma rede neural recorrente e também isoladamente. Além disso, iremos comparar o nosso modelo com o modelo proposto por Cambronero *et al.* (2019) que é o estado da arte. E para treinamento do modelo, utilizaremos os dados coletados por Yao *et al.* (2018) devido aos resultados promissores apontados no artigo. E conforme Cambronero *et al.* (2019), as questões do *StackOverFlow* aproximam-se mais das intenções do usuário, indo ao encontro com a nossa definição de *code retrieval*.

Capítulo 3

Abordagem

Conforme citado no Capítulo 2, uma técnica muito utilizada para *code retrieval* é a *joint embedding*. Para utilizar esta técnica, três fatores devem ser levados em consideração:

- Representação de cada palavra de uma sentença ou trecho de código-fonte
- Representação da sentença e do trecho de código-fonte
- Função objetivo do modelo

A nossa proposta é criar um modelo que faz uso das redes convolucionais na aprendizagem de representação das questões e trechos de código-fonte. Porém, conforme levantado anteriormente, outros aspectos também devem ser levados em consideração. Como as palavras serão representadas e qual função objetivo utilizaremos para que o modelo induza a aproximação das questões aos trechos de código-fonte que são solução. Então, neste capítulo, apresentaremos a nossa abordagem para cada um destes itens.

3.1 Representação de cada palavra de uma sentença ou trecho de código-fonte

As questões e os trechos de código-fonte são compostos por palavras, pontuações, separadores e caracteres de símbolos e operadores matemáticos. Para as questões, que

são descritas em linguagem natural, iremos representá-las como um vetor formado por uma sequência de palavaras, apenas removendo os caracteres de acento e pontuação.

No caso do código-fonte, iremos partir da mesma hipótese que utilizamos no artigo de Rezende Martins e Gerosa (2019), apresentado no Workshop on Software Visualization, Evolution and Maintenance (VEM). Essa hipótese foi levantada por Allamanis *et al.* (2018) e diz que software é uma forma de comunicação humana e tem propriedades estatísticas similares a corpora de linguagem natural. A partir disto, iremos aplicar os mesmos procedimentos adotados para as questões aos trechos de código-fonte. Quer dizer, os trechos de código-fonte serão tratados como uma sequência de palavras e terão os acentos e caracteres especiais removidos.

Para cada palavra devemos definir uma representação. Uma boa representação para as palavras são os vetores de representação distribuída. Pois, segundo Goodfellow *et al.* (2016d), as redes neurais generalizam bem quando as palavras são representadas através de vetores de representação distribuída. E conforme os próprios autores, uma boa representação deve auxiliar na aprendizagem de uma tarefa posterior. No nosso caso, as representações devem auxiliar a tarefa de aprender a encontrar uma correlação entre as questões e os trechos de código-fonte mais relevantes.

Portanto, neste trabalho, cada palavra será representada através de um vetor de representação distribuída. Para mapeá-las, utilizaremos, inicialmente, o algoritmo *word2vec*. Ele será utilizado com *skip-gram*, que prediz as palavras do contexto a partir de uma palavra alvo. Segundo Mikolov *et al.* (2013), *skip-gram* obteve um desempenho melhor em problemas semânticos, e.g., relacionar uma palavra masculina com a equivalente feminina, relacionar o nome de uma capital a um país ou cidade a um estado. No caso do código-fonte, isto é uma característica importante, pois pode ajudar a agrupar os *tokens* de acordo com o tipo. Por exemplo, agrupar a instrução de decisão *while* próxima da instrução *for*. E relacionar a instrução de decisão *if* próxima da *else*.

No nosso caso, o *word2vec* irá mapear cada palavra para um espaço vetorial $\mathbb{R}^d$. Seja $\mathbb{Q}$ o conjunto formado pelas palavras das questões e $\mathbb{C}$ o conjunto constituído pelas palavras presentes nos trechos de código-fonte. Para cada palavra $q \in \mathbb{Q}$ e palavra $c \in \mathbb{C}$

teremos:

$$f : q \rightarrow t_q, t_q \in \mathbb{R}^d \tag{3.1}$$

$$g : c \rightarrow t_c, t_c \in \mathbb{R}^d \tag{3.2}$$

Onde f e g são o *word2vec* com o algoritmo *skip-gram*. f e g irão mapear cada palavra pertencente a um vocabulário a uma matriz distinta $\boldsymbol{T_q}$ e $\boldsymbol{T_c}$. Teremos duas matrizes: $\boldsymbol{T}_q^{|\mathbb{Q}|Xd}$ e outra $\boldsymbol{T}_c^{|\mathbb{C}|Xd}$, onde $|\mathbb{Q}|$ e $|\mathbb{C}|$ são o tamanho do vocabulário das questões e trechos de código-fonte, respectivamente. E d é a dimensão do vetor de representação distribuída.

Com estas duas matrizes $\boldsymbol{T}_q$ e $\boldsymbol{T}_c$, que contém as palavras dos vocabulários das questões e dos trechos de código-fonte mapeadas, podemos representar uma sentença através de vetores de representação distribuída. Por exemplo, dado uma questão $\boldsymbol{s} = \{w_1, w_2, ..., w_n\}$, onde $w_i, 1 \leq i \leq n$ é uma palavra. Podemos representar $\boldsymbol{s}$ através de um outro vetor $\boldsymbol{x}$, onde para cada palavra presente em $\boldsymbol{s}$, elas serão representadas através de um vetor de representação distribuída em $\boldsymbol{x}$. Para isto, basta recuperar o vetor $t_q \in \boldsymbol{T}_q$ correspondente a cada palavra presente em nossa questão. Logo, temos $\boldsymbol{x} = \{x(w_1), x(w_2), ..., x(w_n)\}$, tal que $x(w_i) \in \boldsymbol{T}_q$ e $\boldsymbol{x}(w_i) \in \mathbb{R}^d$ é um vetor de representação distribuída. Com o vetor $\boldsymbol{x}$, o próximo passo é combinar os seus elementos para obter uma representação da sentença.

3.2 Representação da sentença e do trecho de código-fonte

Com cada palavra representada por um vetor de representação distribuída, o próximo passo é combiná-las para obter uma representação para a questão e o trecho de código-fonte. Por exemplo, dado uma sentença $\boldsymbol{x} = \{\boldsymbol{x}(1), \boldsymbol{x}(2), ..., \boldsymbol{x}(n)\}$, onde $\boldsymbol{x}(i) \in \mathbb{R}^d$ um vetor de representação distribuída da i^{th} palavra da sentença, d é a dimensão do

vetor. Iremos combinar os elementos de $\boldsymbol{x}$ através das operações presentes em nossa arquitetura CNN. Basicamente são 3 operações:

- Operação de convolução
- Concatenação
- *Maxpool*

A operação de convolução utiliza um filtro convolucional $\boldsymbol{F} = [\boldsymbol{F}(1), \cdots, \boldsymbol{F}(m)]$, tal que $\boldsymbol{F} \in \mathbb{R}^{mXd}$. Este filtro é aplicado em uma janela de m palavras para produzir um novo vetor. Seja $\boldsymbol{x}(i, i+j)$ referência a concatenação dos vetores $\boldsymbol{x}(i), \boldsymbol{x}(i+1), ..., \boldsymbol{x}(i+j)$. . Ao aplicar $\boldsymbol{F}$ na seguinte janela de palavras $\boldsymbol{x}(i, i+m-1)$, um novo vetor $\boldsymbol{c}(i)$ é calculado da seguinte maneira:

$$\boldsymbol{c}(i) = tanh\left[\left(\sum_{j=i}^{i+m-1} \boldsymbol{x}(j)^T \boldsymbol{F}(j)\right) + b\right] \tag{3.3}$$

onde b é o *bias* e $\boldsymbol{F}$ e b são parâmetros do filtro. Para exemplificar esta operação, as figuras 3.1 e 3.2 abaixo ilustram um exemplo dos dois primeiros passos da aplicação de um filtro convolucional $\boldsymbol{F} \in \mathbb{R}^{mXd}$, onde $m = 2$ e $d = 5$. Este filtro é aplicado a uma sentença formado por 6 palavras, que é representado pelo vetor de representação distribuída $\boldsymbol{x} \in \mathbb{R}^{nXd}$, onde $n = 6$ e $d = 5$. Pelas figuras, podemos perceber que o parâmetro m do filtro $\boldsymbol{F}$ define como as palavras são combinadas. Neste caso, foram combinadas de duas em duas, um bi-gram.

Durante a operação de convolução, o filtro $\boldsymbol{F}$ é aplicado a todas possíveis janelas de palavras utilizando os mesmos pesos para produzir um mapa das características (*feature map*).

$$\boldsymbol{c} = \{\boldsymbol{c}(1), \boldsymbol{c}(2), ..., \boldsymbol{c}(n-m+1)\} \tag{3.4}$$

Em uma arquitetura CNN, existem centenas de filtros convolucionais $\boldsymbol{F}$, também chamados de *kernel*, de diferentes tamanhos m que percorrem toda uma sentença $\boldsymbol{x}$. Cada *kernel* extrai características específicas de n-gram. Após a operação de convolução, i.e.,

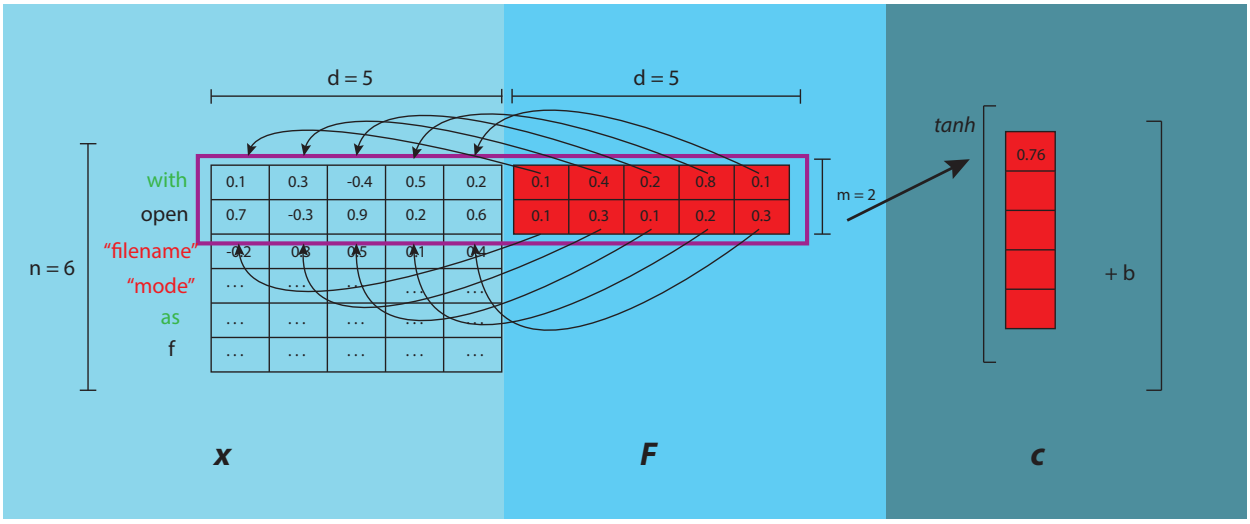


Figura 3.1: *Primeiro passo da operação de convolução. Aplica-se um filtro convolucional* $\boldsymbol{F} \in \mathbb{R}^{mXd}$*, onde* $m = 2$ *e* $d = 5$*, em uma sentença* $\boldsymbol{x} \in \mathbb{R}^{nXd}$*, onde* $n = 6$ *e* $d = 5$*. Este primeiro passo é equivalente a seguinte operação:* $\boldsymbol{c}(1) = tanh\left[\left(\sum_{j=1}^{2} \boldsymbol{x}(j)^T \boldsymbol{F}(j)\right) + b\right]$*. Figura adapatada a partir do texto de Kim (2019).*

o cálculo de $\boldsymbol{c}$, realizamos a concatenação dos vetores de saída dos diferentes filtros. Esta operação de concatenação é realizada apenas entre os filtros que tem diferentes tamanhos de janela, conforme ilustra a Figura 3.3. Então para k tamanhos de janelas diferentes, a operação de concatenação irá gerar um vetor $\boldsymbol{c_m}$ através da junção dos k vetores de saída diferentes.

$$\boldsymbol{c_m} = [\boldsymbol{c}_{m_1}, \boldsymbol{c}_{m_2}, ..., \boldsymbol{c}_{m_k}] \tag{3.5}$$

Onde $\boldsymbol{c_m}$ é o resultado da concatenação dos diferentes vetores $\boldsymbol{c}$, $\boldsymbol{m}$ é um vetor com os diferentes tamanhos de janela, $\boldsymbol{m} = \{m_1, m_2, ..., m_k\}$, e k é a quantidade de elementos. Somente ao final que aplicamos a camada *maxpool*. A camada *maxpool* aplica a operação *max*, obtendo o vetor de representação final da sentença $\boldsymbol{c'_m}$. A operação *max* é aplicada no eixo 0, conforme ilustra a Figura 3.3. Portanto o vetor final $\boldsymbol{c'_m}$ é obtido da seguinte maneira:

$$\boldsymbol{c'_m} = max\left([\boldsymbol{c}_{m_1}, \boldsymbol{c}_{m_2}, ..., \boldsymbol{c}_{m_k}], axis = 0\right) \tag{3.6}$$

Onde $axis$ define ao longo de qual eixo a função será aplicada. De forma equivalente,

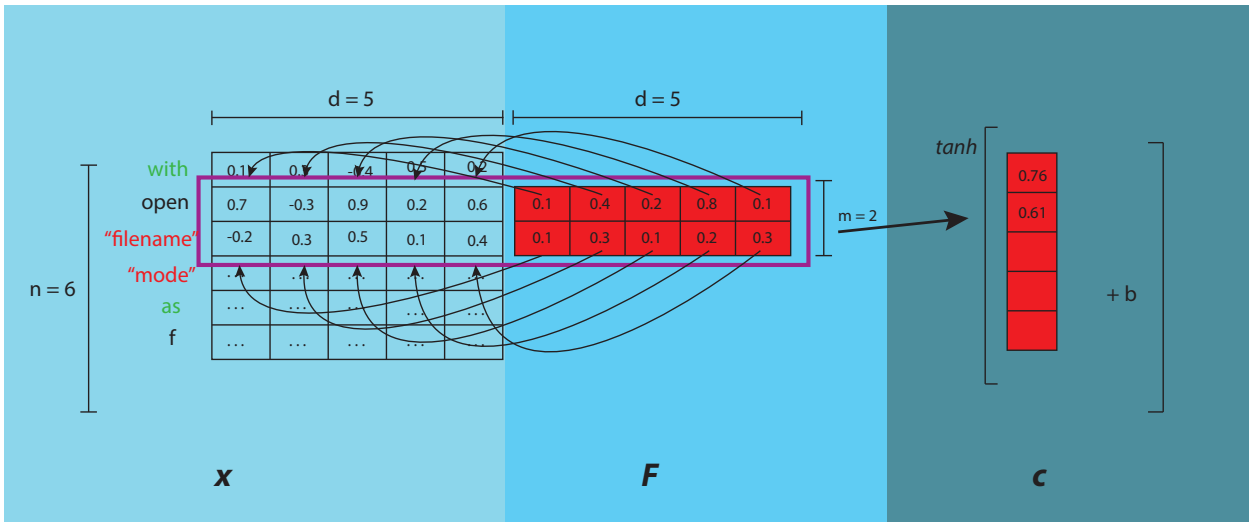


Figura 3.2: *Segundo passo da operação de convolução. Aplica-se o mesmo filtro convolucional* $\boldsymbol{F} \in \mathbb{R}^{mXd}$*, onde* $m = 2$ *e* $d = 5$*, em uma sentença* $\boldsymbol{x} \in \mathbb{R}^{nXd}$*, onde* $n = 6$ *e* $d = 5$*. Neste segundo passo, deslocamos uma posição no eixo* 0*. Este segundo passo é equivalente a:* $\boldsymbol{c}(2) = tanh[(\sum_{j=2}^{3} \boldsymbol{x}(j)^T \boldsymbol{F}(j)) + b]$*. Figura adapatada a partir do texto de Kim (2019).*

$\boldsymbol{c'_m}$ poderia ser calculado do seguinte modo:

$$\boldsymbol{c'_m} = [max(\boldsymbol{c}_{m_1}), max(\boldsymbol{c}_{m_2}), ..., max(\boldsymbol{c}_{m_k})] \tag{3.7}$$

A operação *max* reduz a dimensionalidade extraindo o maior elemento de cada vetor. Esta redução da dimensionalidade é um fator importante para problemas de classificação, por exemplo. Independente do tamanho da entrada e do filtro, a dimensão da camada de saída vai ser mantida. Além disso, a camada *maxpool* é invariante a pequenas translações. Isto permite extrair características relevantes (e.g. palavras referindo-se a leitura de arquivo: *file* e *open*) de uma sentença independente da sua localização e adiciona-las na representação final (Young *et al.*, 2017).

Para exemplificar a operação completa, a Figura 3.3 a seguir ilustra um exemplo da aplicação de diferentes filtros convolucionais com diferentes tamanhos de janela m em uma sentença de 6 palavras. A sentença é representada através de um vetor $\boldsymbol{x} \in \mathbb{R}^{nXd}$, onde $n = 6$ e $d = 5$. Para um vetor $\boldsymbol{x} = \{\boldsymbol{x}(1), \boldsymbol{x}(2), ..., \boldsymbol{x}(n)\}$, $\boldsymbol{x}(i) \in \mathbb{R}^d, 1 \leq i \leq n$ é um vetor de representação distribuída e representa a i^{th} palavra da sentença. Neste exemplo, temos 3 filtros $\boldsymbol{F} \in \mathbb{R}^{mxd}$, onde $d = 5$ e m varia para cada um. Neste caso, os valores de m são 2, 3 e 4. Cada filtro tem tamanho $f = 2$, totalizando 6 filtros que são aplicados ao vetor $\boldsymbol{x}$, ao final.

No nosso trabalho, utilizaremos c'_m obtida através da operação 3.7 como o vetor de representação final da nossa sentença. No nosso caso, a sentença pode ser tanto uma questão quanto um trecho de código-fonte. O próximo passo é definir uma função objetivo que induza o nosso modelo a mapear estes vetores em um espaço vetorial, de tal forma que os vetores de representação de uma questão fiquem próximos dos vetores de trechos de código anotados como solução.

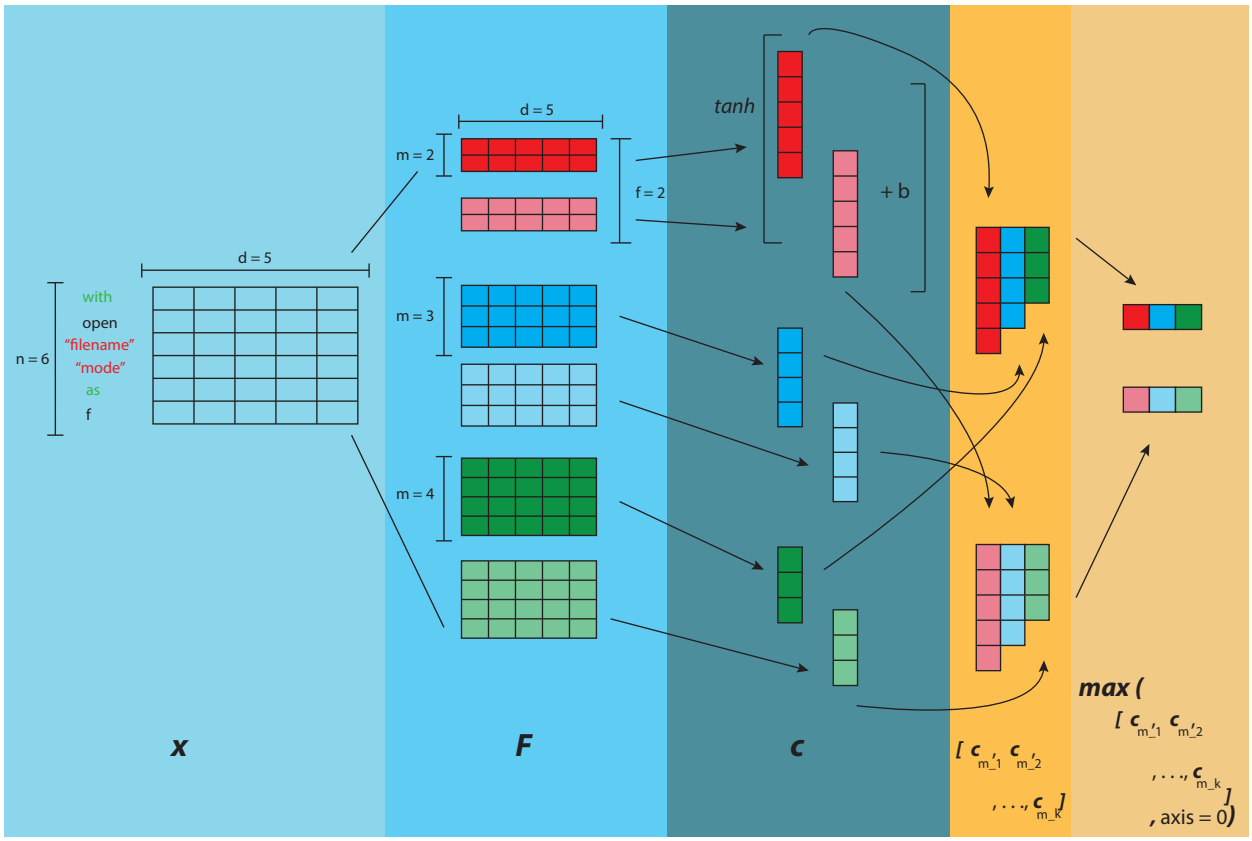


Figura 3.3: *Ilustração das 3 operações realizadas pela nossa arquitetura CNN para obter o vetor de representação final c'_m. Neste exemplo, utilizamos 3 filtros $F \in \mathbb{R}^{nXdXf}$ com diferentes janelas m, onde $m = \{2, 3, 4\}$. Cada filtro tem tamanho $f = 2$. Temos um total de 6 filtros ao final. Cada filtro é aplicado a sentença $x \in \mathbb{R}^{nXd}$, onde $n = 6$ e $d = 5$. O primeiro passo é a obtenção do vetor c de características. Posteriormente, é realização a operação de concatenação, no qual é obtido o vetor c_m, conforme descrito na operação 3.5. Ao final, a operação max é aplicada no eixo 0, obtendo o vetor de representação final c'_m. Este vetor c'_m será o nosso vetor de representação das nossas sentenças, i.e., o vetor de representação das questões e trechos de código-fonte. Figura adaptada do artigo de Zhang e Wallace (2015)*

3.3 Função objetivo

O nosso objetivo é encontrar um modelo que correlacione as questões as respostas em um mesmo espaço vetorial. Lembrando que as questões e os trechos de código serão representados por um vetor calculado a partir da nossa arquitetura CNN. Para encontrar este modelo, adotaremos a mesma abordagem utilizada por Feng *et al.* (2015), o método *pairwise*. O método *pairwise* consiste em treinar o modelo para classificar as respostas corretas com uma pontuação maior do que as incorretas. Formalmente, o modelo será treinado do seguinte modo:

Seja $\mathbb{Q}$ um conjunto de questões e $\mathbb{C}$ um conjunto dos trechos de código-fonte. $\mathbb{Q}$ e $\mathbb{C}$ compõem o conjunto de dados de treinamento. O dado de entrada fornecido para o nosso modelo será uma tripla $< \boldsymbol{q}, \boldsymbol{c}^+, \boldsymbol{c}^- >$, onde $\boldsymbol{c}^+ \in \mathbb{C}$ é uma resposta anotada como correta para a questão $\boldsymbol{q} \in \mathbb{Q}$ e $\boldsymbol{c}^- \in \mathbb{C}$ é uma resposta incorreta. O objetivo é induzir o nosso modelo a classificar $\boldsymbol{c}^+$ com uma pontuação maior que $\boldsymbol{c}^-$. Para isto, utilizaremos a seguinte função de custo *hinge*:

$$J = max(0, m - h_\theta(\boldsymbol{q}, \boldsymbol{c}^+) + h_\theta(\boldsymbol{q}, \boldsymbol{c}^-)) \tag{3.8}$$

Onde m é uma margem e h_θ é uma função de similaridade, e.g., *cosine*. Lembrando que o objetivo em uma rede neural é minimizar a função custo J. Desta maneira, o modelo vai ser incentivado a satisfazer a seguinte condição: $h_\theta(\boldsymbol{q}, \boldsymbol{c}^+) - h_\theta(\boldsymbol{q}, \boldsymbol{c}^-) \geq m$. Quer dizer, a função *hinge* induz o modelo a classificar as respostas corretas com uma pontuação maior do que as incorretas por uma certa margem m.

Dado que estamos utilizando a função *cosine* de similaridade, $h_\theta(\boldsymbol{q}, \boldsymbol{c})$ pode ser definido como $\{h_\theta(\boldsymbol{q}, \boldsymbol{c}) \in \mathbb{R} | -1 \leq h_\theta(\boldsymbol{q}, \boldsymbol{c}) \leq 1\}$, onde $\boldsymbol{q} \in \mathbb{Q}$, $\boldsymbol{c} \in \mathbb{C}$ e -1 indica vetores exatamente opostos e 1 vetores exatamente iguais. E conforme citado anteriormente, a nossa função $\mathbb{J}$ incentiva a seguinte condição: $h_\theta(\boldsymbol{q}, \boldsymbol{c}^+) \geq h_\theta(\boldsymbol{q}, \boldsymbol{c}^-) + m$. Desta forma, estamos estimulando os vetores $\boldsymbol{c}^+$ a ocuparem regiões próximas de $\boldsymbol{q}$, enquanto os vetores $\boldsymbol{c}^-$ serão estimulados a ocuparem regiões mais distantes. Este estímulo está de acordo com o nosso objetivo, que é encontrar um modelo que correlacione as questões as suas respostas.

3.4 Considerações

A nossa proposta apresentada neste capítulo foi a utilização de redes convolucionais na representação de questões e trechos de código-fonte. Conforme apresentado na Seção 3.2, a nossa arquitetura utiliza as operações de convolução e *maxpool*. Estas operações partem da hipótese de que função que deve ser aprendida contém apenas interações locais e são invariantes a pequenas translações. Para a tarefa de *code retrieval*, a nossa hipótese, é que a identificação das interações locais entre as palavras pode auxiliar na obtenção de uma representação contextualizada. Por exemplo, para inferir o contexto do trecho de código a seguir, o CNN deveria ser capaz de identificar a interação entre as palavras *csv* e *writer* na linha 4, por exemplo.

CNN

```
1 import csv
2
3 with open("output.csv", "wb") as f:
4     writer = csv.writer(f)
5     writer.writerows(a)
```

Um ponto observado por Young *et al.* (2017) em outros trabalhos, é que o CNN, normalmente, não consegue inferir o contexto em sentenças curtas. Para o nosso conjunto de dados, isto é um problema, pois as questões são compostas em média por 9 palavras e no máximo 32. Já os trechos de código tem em média 50 palavras e no máximo 300. Uma forma de mitigar este problema é através do uso de um vetor de representação distribuída. No nosso caso, os vetores de entrada serão compostos por vetores de representação distribuída obtidos através do algoritmo não-supervisionado *word2vec*.

Além disso, segundo Goodfellow *et al.* (2016b), CNN não consegue fazer associações entre palavras muito distantes em uma sentença. No exemplo anterior, o CNN teria dificuldade para associar a declaração da biblioteca *csv* na linha 1 com a função *writerows* na linha 5. A nossa hipótese é que uma associação muito distante entre as palavras de

uma sentença não seja tão importante para a recuperação de trecho de código-fonte no nosso conjunto de dados. O nosso conjunto de dados é formado por questões do tipo *how-to-do-it* coletadas a partir do site Stack Overflow. Este tipo de questão normalmente contém respostas curtas e diretas (Yao *et al.*, 2018). Uma dependência distante entre as palavras seria apropriado, provavelmente, para questões e trechos de código que envolvam regras de negócio, por exemplo.

Ao optar por uma arquitetura, temos que levar em consideração as suas características inerentes e suas limitações. No caso do CNN, ele prioriza as interações locais ao invés de associações de palavras muito distantes. A nossa hipótese inicial é que esta característica seja mais importante para a recuperação de trecho de código-fonte no conjunto de dados que estamos utilizando. Dado estas características, reiteramos a nossa pergunta de pesquisa que pretendemos responder ao longo deste trabalho:

- Será que o CNN é capaz de extrair as características latentes e mais importantes de modo a facilitar o modelo a encontrar uma correlação entre os trechos de código-fonte e as questões?

Indiretamente, dado que o CNN prioriza interações locais, estaremos respondendo as seguintes perguntas:

- As interações locais auxiliam na aproximação das questões aos trechos de código-fonte?
- A partir das interações locais, é possível inferir o contexto do trecho de código?

O que entedemos por características latentes e importantes são palavras ou variáveis latentes que permitam inferir o contexto de um trecho de código. No exemplo anterior, as palavras *csv*, *writer* e *writerows* permitem deduzir o contexto do trecho de código. Que no caso, refere-se a escrita dos valores de cada elemento de um vetor em um arquivo CSV. Nem sempre estas características são observáveis diretamente nos dados. Nestes casos, as redes deep learning capturam estas características através de variáveis latentes ou *hidden*, h. Através da variável *hidden* h, ela consegue identificar uma dependência

indireta entre duas variáveis quaisquer v_i e v_j através da dependência direta entre v_i e $\boldsymbol{h}$ e $\boldsymbol{h}$ e v_j (Goodfellow *et al.*, 2016e).

Para avaliar se o modelo está inferindo o contexto do trecho de código corretamente, verificaremos se o modelo está aproximando as questões aos trechos de código que são respostas. Conforme descrito na Seção 3.3, o modelo será induzido a fazer esta aproximação durante o treinamento através da função *hinge*. Mas para saber se o modelo aprendeu, avaliaremos esta aproximação em outro conjunto de dados. No nosso caso, avaliaremos em um conjunto de dados anotados manualmente. O desempenho do modelo será medido através da métrica MRR descrita no Capítulo 4. Um valor alto do MRR indica que o modelo está aproximando corretamente as questões aos trechos de código-fonte anotados como respostas. Para nós, isto é um indicativo de que o modelo está inferindo o contexto do trecho de código corretamente.

Capítulo 4

Experimento piloto

Este capítulo é parte de um estudo preliminar (de Rezende Martins e Gerosa, 2019) apresentado no Workshop on Software Visualization, Evolution and Maintenance (VEM) de 2019. Esse estudo foi um piloto para avaliar o uso das redes convolucionais no problema de *code retrieval*. Utilizamos uma arquitetura com uma pequena variação em relação a nossa arquitetura proposta no Capítulo 3. E para comparar, utilizamos duas arquiteturas: uma arquitetura bi-LSTM com CNN e outra arquitetura de referência *Embedding*. Os resultados da arquitetura CNN foram promissores. A arquitetura apresentou um valor de métrica Mean Reciprocal Rank (MRR) de $0,58$. Em 75% das vezes, as respostas corretas apareceram entre as 3 primeiras posições, de um total de 50 possíveis respostas.

4.1 Conjunto de dados

Para este estudo, utilizamos parte do conjunto de dados disponibilizado por Yao *et al.* (2018). Esse conjunto é formado por $\mathbf{147.546}$ pares de questões e trechos de código-fonte em Python e $\mathbf{119.519}$ em SQL. Estes pares foram coletados do site Stack Overflow. Uma peculiaridade desses dados em relação aos dados utilizados nos outros trabalhos Allamanis *et al.* (2015); Iyer *et al.* (2016), é o fato de conter questões do tipo *how-to-do-it*. As respostas para este tipo de questão costumam ser mais diretas e ter apenas um trecho de código-fonte como solução (Yao *et al.*, 2018). Essas questões normalmente expressam as intenções do desenvolvedor, indo ao encontro com a nossa definição da Seção 2.1 adotada para o problema de *code retrieval*.

Os dados disponibilizados por Yao *et al.* (2018) são divididos em 3 (três) subconjuntos distintos. Um subconjunto é formado apenas por questões coletadas do Stack Overflow que continham apenas um trecho de código-fonte na descrição da resposta. O outro é formado por questões que apresentavam mais de um trecho de código-fonte na descrição. O terceiro é formado por pares de questões e trechos de código-fonte anotados manualmente.

Nos casos em que há mais de um trecho de código-fonte na descrição da resposta, um trecho de código não é necessariamente uma solução para a pergunta. No exemplo a seguir, os trechos *T2* e *T4* não são soluções para a questão. Já o trecho *T2* apenas inicializa um array e o trecho *T4* mostra a saída do comando *print* para o vetor já convertido[1].

[1]Site: https://stackoverflow.com/questions/34349762/convert-2d-numpy-array-to-string Esta questão apresenta 6 trechos de código-fonte, mas para melhor visualização, optamos por colocar apenas 4 trechos em nosso exemplo

Título da questão: convert 2d numpy array to string 1

Trecho T1: A one-liner will do:

```
b = '\n'.join('\t'.join('%0.3f' %x for x in y) for y in a)
```

Trecho T2: Using a simpler example:

```
>>> a = np.arange(25, dtype=float).reshape(5, 5)
>>> a
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.],
       [ 10.,  11.,  12.,  13.,  14.],
       [ 15.,  16.,  17.,  18.,  19.],
       [ 20.,  21.,  22.,  23.,  24.]])
```

Trecho T3: is equivalent to:

```
res = []
for y in a:
    res.append('\t'.join('%0.3f' %x for x in y))
b = '\n'.join(res)
```

Trecho T4: prints this:

```
0.000   1.000   2.000   3.000   4.000
5.000   6.000   7.000   8.000   9.000
10.000  11.000  12.000  13.000  14.000
15.000  16.000  17.000  18.000  19.000
20.000  21.000  22.000  23.000  24.000
```

Para diferenciar estes casos, Yao *et al.* (2018) anotaram os pares com 1, quando o trecho é solução para a pergunta e 0, em caso contrário. Esta anotação foi feita automaticamente por um framework proposto por Yao *et al.* (2018). Ao final, o conjunto de dados é dividido da seguinte maneira:

	Questão	
Código-fonte	**Python**	**SQL**
Apenas 1 trecho de código na descrição da resposta	85.294	75.637
Trechos de código-fonte anotados automaticamente	60.083	41.826
Trechos de código-fonte anotados manualmente	2.169	2.056
Total	**147.546**	**119.519**

Tabela 4.1: *Divisão do conjunto de dados disponibilizado por Yao et al. (2018). O conjunto formado por "Trechos de código-fonte anotados automaticamente" contém questões que tem mais de um trecho de código-fonte por resposta. Quando há mais de um trecho de código-fonte por resposta, nem todo trecho é uma solução. Neste caso, Yao et al. (2018) criaram um framework para anotá-los automaticamente. Eles obtiveram F1 de* $0,916$ *e acurácia de* $0,911$ *em seus testes.*

4.2 Treinamento e avaliação

Para o treinamento do modelo, utilizamos apenas os pares de questões e trechos de código-fonte em Python. Inicialmente, utilizamos apenas os pares anotados automaticamente. Esse conjunto de dados apresenta uma variabilidade maior, em torno de 27% das questões contém mais de um trecho de código-fonte anotados como correto, i.e., 27% das perguntas tem mais de uma opção de solução.

Adotamos o mesmo procedimento de treinamento e avaliação proposto por Iyer *et al.* (2016). Para o treinamento, foi utilizado o conjunto com 60.083 pares. Para escolha do modelo e avaliação final, foi utilizado o conjunto de dados anotados manualmente. A divisão das amostras para treinamento e avaliação podem ser visualizadas na Tabela 4.2.

Amostras	**Quantidade de pares** $< q_i, c_i^+ >$
Treinamento	60.083
DEV	1.085
EVAL	1.084
Total	**62.252**

Tabela 4.2: *Divisão das amostras para treinamento e avaliação. O conjunto de dados é formado por pares* $< q_i, c_i^+ >$*, onde* q_i *é uma questão e* c_i^+ *é um trecho de código-fonte anotado como correto. O conjunto formado por pares anotados manualmente foi dividido em DEV e EVAL conforme o procedimento descrito por Iyer et al. (2016).*

O procedimento de treinamento utilizado foi: O modelo é treinado durante 80 épocas. Caso a função de perda *hinge* fique abaixo de $0,001$, o treinamento é interrompido. A cada época, o modelo é avaliado na amostra *DEV*. O intuito desta avaliação é obter o melhor

modelo conforme a métrica MRR. Esta avaliação na amostra DEV é feita da seguinte maneira:

Para cada par $< q_i, c_i^+ >$ da amostra *DEV*, onde q_i uma questão e c_i^+ uma questão anotada como correta. Outros 49 distratores c' são selecionados aleatoriamente da amostra de treinamento, tal que $c_i^+ \neq c'$. Para cada questão, o modelo calcula a similaridade entre a questão e os trechos de código-fonte. O cálculo de similaridade é feito através da função h_θ, onde h_θ é a função *cosine*.

Posteriormente, os trechos de código-fonte são ordenados de forma decrescente, do mais similar (maior pontuação) ao menos similar (menor pontuação). Com os trechos ordenados, obtém-se a posição do trecho c_i^+ para cálculo do *reciprocal rank*. *Reciprocal rank* é o inverso da posição da primeira ocorrência de c_i^+ encontrada no resultado. Com o *reciprocal rank*, calcula-se o MRR. MRR é a média do *reciprocal rank* para a amostra inteira (Gu *et al.*, 2018):

$$MRR = \frac{1}{n} * \sum_{i=1}^{n} \frac{1}{p_i^+} \tag{4.1}$$

Onde n é a quantidade de questões presentes na amostra, p_i^+ é a posição da primeira ocorrência do trecho c_i^+ entre os trechos ordenados.

Este procedimento é repetido durante 20 vezes. A cada iteração, outros 49 distratores são selecionados. Ao final, obtém-se a média *MRR* do modelo. O modelo que obtiver a maior média *MRR* ao final do treinamento é escolhido. A avaliação final na amostra *EVAL* utiliza o mesmo procedimento da amostra *DEV* descrito acima.

4.3 Pré-processamento

Diferente do trabalho de Tan *et al.* (2015), nós geramos duas matrizes de representação distribuída distintas $\boldsymbol{T_q}$ e $\boldsymbol{T_c}$ para o vocabulário das questões e para o vocabulário dos trechos de código-fonte, conforme citado no Capítulo 3. Antes de gerar a matriz $\boldsymbol{T_c}$, utilizamos uma função disponibilizada por Yao *et al.* (2018) para fazer um pré-processamento dos trechos de código-fonte. Essa função substitui os valores literais numéricos e textos

pelas palavras *"NUMBER"* e *"STRING"*. Além disso, os comentários são removidos e as variáveis são substituídas por *"VAR"*. O caractere de quebra de linha é substituído por *"NEWLINE"*.

As matrizes $\boldsymbol{T_q}$ e $\boldsymbol{T_c}$ são geradas pelo *word2vec* com *skip-gram*. Na Figura 4.1 a seguir, podemos visualizar um exemplo da representação do vetor de representação distribuída gerada pelo *word2vec*. Essa imagem foi criada com o auxílio da ferramenta *t-SNE*. Ela auxilia na visualização de dados de dimensão elevada (Khadder, 2019; scikit learn, 2019). De acordo com a Figura 4.1, as palavras *from* e *import* são similares. Assim como *if* e *else*, e também *list*, *dict* e *set*, por exemplo.

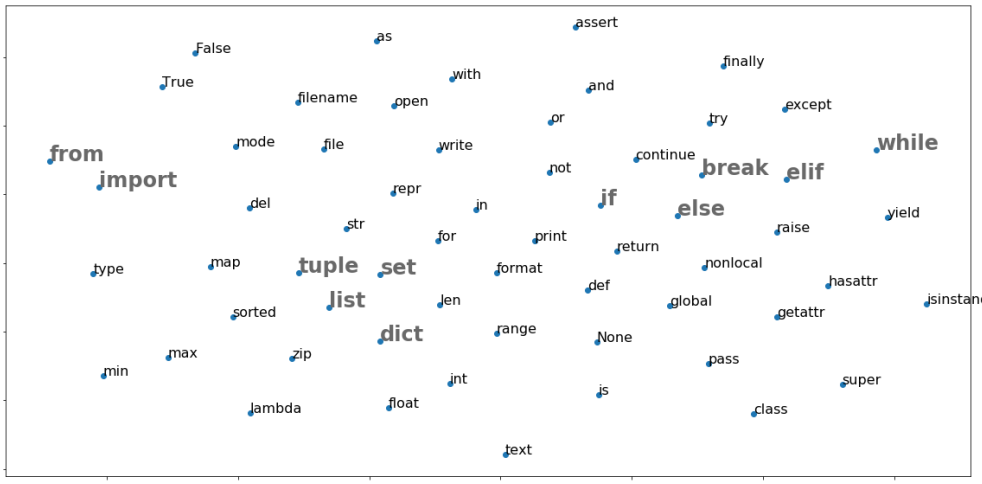


Figura 4.1: *Representação em 2D do vetor de representação distribuída de trechos de código-fonte. Imagem gerada através da ferramenta t-SNE. O vetor de representação distribuída foi criado a partir da amostra de trechos de código-fonte em Python disponibilizada por Yao et al. (2018). Vetor criado utilizando o word2vec com o algoritmo skip-gram e o parâmetro window com o valor* 5.

4.4 Arquiteturas

Para este estudo, comparamos a arquitetura CNN com outras duas arquiteturas: bi-LSTM com CNN e *Embedding*. A arquitetura *Embedding* combina os vetores de representação distribuída de cada palavra através de uma camada *max pool*. A arquitetura CNN utilizada neste estudo, utiliza uma camada inicial *hidden*. Esta camada obtém um

vetor z a partir das questões e trechos de código-fonte. Este vetor z servirá como entrada para a nossa arquitetura CNN que foi descrita na Seção 3.2.

$$\boldsymbol{z} = tanh(\boldsymbol{W} * \boldsymbol{x} + \boldsymbol{b}) \tag{4.2}$$

Onde $\boldsymbol{W}$ é a matriz de pesos, $\boldsymbol{x}$ é vetor que contém os vetores de representação distribuída das palavras e $\boldsymbol{b}$ é o vetor *bias*. As figuras 4.2, 4.3 e 4.4 ilustram as arquiteturas utilizadas:

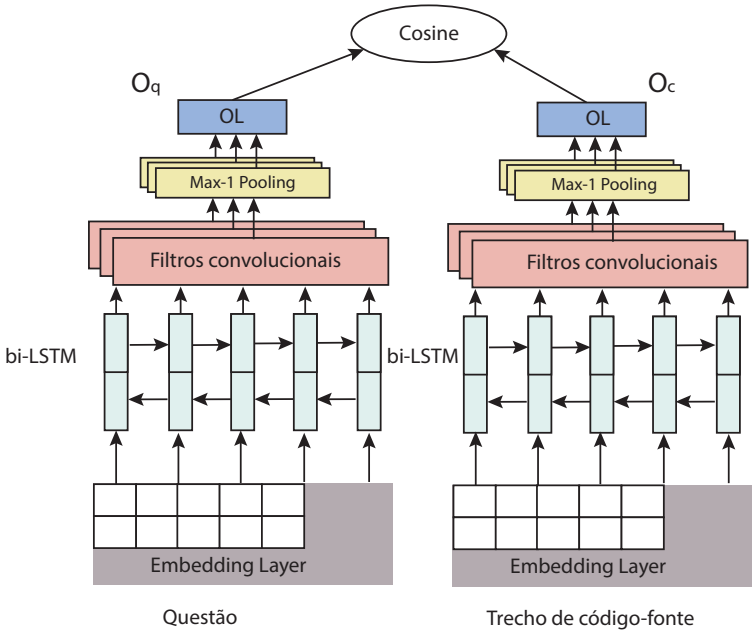


Figura 4.2: *Figura da arquitetura bi-LSTM com CNN. Figura utilizada no artigo de Rezende Martins e Gerosa (2019)*

Nas figuras 4.2, 4.3 e 4.3, a palavra *Embedding Layer* refere-se ao vetor de representação distribuída das palavras. *OL* é um acrônimo de *Output Layer* ou camada de saída. O_q indica a camada de saída da questão. O_c indica a camada de saída do trecho de código-fonte.

Todas as arquiteturas utilizam uma camada *max pool* e, ao final, calculam a similaridade através da função *cosine*. Não realizamos otimização ou ajustes dos hiper-parâmetros dos modelos. Utilizamos os mesmos hiper-parâmetros propostos por Tan *et al.* (2015) para todas as arquiteturas. Com exceção do hiper-parâmetro filtro para a arquitetura CNN, neste caso, alteramos o valor para 100, pois o valor 1000 proposto por Tan *et al.* (2015) aumentou a capacidade do modelo e estava causando *overfitting*. Para a função de perda

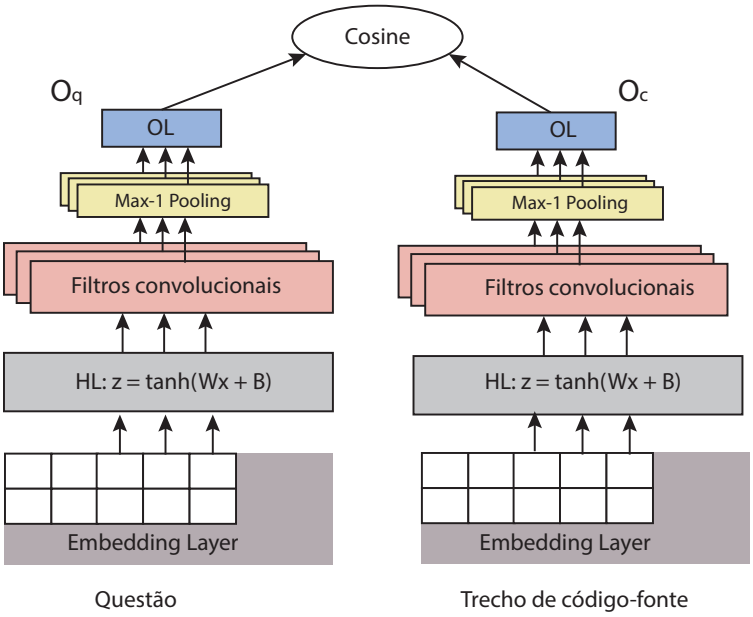


Figura 4.3: *Figura da arquitetura CNN com a primeira camada de hidden layer proposta para o artigo de Rezende Martins e Gerosa (2019).*

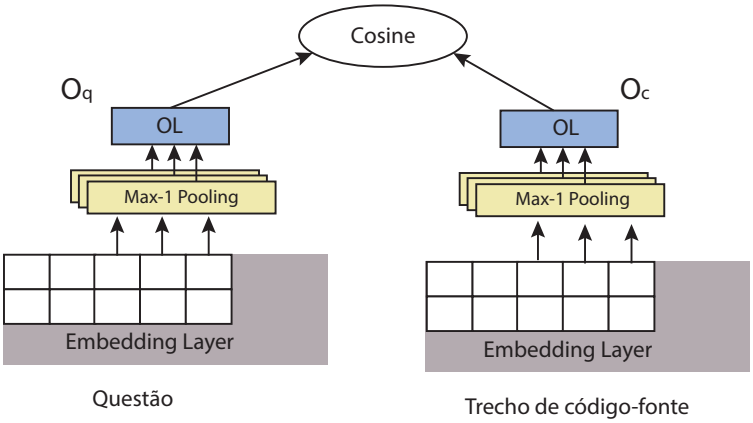


Figura 4.4: *Figura da arquitetura de referência Embedding para comparação. Figura utilizada no artigo de Rezende Martins e Gerosa (2019)*

hinge, utilizamos o valor $0,009$ proposto por Feng *et al.* (2015) para a margem. Em relação a dimensão do vetor de representação distribuída das palavras (*word embedding*), utilizamos o valor 100.

4.5 Resultados preliminares

Os resultados preliminares foram coletados a partir da avaliação dos modelos na amostra *EVAL*. O valor final MRR é a média obtida após 20 iterações. Conforme a Tabela 4.3, a arquitetura CNN obteve um resultado próximo da arquitetura bi-LSTM com

CNN. Apesar do resultado ser relativamente menor, o seu tempo de duração de treinamento é de apenas 6s, enquanto o treinamento do bi-LSTM durou cerca de 48 minutos. Tanto o treinamento quanto a avaliação foram executadas na plataforma Colab do Google. No momento de treinamento e coleta dos resultados, a execução foi feita em uma máquina virtual com acesso a uma vGPU Tesla K80.

Modelos	**Resultados (MRR)**
Embedding	$0,52 \pm 0,01$
CNN	$\mathbf{0,58 \pm 0,01}$
bi-LSTM-CNN	$\mathbf{0,60 \pm 0,02}$

Tabela 4.3: *Resultado preliminar do modelo CNN proposto para o artigo de Rezende Martins e Gerosa (2019) em comparação com outras duas arquiteturas (bi-LSTM com CNN e Embedding). Estes resultados foram obtidos a partir da amostra EVAL.*

Para entender um pouco melhor o resultado da média harmônica MRR, a figura abaixo exibe as posições da primeira ocorrência do trecho de código-fonte encontradas durante a avaliação dos modelos.

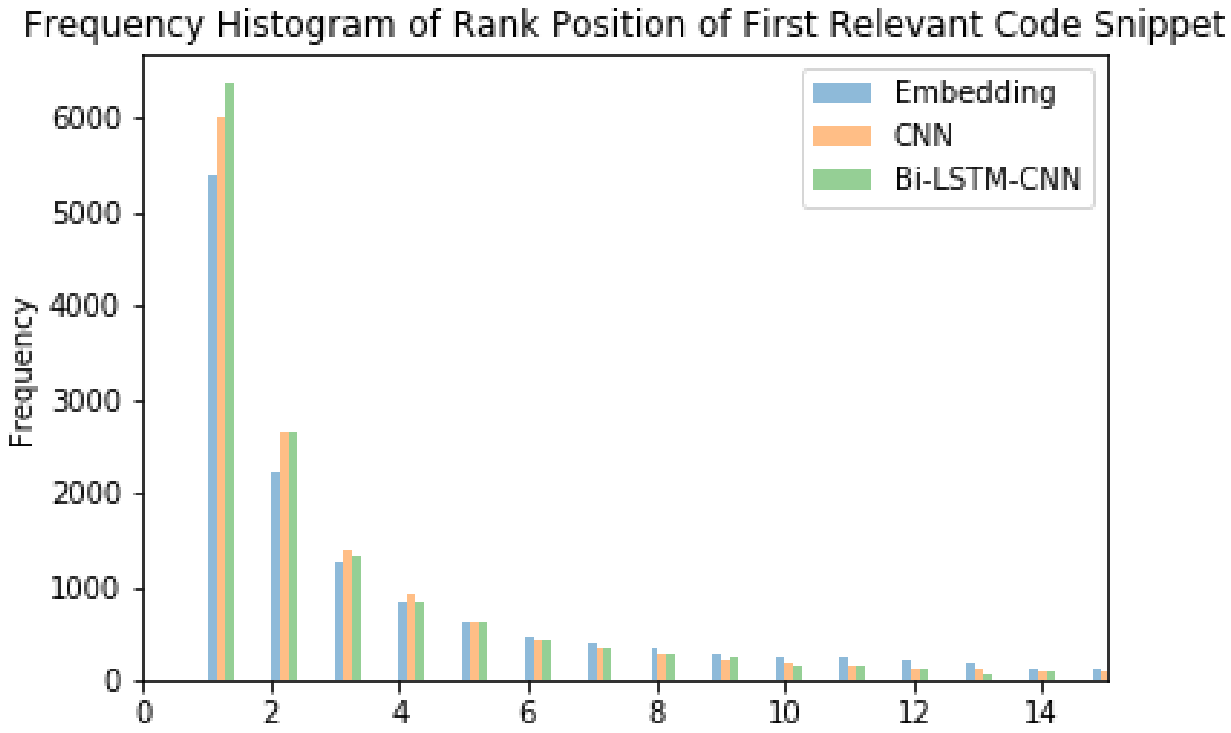


Figura 4.5: *Figura das primeiras posições observadas para o trecho de código-fonte anotado como correto.*

Tanto o CNN quanto o bi-LSTM com CNN conseguiram classificar os trechos de

código-fonte entre as 3 (três) primeiras posições em 75% dos casos. O modelo bi-LSTM com CNN obteve uma precisão TOP-1 de 51%, i.e., em mais da metade das vezes, o trecho de código-fonte relevante ficou na primeira posição. Já a nossa arquitetura proposta obteve um TOP-1 de aproximadamente 41%. Um ponto a ser levantado é que a nossa métrica MRR só leva em consideração a posição apontada pelo modelo para o trecho de código-fonte anotado como correto. Mesmo que o modelo apresente um outro trecho de código-fonte que também é solução, o nosso método de avaliação não leva em consideração. Neste caso, o modelo é penalizado.

No exemplo a seguir[2], a rede bi-LSTM com CNN conseguiu apontar corretamente o trecho de código-fonte anotado como correto na primeira posição. A nossa arquitetura CNN, apesar de não apresentar o trecho propriamente anotado, ele apresentou outro trecho que também serve como solução. Neste caso, apesar de ter sido penalizado, ele conseguiu responder a questão. Este é um ponto importante que teremos que analisar com cautela ao longo deste trabalho.

[2]Este exemplo refere-se a questão https://stackoverflow.com/questions/24593478/python-and-appending-items-to-text-and-excel-file

Python and appending items to text and excel file 2

bi-LSTM com CNN

```
Yvalues = [1, 2, 3, 4, 5]
file_out = open('file.csv','wb')
mywriter=csv.writer(file_out, delimiter = '\n')
mywriter.writerow(Yvalues)
file_out.close()
```

CNN

```
import csv


with open("output.csv", "wb") as f:
    writer = csv.writer(f)
    writer.writerows(a)
```

4.5.1 Ameaças à validade

Conforme citado anteriormente, Yao *et al.* (2018) anotaram o conjunto de dados utilizando um framework proposto em seu artigo. Para anotá-los, os autores treinaram uma rede neural no conjunto de dados anotado manualmente. Em nosso trabalho, fizemos o caminho inverso. Treinamos os nossos modelos nos dados anotados automaticamente e avaliamos no conjunto anotado manualmente. Para diminuir o viés, adotamos o procedimento proposto por Iyer *et al.* (2016) descrito na Seção 4.2.

4.5.2 Considerações

Os resultados apresentados pelo nosso modelo CNN parecem bastante promissores, a nosso ver. O próximo passo é comparar o nosso modelo com o modelo proposto por

Cambronero *et al.* (2019), que é o estado da arte.

Não fizemos ajustes dos hiper-parâmetros e nem uso de regularização durante o treinamento. Na Figura 4.7 abaixo, podemos ver que o CNN apresenta uma diferença grande entre a curva de erro na amostra de validação e da amostra de treinamento, principalmente a partir da época 5 (eixo X), próximo da interrupção do treinamento. Isto é um indicativo de *overfitting*. A arquitetura bi-LSTM com CNN apresenta o mesmo problema (ver Figura 4.6). Já no caso da arquitetura *Embedding*, Figura 4.8, as duas curvas de erro estão próximas até a interrupção.

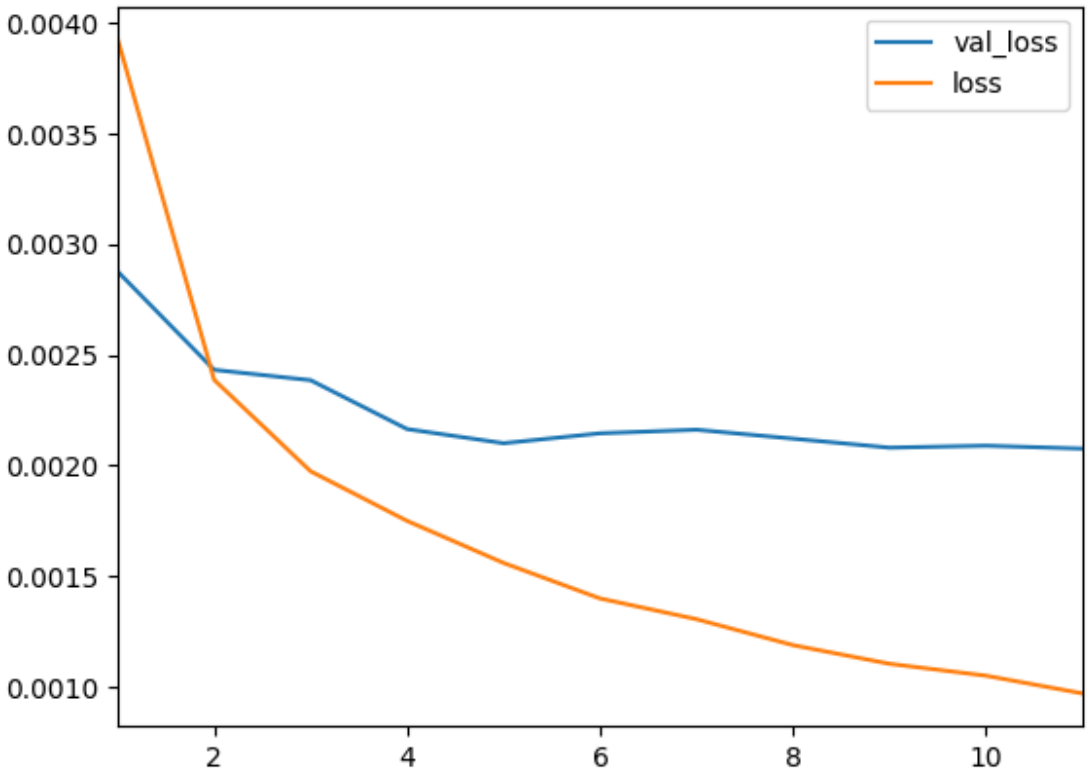


Figura 4.6: *Gráfico do treinamento do modelo bi-LSTM com CNN. Gráfico do erro de validação (*val_loss*) e erro na amostra de treinamento (*loss*) por época (*eixo X*). O melhor modelo em relação a métrica MRR foi obtido na época* 9.

Podemos perceber que de acordo com as figuras 4.7 e 4.6, tanto o CNN quanto o bi-LSTM com CNN tem uma margem a melhorar. Além da margem de melhora no treinamento, uma análise mais detalhada do resultado é necessária. Conforme o exemplo citado na seção anterior, o modelo CNN apresentou uma resposta correta para a per-

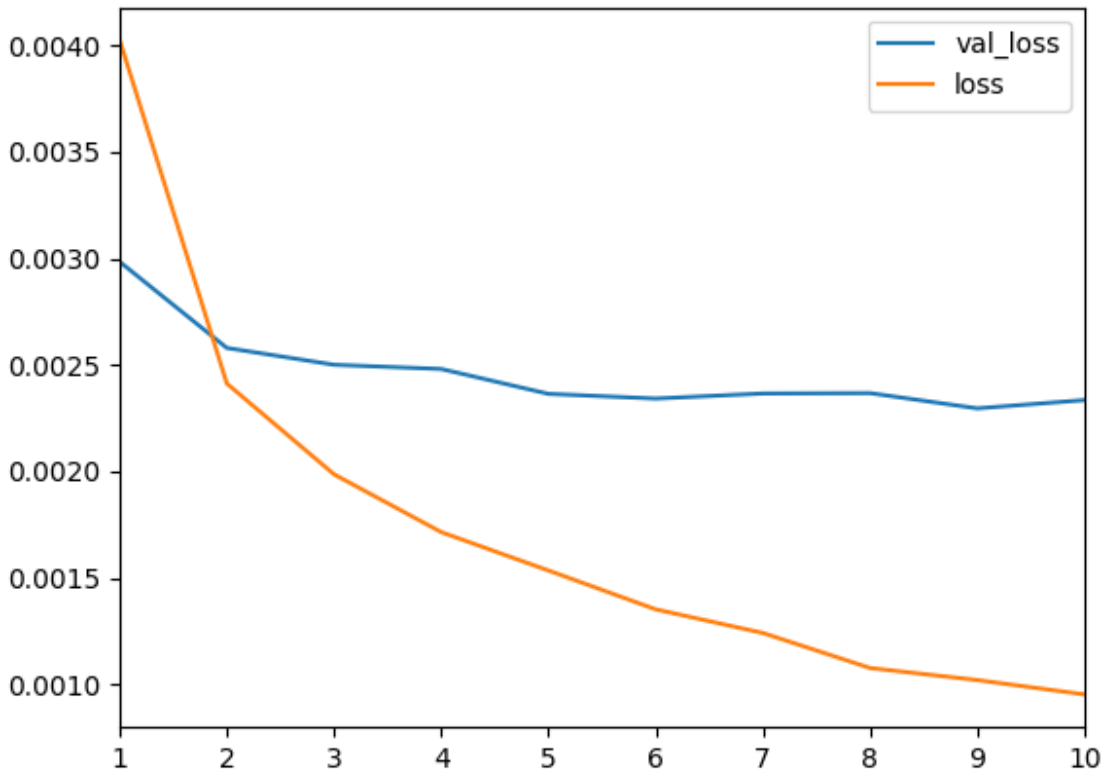


Figura 4.7: *Gráfico do treinamento do CNN. Gráfico do erro de validação (*val_loss*) e erro na amostra de treinamento (*loss*) por época (*eixo X*). O melhor modelo em relação a métrica MRR foi obtido na época* 9.

gunta, porém foi penalizada pois não era exatamente o trecho anotado como correto. Caso haja mais casos deste tipo, podemos adotar o mesmo procedimento proposto por Cambronero *et al.* (2019) de avaliação automática. Um trecho será considerado correto quando a diferença do resultado da função h_θ de similaridade entre ele e o trecho anotado como correto estiver dentro de um certo intervalo.

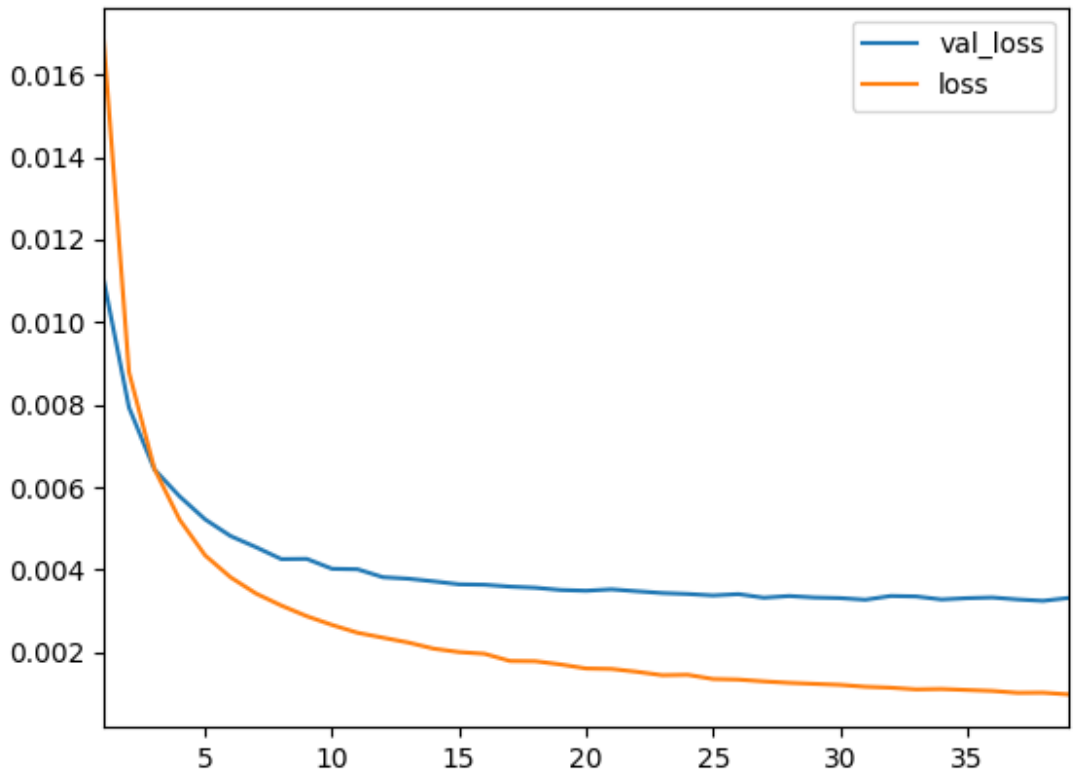


Figura 4.8: *Gráfico do treinamento do modelo Embedding. Gráfico do erro de validação (*val_loss*) e erro na amostra de treinamento (*loss*) por época (*eixo X*). O melhor modelo em relação a métrica MRR foi obtido na época* 39.

Capítulo 5

Cronograma

5.1 Próximos passos

A proposta deste trabalho é avaliar o uso das redes convolucionais no problema de *code retrieval*. Conforme Goodfellow *et al.* (2016c), o primeiro passo é definir um objetivo, um valor alvo para o modelo. No nosso caso, o objetivo é obter um resultado comparável ao modelo proposto por Cambronero *et al.* (2019), que é o estado da arte atualmente.

O processo de treinamento de uma rede neural envolve algumas etapas. Desde o processo de tomada de decisão para uso de deep learning, coleta dos dados, seleção da arquitetura, treinamento e avaliação do modelo. Algumas etapas foram parcialmente concluídas durante o estudo preliminar. A Figura 5.1 a seguir ilustra as etapas do processo de treinamento e o andamento através de um mapa de calor. A Tabela 5.1 lista as principais atividades em andamento e/ou concluídas.

Etapa	**Atividade**	**Tarefa**	**Forma de avaliação**	**Status**
Coleta/Pré-processamento dos dados	Utilização dos dados coletados por Yao *et al.* (2018)			Concluído

Coleta/Pré-processamento dos dados	Pré-processamento dos dados	Verificar a possibilidade de quebra das palavras do código-fonte de acordo com a convençao de nomenclatura (*snake case*). E analisar a possibilidade de remoção de *stop words*	Análise das curvas de erros de validação e treinamento. Melhora da métrica MRR.	Parcialmente concluído
Coleta/Pré-processamento dos dados	Treinamento não-supervisionado: *word2vec*	Análise do vetor de representação distribuída e se há necessidade de ajustes nos parâmetros de treinamento e dimensão do vetor	Visualização do t-SNE para as 50 palavras mais frequentes das questões e trechos de código-fonte e suas relações.	Parcialmente concluído
Definição do tipo de rede neural/arquitetura	Modelo proposto: CNN			Concluído
Definição do tipo de rede neural/arquitetura	Modelos para comparação: *Embedding* e o *bi-modal embedding* com o mecanismo de atenção proposto por Cambronero *et al.* (2019)	Implementar o modelo proposto por Cambronero *et al.* (2019)		Em andamento

Definição do tipo de rede neural/arquitetura	Definição do objetivo: Resultado comparável ao modelo proposto por Cambronero *et al.* (2019) em um mesmo ambiente de testes e conjunto de dados	Inicialmente, buscamos um desempenho superior em pelo menos 10p.p. em relação ao modelo do Cambronero *et al.* (2019). Dependendo do desempenho do nosso modelo, este valor alvo pode ser revisto	MRR	Em andamento
Definição do tipo de rede neural/arquitetura	Seleção dos hiper-parâmetros do modelo	Esta tarefa será feita em conjunto com o treinamento. Conforme o desempenho do modelo, ajustes nos hiper-parâmetros devem ser feitos para aumentar ou diminuir a capacidade do modelo	Análise da curva de erro de validação e treinamento vs capacidade do modelo	Em andamento
Seleção do algoritmo de treinamento	Algoritmo de otimização para o modelo	Inicialmente, utilizamos o algoritmo de otimização Adam para o modelo. Uma alternativa é verificar o desempenho para outros algoritmos como SGD e RMSprop	Análise da curva de erro de validação e treinamento vs época	Parcialmente Concluído
Treinamento	Regularização	Utilizar técnica de regularização *dropout* ou *batch normalization*	Análise da curva de erro de validação e treinamento vs época	Não iniciado
Treinamento	Função objetivo	Função de perda *hinge*		Concluído

Treinamento	Critério de parada	Adotamos os mesmos critérios de treinamento proposto por Iyer *et al.* (2016). O treinamento é feito durantes 80 épocas ou enquanto o erro for maior que um certo valor. No caso, estamos utilizando o valor $0,001$		Concluído
Análise do desempenho da rede neural	*Overfitting* e extrapolação	Diminuir a diferença entre o erro de validação e o erro de treinamento do modelo CNN. Inicialmente, utilizaremos técnicas de regularização e algortimos de otimização.	Análise através das curvas de erro de validação e treinamento vs época e capacidade do modelo	Em andamento
Análise do desempenho da rede neural	Visualização do pior caso	Analisar as classificações feita pelo modelo através dos piores casos. No nosso caso, os piores casos são os trechos com os menores valores de MRR. Conforme Goodfellow *et al.* (2016c), esta análise permite identificar possíveis problemas no conjunto de dados.	Verificação manual	Em andamento

Análise do desempenho da rede neural	Avaliação manual	Avaliar manualmente uma amostra de questões e trechos de código-fonte classificados pelos modelos	Verificação manual	Em andamento
Uso do modelo	Busca	Coletar ou propor novas questões ao modelo e avaliar as respostas. Verificar também o desempenho do modelo na busca por trechos similares em um corpus de busca diferente, e.g., Github	Verificação manual	Não iniciado

Tabela 5.1: *Relação das principais atividades realizadas e a serem cumpridas neste trabalho.*

5.2 Cronograma

O processo de treinamento de um modelo é um ciclo, conforme a Figura 5.1. Para facilitar o acompanhamento do progresso do trabalho, preferimos definir alguns marcos a serem alcançados até a entrega. Os marcos que julgamos importantes são:

1. Implementação do modelo proposto por Cambronero *et al.* (2019)
2. Adição de regularização aos modelos
3. Avaliação dos piores casos
4. Avaliação manual
5. Compilação dos resultados
6. Submissão de um artigo com os resultados em um congresso internacional

O cronograma dos próximos passos pode ser visualizado através da Figura 5.2.

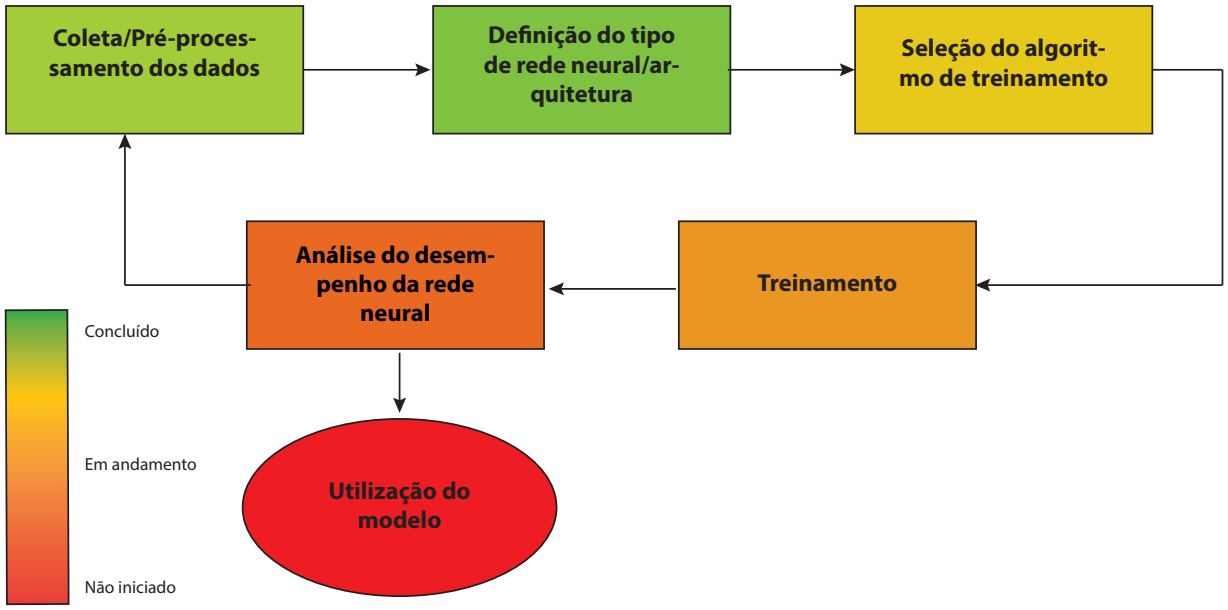
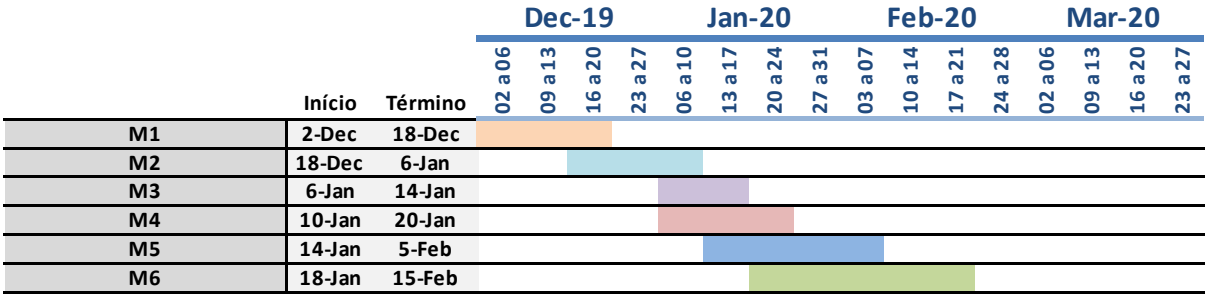


Figura 5.1: *Etapas do processo de treinamento de uma rede neural. As etapas de* Análise do desempenho da rede neural *e* Treinamento *estão no início. Enquanto a* Coleta/Pré-processamento dos dados *e* Definição do tipo de rede neural/arquitetura *estão parcialmente concluídas. Figura adaptada do livro Demuth et al. (2014)*



Marcos
M1 - Implementação do modelo proposto por Cambronero et al. (2019)
M2 - Adição de regularização aos modelos
M3 - Avaliação dos piores casos
M4 - Avaliação manual
M5 - Compilação dos resultados
M6 - Submissão de um artigo com os resultados em um congresso internacional

Figura 5.2: *Cronograma dos próximos passos até a entrega do trabalho*

Referências Bibliográficas

Allamanis *et al.***(2015)** Miltiadis Allamanis, Daniel Tarlow, Andrew D. Gordon e Yi Wei. Bimodal modelling of source code and natural language. Em *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, páginas 2123–2132. JMLR.org. URL http://dl.acm.org/citation.cfm?id=3045118.3045344. Citado na pág. 1, 3, 11, 12, 13, 14, 15, 29

Allamanis *et al.***(2018)** Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu e Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4):81:1–81:37. ISSN 0360-0300. doi: 10.1145/3212695. URL http://doi.acm.org/10.1145/3212695. Citado na pág. 12, 18

Cambronero *et al.***(2019)** José Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen e Satish Chandra. When deep learning met code search. *CoRR*, abs/1905.03813. URL http://arxiv.org/abs/1905.03813. Citado na pág. xi, 1, 3, 5, 7, 9, 11, 12, 13, 14, 15, 16, 40, 41, 43, 44, 45, 47

Chen e Zhou(2018) Qingying Chen e Minghui Zhou. A neural framework for retrieval and summarization of source code. Em *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, páginas 826–831, New York, NY, USA. ACM. ISBN 978-1-4503-5937-5. doi: 10.1145/3238147.3240471. URL http://doi.acm.org/10.1145/3238147.3240471. Citado na pág. 1, 5, 13, 14, 15

de Rezende Martins e Gerosa(2019) Marcelo de Rezende Martins e Marco Aurélio Gerosa. Um estudo preliminar sobre o uso de uma arquitetura deep learning para seleção de respostas no problema de recuperação de código-fonte. Em *Anais do VII Workshop on Software Visualization, Evolution and Maintenance (VEM)*, pá-

ginas 94–101, Porto Alegre, RS, Brasil. SBC. doi: 10.5753/vem.2019.7589. URL https://sol.sbc.org.br/index.php/vem/article/view/7589. Citado na pág. ix, x, xi, 18, 29, 35, 36, 37

Demuth *et al.*(2014) Howard B. Demuth, Mark H. Beale, Orlando De Jess e Martin T. Hagan. *Neural Network Design*, chapter 22. Martin Hagan, USA, 2nd edição. ISBN 0971732116, 9780971732117. Citado na pág. x, 48

Feng *et al.*(2015) M. Feng, B. Xiang, M. R. Glass, L. Wang e B. Zhou. Applying deep learning to answer selection: A study and an open task. Em *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, páginas 813–820. doi: 10.1109/ASRU.2015.7404872. Citado na pág. 13, 24, 36

Goodfellow *et al.*(2016a) Ian Goodfellow, Yoshua Bengio e Aaron Courville. *Deep Learning*, chapter 12, páginas 443–485. MIT Press. Citado na pág. v

Goodfellow *et al.*(2016b) Ian Goodfellow, Yoshua Bengio e Aaron Courville. *Deep Learning*, chapter 9, páginas 330–372. MIT Press. Citado na pág. 2, 25

Goodfellow *et al.*(2016c) Ian Goodfellow, Yoshua Bengio e Aaron Courville. *Deep Learning*, chapter 11, páginas 421–442. MIT Press. Citado na pág. 43, 46

Goodfellow *et al.*(2016d) Ian Goodfellow, Yoshua Bengio e Aaron Courville. *Deep Learning*, chapter 15, páginas 526–557. MIT Press. Citado na pág. 2, 6, 7, 11, 18

Goodfellow *et al.*(2016e) Ian Goodfellow, Yoshua Bengio e Aaron Courville. *Deep Learning*, chapter 16, páginas 558–589. MIT Press. Citado na pág. 27

Google(2019a) Google. Colaboratory, 2019a. URL https://research.google.com/colaboratory/faq.html. Citado na pág. v

Google(2019b) Google. Machine learning glossary, 2019b. URL https://developers.google.com/machine-learning/glossary/. Citado na pág. v, vi

Gu *et al.*(2018) Xiaodong Gu, Hongyu Zhang e Sunghun Kim. Deep code search. Em *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18,

páginas 933–944, New York, NY, USA. ACM. ISBN 978-1-4503-5638-1. doi: 10.1145/3180155.3180167. URL http://doi.acm.org/10.1145/3180155.3180167. Citado na pág. 1, 11, 12, 13, 14, 15, 33

Hinton *et al.*(1986) G. E. Hinton, J. L. McClelland e D. E. Rumelhart. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter Distributed Representations, páginas 77–109. MIT Press, Cambridge, MA, USA. ISBN 0-262-68053-X. URL http://dl.acm.org/citation.cfm?id=104279.104287. Citado na pág. vi

Iyer *et al.*(2016) Srinivasan Iyer, Ioannis Konstas, Alvin Cheung e Luke Zettlemoyer. Summarizing source code using a neural attention model. Em *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, páginas 2073–2083, Berlin, Germany. Association for Computational Linguistics. doi: 10.18653/v1/P16-1195. URL https://www.aclweb.org/anthology/P16-1195. Citado na pág. xi, 1, 3, 6, 12, 13, 14, 15, 16, 29, 32, 39, 46

Jupyter(2019) Jupyter. Jupyter notebook, 2019. URL https://jupyter-notebook.readthedocs.io/en/latest/. Citado na pág. v

Khadder(2019) Simba Khadder. How does t-sne work in simple words?, 2019. URL https://www.quora.com/How-does-t-SNE-work-in-simple-words. Citado na pág. 34

Kim(2019) Joshua Kim. Understanding how convolutional neural network (cnn) perform text classification with word embeddings, 2019. URL http://www.joshuakim.io/understanding-how-convolutional-neural-network-cnn-perform-text-classification-with-word-embeddings/. Citado na pág. ix, 21, 22

Lai *et al.*(2018) Tuan Manh Lai, Trung Bui e Sheng Li. A review on deep learning techniques applied to answer selection. Em *Proceedings of the 27th International Conference on Computational Linguistics*, páginas 2132–2144, Santa Fe, New Mexico, USA. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/C18-1181. Citado na pág. 2, 11, 13

Mikolov *et al.***(2013)** Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado e Jeff Dean. Distributed representations of words and phrases and their compositionality. Em *NIPS*, páginas 3111–3119. Curran Associates, Inc. URL http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf. Citado na pág. 7, 18

Sachdev *et al.***(2018)** Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen e Satish Chandra. Retrieval on source code: A neural code search. Em *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, páginas 31–41, New York, NY, USA. ACM. ISBN 978-1-4503-5834-7. doi: 10.1145/3211346.3211353. URL http://doi.acm.org/10.1145/3211346.3211353. Citado na pág. 1, 13, 14, 15

scikit learn(2019) scikit learn. sklearn.manifold.tsne, 2019. URL https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html. Citado na pág. 34

Tan *et al.***(2015)** Ming Tan, Cicero dos Santos, Bing Xiang e Bowen Zhou. Lstm-based deep learning models for non-factoid answer selection. *CoRR*, abs/1511.04108. URL http://arxiv.org/abs/1511.04108. Citado na pág. 10, 13, 33, 35

Wikipedia(2019a) Wikipedia. Docstring, 2019a. URL https://en.wikipedia.org/wiki/Docstring. Citado na pág. v

Wikipedia(2019b) Wikipedia. Git, 2019b. URL https://en.wikipedia.org/wiki/Git. Citado na pág. v

Yao *et al.***(2018)** Ziyu Yao, Daniel S. Weld, Wei-Peng Chen e Huan Sun. Staqc: A systematically mined question-code dataset from stack overflow. Em *Proceedings of the 2018 World Wide Web Conference*, WWW '18, páginas 1693–1703, Republic and Canton of Geneva, Switzerland. International World Wide Web Conferences Steering Committee. ISBN 978-1-4503-5639-8. doi: 10.1145/3178876.3186081. URL https://doi.org/10.1145/3178876.3186081. Citado na pág. ix, xi, 1, 3, 4, 12, 13, 16, 26, 29, 30, 31, 32, 33, 34, 39, 43

Yao *et al.*(2019) Ziyu Yao, Jayavardhan Reddy Peddamail e Huan Sun. Coacor: Code annotation for code retrieval with reinforcement learning. Em *The World Wide Web Conference*, WWW '19, páginas 2203–2214, New York, NY, USA. ACM. ISBN 978-1-4503-6674-8. doi: 10.1145/3308558.3313632. URL http://doi.acm.org/10.1145/3308558.3313632. Citado na pág. 6

Young *et al.*(2017) Tom Young, Devamanyu Hazarika, Soujanya Poria e Erik Cambria. Recent trends in deep learning based natural language processing. *CoRR*, abs/1708.02709. URL http://arxiv.org/abs/1708.02709. Citado na pág. 2, 10, 11, 22, 25

Zhang *et al.*(2019) Shuai Zhang, Lina Yao, Aixin Sun e Yi Tay. Deep learning based recommender system: A survey and new perspectives. *ACM Comput. Surv.*, 52(1): 5:1–5:38. ISSN 0360-0300. doi: 10.1145/3285029. URL http://doi.acm.org/10.1145/3285029. Citado na pág. 2, 11

Zhang e Wallace(2015) Ye Zhang e Byron C. Wallace. A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. *CoRR*, abs/1510.03820. URL http://arxiv.org/abs/1510.03820. Citado na pág. ix, 23